# 14 Templates [temp]

1 A *template* defines a family of types or functions.

*template-declaration*:
  `export`$_{opt}$ `template <` *template-parameter-list* `>` *declaration*

*template-parameter-list*:
  *template-parameter*
  *template-parameter-list* `,` *template-parameter*

The *declaration* in a *template-declaration* shall

— declare or define a function or a class, or

— define a member function, a member class or a static member of a class template or of a class nested within a class template, or

— define a member template of a class or class template.

A *template-declaration* is a *declaration*. A *template-declaration* is also a definition if its *declaration* defines a function, a class, or a static data member.

2 A *template-declaration* can appear only as a namespace scope or class scope declaration. In a function template declaration, the *declarator-id* shall be a *template-name* (i.e., not a *template-id*). [*Note:* in a class template declaration, if the *declarator-id* is a *template-id*, the declaration declares a class template partial specialization (14.5.4). ]

3 In a *template-declaration*, explicit specialization, or explicit instantiation the *init-declarator-list* in the declaration shall contain at most one declarator. When such a declaration is used to declare a class, no declarator is permitted.

4 A template name may have linkage (_basic.link_). A template, a template explicit specialization (14.7.3), or a class template partial specialization shall not have C linkage. If the linkage of one of these is something other than C or C++, the behavior is implementation-defined. Template definitions shall obey the one definition rule (_basic.def.odr_).

5 The name of a class template shall not be declared to refer to any other template, class, function, object, enumeration, enumerator, namespace, or type in the same scope (_basic.scope_). Except that a function template can be overloaded either by (non-template) functions with the same name or by other function templates with the same name (14.8.3), a template name declared in namespace scope or in class scope shall be unique in that scope.

6 A non-inline function template, a non-inline member function template, a non-inline member function of a class template or a static data member of a class 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 non-inline function members, static data members, member classes, and non-inline member templates which are defined in that translation unit exported.

7 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); no diagnostic is required. 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.

8 [*Note:* an implementation may require that a translation unit containing the definition of an exported template be compiled before any translation unit containing an instantiation of that template. ]

| **Box 1** |
|---|
| More work is needed to describe the semantics of `export` and exported templates. See core issues 814 and 878. |

### 14.1 Template parameters [temp.param]

1 The syntax for *template-parameter*s is:

*template-parameter:*
  *type-parameter*
  *parameter-declaration*

*type-parameter:*
  `class` *identifier*$_{opt}$
  `class` *identifier*$_{opt}$ = *type-id*
  `typename` *identifier*$_{opt}$
  `typename` *identifier*$_{opt}$ = *type-id*
  `template` < *template-parameter-list* > `class` *identifier*$_{opt}$
  `template` < *template-parameter-list* > `class` *identifier*$_{opt}$ = *template-name*

There is no semantic difference between `class` and `typename` in a *template-parameter*. `typename` followed by an *unqualified-id* names a template type parameter. `typename` followed by a *qualified-name* denotes the type in a non-type *parameter-declaration*. A storage class shall not be specified in a *template-parameter* declaration. [*Note:* a template parameter may be a class template. For example,

```
template<class T> class myarray { /* ... */ };

template<class K, class V, template<class T> class C = myarray>
class Map {
        C<K> key;
        C<V> value;
        // ...
};
```

—*end note*]

2 A *type-parameter* defines its *identifier* to be a *type-name* (if declared with `class` or `typename`) or *template-name* (if declared with `template`) in the scope of the template declaration. [*Note:* because of the name look up rules, a *template-parameter* that could be interpreted as either a non-type *template-parameter* or a *type-parameter* (because its *identifier* is the name of an already existing class) is taken as a *type-parameter*. For example,

```
class T { /* ... */ };
int i;

template<class T, T i> void f(T t)
{
        T t1 = i;       // template-parameters T and i
        ::T t2 = ::i;   // global namespace members T and i
}
```

Here, the template `f` has a *type-parameter* called `T`, rather than an unnamed non-type *template-parameter* of class `T`. ]

3 A non-type *template-parameter* shall have one of the following (optionally *cv-qualified*) types:

— integral type, accepting an integral constant expression as an argument,

— enumeration type, accepting an integral constant expression as an argument,

— pointer to object, accepting an address constant expression designating a named object with external linkage,

— reference to object, accepting an lvalue expression designating a named object with external linkage,

— pointer to function, accepting an expression of type pointer to function designating a function with external linkage,

— reference to function, accepting an lvalue expression designating a function with external linkage,

— pointer to member, accepting an address constant expression designating a named member of a class.

4 [*Note:* other types are disallowed either explicitly below or implicitly by the rules governing the form of *template-argument*s (14.3). ] The top-level *cv-qualifiers* on the *template-parameter* are ignored when determining its type.

5 A non-type non-reference *template-parameter* is not an lvalue. It shall not be assigned to or in any other way have its value changed. A non-type non-reference *template-parameter* cannot have its address taken. When a non-type non-reference *template-parameter* is used as an initializer for a reference, a temporary is always used. [*Example:*

```
template<const X& x, int i> void f()
{
        i++; // error: change of template-parameter value

        &x; // ok
        &i; // error: address of non-reference template-parameter

        int& ri = i; // error: non-const reference bound to temporary
        const int& cri = i; // ok: const reference bound to temporary
}
```

—*end example*]

6 A non-type *template-parameter* shall not be of type `void`. A non-type *template-parameter* shall not be of floating type. [*Example:*

```
template<double d> class X;    // error
template<double* pd> class Y;  // ok
template<double& rd> class Z;  // ok
```

—*end example*]

7 Any non-type *template-parameter* of type “array of `T`” or “function returning `T`” is adjusted to be of type “pointer to `T`” or “pointer to function returning `T`”, respectively. [*Example:*

```
template<int a[5]> struct S { /* ... */ };
int v[5];
int* p = v;
S<v> x; // fine
S<p> y; // also fine
```

—*end example*]

8 A *default template-argument* is a type, value, or template specified after = in a *template-parameter*. A default *template-argument* may be specified for both a type and non-type *template-parameter*. A default *template-argument* may be specified in a class template declaration or a class template definition. A default *template-argument* shall not be specified in a function template declaration or a function template definition. The set of default *template-argument*s available for use with a template in a translation unit shall only be provided by the first declaration of the template in that translation unit.

9 If a *template-parameter* has a default *template-argument,* all subsequent *template-parameter*s shall have a default *template-argument* supplied. [*Example:*

```
template<class T1 = int, class T2> class B; // error
```

—*end example*]

10 When parsing a *default template-argument* for a non-type *template-parameter*, the first non-nested > is taken as the end of the *template-parameter-list* rather than a greater-than operator. [*Example:*

```
template<int i = 3 > 4 >    // syntax error
  class X { /* ... */ };

template<int i = (3 > 4) >  // ok
  class Y { /* ... */ };
```

—*end example*]

### 14.2 Names of template specializations [temp.names]

1 A template specialization (14.7) can be referred to by a *template-id*:

*template-id*:
  *template-name* < *template-argument-list*$_{opt}$ >

*template-name*:
  *identifier*

*template-argument-list*:
  *template-argument*
  *template-argument-list* , *template-argument*

*template-argument*:
  *assignment-expression*
  *type-id*
  *template-name*

[*Note:* the name look up rules (_basic.lookup_) are used to associate the use of a name with a template declaration; that is, to identify a name as a *template-name*. ]

2 For a *template-name* to be explicitly qualified by the template arguments, the name must be known to refer to a template.

3 After name look up (_basic.lookup_) finds that a name is a *template-name*, if this name is followed by a <, the < is always taken as the beginning of a *template-argument-list* and never as a name followed by the less-than operator. When parsing a *template-id*, the first non-nested >[1] is taken as the end of the *template-argument-list* rather than a greater-than operator. [*Example:*

```
template<int i> class X { /* ... */ };

X< 1>2 >        x1; // syntax error
X<(1>2)>        x2; // ok

template<class T> class Y { /* ... */ };
Y< X<1> >       x3; // ok
Y<X<6>> 1> >    x4; // ok: Y< X< (6>>1) > >
```

—*end example*]

[1] A > that encloses the *type-id* of a `dynamic_cast`, `static_cast`, `reinterpret_cast` or `const_cast` is considered nested for the purpose of this description.

4 When the name of a member template specialization appears after `.` or `->` in a *postfix-expression*, or after `::` in a *qualified-id* that explicitly depends on a template-argument (14.6.2), the member template name must be prefixed by the keyword template. Otherwise the name is assumed to name a non-template. [*Example:*

```
class X {
public:
        template<size_t> X* alloc();
};
void f(X* p)
{
        X* p1 = p->alloc<200>();
                // ill-formed: < means less than

        X* p2 = p->template alloc<200>();
                // fine: < starts explicit qualification
}
```

—*end example*]

**Box 2**

core issue 883: Can `template` be used to specify that an unqualified function name refers to a template specialization?

```
namespace A {
        struct B { };
        template<class T> void f(T t);
}
void g(A::B b) {
        f<3>(b); // Is A::f considered?
}
```

Does type-dependent (Koenig) lookup apply to the lookup of `f`? Without explicit template arguments, the answer is currently yes, because the lookup of `f` (other than to determine whether it is a type) can be deferred until after the function arguments have been parsed. But with explicit template arguments, there is no way to parse the expression without knowing that `f` is a template.

5 If a name prefixed by the keyword `template` is not the name of a member template, the program is ill-formed. [*Note:* the keyword `template` may not be applied to non-template members of class templates. ]

6 A *template-id* that names a class template specialization is a *class-name* (_class_).

### 14.3 Template arguments [temp.arg]

1 The types of the *template-argument*s specified in a *template-id* shall match the types specified for the template in its *template-parameter-list*. [*Example:*

```
template<class T> class Array {
        T* v;
        int sz;
public:
        explicit Array(int);
        T& operator[](int);
        T& elem(int i) { return v[i]; }
        // ...
};
```

```
Array<int> v1(20);
typedef complex<double> dcomplex; // complex is a standard
                                  // library template
Array<dcomplex> v2(30);
Array<dcomplex> v3(40);

void bar() {
        v1[3] = 7;
        v2[3] = v3.elem(4) = dcomplex(7,8);
}
```

—*end example*] If the use of a *template-argument* gives rise to an ill-formed construct in the implicit instantiation of a template specialization, the instantiation is ill-formed.

2 In a *template-argument*, an ambiguity between a *type-id* and an expression is resolved to a *type-id*. [*Example:*

```
template<class T> void f();
template<int I> void f();

void g()
{
        f<int()>(); // ``int()'' is a type-id: call the first f()
}
```

—*end example*]

3 A *template-argument* for a non-type non-reference *template-parameter* shall be an integral constant-expression of integral or enumeration type, the name of a non-type non-reference template parameter, the address of an object or a function with external linkage, or a non-overloaded pointer to member. The address of an object or function shall be expressed as `&f` (except when `f` is a function or an array in which case it can be expressed as `f`), or `&X::f` where `f` is the function or object name. In the case of `&X::f`, `X` shall be a (possibly qualified) name of a class and `f` the name of a static member of `X`. A pointer to member shall be expressed as `&X::m` where `X` is a (possibly qualified) name of a class and `m` is the name of a non-static member of `X`. In particular, a string literal (_lex.string_) is not an acceptable *template-argument* because a string literal is an object with internal linkage. [*Example:*

```
template<class T, char* p> class X {
        // ...
        X();
        X(const char* q) { /* ... */ }
};

X<int,"Studebaker"> x1; // error: string literal as template-argument

char p[] = "Vivisectionist";
X<int,p> x2; // ok
```

—*end example*]

4 Addresses of array elements and of non-static class members shall not be used as *template-argument*s. [*Example:*

```
template<int* p> class X { };

int a[10];
struct S { int m; static int s; } s;

X<&a[2]> x3; // error: address of array element
X<&s.m> x4;  // error: address of non-static member
X<&s.s> x5;  // error: &S::s must be used
X<&S::s> x6; // ok: address of static member
```

—*end example*]

5 A *template-argument* for a non-type *template-parameter* that is a reference shall be the name of an object or function with external linkage. A non-type *template-parameter* that is a reference shall not be bound to a temporary, an unnamed lvalue, or a named lvalue that does not have external linkage. [*Example:*

```
template<const int& CRI> struct B { /* ... */ };

B<1> b2; // error: temporary required for template argument

int c = 1;
B<c> b1; // ok
```

—*end example*]

6 For a non-type *template-parameter* of integral or enumeration type, integral promotions (_conv.prom_) and integral conversions (_conv.integral_) are applied to an expression used as a *template-argument* to bring it to the type of its corresponding *template-parameter*. For a non-type *template-parameter* of pointer type, qualification conversions (_conv.qual_), the derived-to-base pointer conversions and the pointer conversions to `void*` (_conv.ptr_) are applied to an expression used as a *template-argument* to bring it to the type of its corresponding *template-parameter*. For a non-type *template-parameter* of pointer to member type, qualification conversions (_conv.qual_) and the base-to-derived pointer to member conversions (_conv.mem_) are applied to an expression used as a *template-argument* to bring it to the type of its corresponding *template-parameter*. For a non-type *template-parameter* of reference type, the expression used as a *template-argument* shall be reference-compatible (_dcl.init.ref_) with the type of the corresponding *template-parameter*. [*Example:*

```
template<const int* pci> struct X { /* ... */ };
int ai[10];
X<ai> xi;  // array to pointer and qualification conversions

struct Base { /* ... */ };
struct Derived : Base { /* ... */ };
template<Base& b> struct Y { /* ... */ };
Derived d;
Y<d> yd;   // template-argument is reference-compatible
           // with template-parameter
```

—*end example*]

7 An argument to a non-type *template-parameter* of pointer to function type shall have exactly the type specified by the *template-parameter*. [*Note:* this allows selection from a set of overloaded functions. ] [*Example:*

```
void f(char);
void f(int);

template<void (*pf)(int)> struct A { /* ... */ };

A<&f> a; // selects f(int)
```

—*end example*]

8 If a declaration acquires a function type through a *template-argument* of function type and this causes a declaration that does not use the syntactic form of a function declarator to have function type, the program is ill-formed. [*Example:*

```
template<class T> struct A {
        static T t;
};
typedef int function();
A<function> a;  // ill-formed: would declare A<function>::t
                // as a static member function
```

—*end example*]

9 A local type, a type with no linkage, an unnamed type or a type compounded from any of these types shall not be used as a *template-argument* for a template *type-parameter*. [*Example:*

```
template <class T> class X { /* ... */ };
void f()
{
  struct S { /* ... */ };

  X<S> x3;  // error: local type used as template-argument
  X<S*> x4; // error: pointer to local type used as template-argument
}
```

—*end example*]

10 The name of a *template-argument* shall be accessible at the point where it is used as a *template-argument*. [*Note:* if the name of the *template-argument* is accessible at the point where it is used as a *template-argument*, there is no further access restriction in the resulting instantiation where the corresponding *template-parameter* name is used. ] [*Example:*

```
template<class T> class X {
        static T t;
};

class Y {
private:
        struct S { /* ... */ };
        X<S> x;  // ok: S is accessible
                 // X<Y::S> has a static member of type Y::S
                 // ok: even though Y::S is private
};

X<Y::S> y; // error: S not accessible
```

—*end example*] For a *template-argument* of class type, the template definition has no special access rights to the inaccessible members of the template argument type.

11 When default *template-argument*s are used, a *template-argument* list can be empty. In that case the empty <> brackets shall still be used as the *template-argument-list.* [*Example:*

```
template<class T = char> class String;
String<>* p; // ok: String<char>
String* q;   // syntax error
```

—*end example*]

12 An explicit destructor call (_class.dtor_) for an object that has a type that is a class template specialization may explicitly specify the *template-argument*s. [*Example:*

```
template<class T> struct A {
        ~A();
};
int main() {
        A<int>* p;
        p->A<int>::~A(); // ok: destructor call
        p->A<int>::~A<int>(); // ok: destructor call
}
```

—*end example*]

### 14.4 Type equivalence [temp.type]

1 Two *template-id*s refer to the same class or function if their template names are identical, they refer to the same template, their type *template-argument*s are the same type and, their non-type *template-argument*s have identical values. [*Example:*

```
template<class E, int size> class buffer { /* ... */ };
buffer<char,2*512> x;
buffer<char,1024> y;
```

declares `x` and `y` to be of the same type, and

```
template<class T, void(*err_fct)()> class list { /* ... */ };
list<int,&error_handler1> x1;
list<int,&error_handler2> x2;
list<int,&error_handler2> x3;
list<char,&error_handler2> x4;
```

declares `x2` and `x3` to be of the same type. Their type differs from the types of `x1` and `x4`. ]

### 14.5 Template declarations [temp.decls]

1 A *template-id*, that is, the *template-name* followed by a *template-argument-list* shall not be specified in the declaration of a primary template declaration. [*Example:*

```
template<class T1, class T2, int I> class A<T1, T2, I> { }; // error
template<class T1, int I> void sort<T1, I>(T1 data[I]);     // error
```

—*end example*] [*Note:* however, this syntax is allowed in class template partial specializations (14.5.4). ]

#### 14.5.1 Class templates [temp.class]

1 A class *template* defines the layout and operations for an unbounded set of related types. [*Example:* a single class template `List` might provide a common definition for list of `int`, list of `float`, and list of pointers to `Shape`s. ]

2 [*Example:* An array class template might be declared like this:

```
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};
```

The prefix `template <class T>` specifies that a template is being declared and that a *type-name* `T` will be used in the declaration. In other words, `Array` is a parameterized type with `T` as its parameter. ]

3 When a member function, a member class, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the *template-parameter*s are those of the class template. The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. [*Example:*

```
template<class T1, class T2> struct A {
    void f1();
    void f2();
};

template<class T2, class T1> void A<T2,T1>::f1() { } //ok
template<class T2, class T1> void A<T1,T2>::f2() { } //error
```

—*end example*] *

### 14.5.1.1 Member functions of class templates [temp.mem.func]

1 A member function template may be defined outside of the class template definition in which it is declared. [*Example:* *

```
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};
```

declares three function templates. The subscript function might be defined like this:

```
template<class T> T& Array<T>::operator[](int i)
{
    if (i<0 || sz<=i) error("Array: range error");
    return v[i];
}
```

—*end example*]

2 The *template-arguments* for a member function of a class template are determined by the *template-arguments* of the type of the object for which the member function is called. [*Example:* the *template-argument* for `Array<T>::operator[]()` will be determined by the `Array` to which the subscripting operation is applied. |

```
Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7;               // Array<int>::operator[]()
v2[3] = dcomplex(7,8);   // Array<dcomplex>::operator[]()
```

—*end example*]

### 14.5.1.2 Member classes of class templates [temp.mem.class]

1 A class member of a class template may be defined outside the class template definition in which it is declared. [*Note:* the class member must be defined before its first use that requires a an instantiation (14.7.1). For example, |

```
template<class T> struct A {
        class B;
};
A<int>::B* b1;  // ok: requires A to be defined but not A::B
template<class T> class A<T>::B { };
A<int>::B  b2;  // ok: requires A::B to be defined
```

—*end note*]

### 14.5.1.3 Static data members of class templates [temp.static]

1 A definition for a static data member may be provided in a namespace scope enclosing the definition of the static member's class template. [*Example:*

```
template<class T> class X {
        static T s;
};
template<class T> T X<T>::s = 0;
```

—*end example*]

### 14.5.2 Member templates [temp.mem]

1 A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class template definition shall be specified with the *template-parameter*s of the class template followed by the *template-parameter*s of the member template. [*Example:*

```
template<class T> class string {
public:
        template<class T2> int compare(const T2&);
        template<class T2> string(const string<T2>& s) { /* ... */ }
        // ...
};

template<class T> template<class T2> int string<T>::compare(const T2& s)
{
        // ...
}
```

—*end example*]

2 A local class shall not have member templates. Access control rules (_class.access_) apply to member template names. A destructor shall not be a member template. A normal (non-template) member function with a given name and type and a member function template of the same name, which could be used to generate a specialization of the same type, can both be declared in a class. When both exist, a reference refers to the non-template unless an explicit template argument list is supplied. [*Example:*

```
template <class T> struct A {
        void f(int);
        template <class T2> void f(T2);
};

template <> void A<int>::f(int) { }  // non-template member
template <> template <> void A<int>::f<>(int) { }  // template member

int main()
{
        A<char> ac;
        ac.f(1);    // non-template
        ac.f('c');  // template
        ac.f<>(1);  // template
}
```

—*end example*]

3 A member function template shall not be virtual. [*Example:*

```
template <class T> struct AA {
        template <class C> virtual void g(C); // error
        virtual void f(); // ok
};
```

—*end example*]

4 A specialization of a member function template does not override a virtual function from a base class. [*Example:*

```
class B {
        virtual void f(int);
};

class D : public B {
        template <class T> void f(T);  // does not override B::f(int)
        void f(int i) { f<>(i); }  // overriding function that calls
                                   // the template instantiation
};
```

—*end example*]

5 A specialization of a template conversion operator is referenced in the same way as a non-template conversion operator that converts to the same type. [*Example:*

```
struct A {
        template <class T> operator T*();
};
template <class T> A::operator T*(){ return 0; }
template <> A::operator char*(){ return 0; }  // specialization
template A::operator void*();                 // explicit instantiation

int main()
{
        A      a;
        int*   ip;

        ip = a.operator int*();  // explicit call to template operator
                                 // A::operator int*()
}
```

—*end example*] [*Note:* because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates. ]

6 If more than one conversion template can produce the required type, the partial ordering rules (14.5.5.2) are used to select the "most specialized" version of the template that can produce the required type. As with other conversion functions, the type of the implicit `this` parameter is not considered. [*Note:* members of base classes are considered equally with members of the derived class, except that a derived class conversion function hides a base class conversion function that converts to the same type. —*end note*]

### 14.5.3 Friends [temp.friend]

1 A friend of a class template can be a template, a specialization of a function template or class template, or an ordinary (non-template) function or class. For a friend declaration that is not a template declaration, if the name of the friend has explicit *template-argument*s, the friend declaration refers to the specialization(s) of a class or function template; otherwise, the friend refers to an ordinary class or function. [*Example:*

```
template<class T> class task {
    // ...
    friend void next_time();
    friend void process(task<T>*);
    friend task<T>* preempt<T>(task<T>*);
    template<class C> friend int func(C);

    friend class task<int>;
    template<class P> friend class frd;
    // ...
};
```

Here, each specialization of the `task` class template has the function `next_time` as a friend; because `process` does not have explicit *template-argument*s, each specialization of the `task` class template has an appropriately typed function `process` as a friend, and this friend is not a function template specialization; because the friend `preempt` has an explicit *template-argument* `<T>`, each specialization of the `task` class template has the appropriate specialization of the function template `preempt` as a friend; and each specialization of the `task` class template has all specializations of the function template `func` as friends. Similarly, each specialization of the `task` class template has the class template specialization `task<int>` as a friend, and has all specializations of the class template `frd` as friends. ]

**Box 3**

core issue 890: John Spicer believes that if the friend declaration has a declarator that is qualified, then the friend declaration should be able to refer to a previously declared template even though explicit template arguments are not specified.

2 A friend template may be declared within a non-template class. A friend function template may be defined within a non-template class. In these cases, all specializations of the class or function template are friends of the class granting friendship. [*Example:*

```
class A {
        template<class T> friend class B; // ok
        template<class T> friend void f(T){ /* ... */ } // ok
};
```

—*end example*]

3 A template friend declaration specifies that all specializations of that template, whether they are implicitly instantiated (14.7.1), partially specialized (14.5.4) or explicitly specialized (14.7.3), are friends of the class containing the template friend declaration. [*Example:*

```
class X {
        template<class T> friend struct A;
        class Y { };
};

template<class T> struct A { X::Y ab; }; //ok
template<class T> struct A<T*> { X::Y ab; }; //ok
```

—*end example*]

4 When a function is defined in a friend function declaration in a class template, the function is defined when the class template is first instantiated. The function is defined even if it is never used. [*Note:* if the function definition is ill-formed for a given specialization of the enclosing class template, the program is ill-formed even if the function is never used. ]

5 A member of a class template may be declared to be a friend of a non-template class. In this case, the corresponding member of every specialization of the class template is a friend of the class granting friendship. [*Example:*

```
template<class T> struct A {
        struct B { };
        void f();
};

class C {
        template<class T> friend struct A<T>::B;
        template<class T> friend void A<T>::f();
};
```

—*end example*]

6 [*Note:* a friend declaration may first declare a member of an enclosing namespace scope (14.6.5). ]

7 A friend template shall not be declared in a local class.

8 Friend declarations shall not declare partial specializations. [*Example:*

```
template<class T> class A { };
class X {
        template<class T> friend class A<T*>; // error
};
```

—*end example*]

### 14.5.4 Class template partial specializations [temp.class.spec]

1 A *primary* class template declaration is one in which the class template name is an identifier. A template declaration in which the class template name is a *template-id*, is a *partial specialization* of the class template named in the *template-id*. A partial specialization of a class template provides an alternative definition of the template that is used instead of the primary definition when the arguments in a specialization match those given in the partial specialization (14.5.4.1). The primary template shall be declared before any specializations of that template. If a template is partially specialized then that partial specialization shall be declared before the first use of that partial specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs. Each class template partial specialization is a distinct template and definitions shall be provided for the members of a template partial specialization (14.5.4.3).

2 [*Example:*

```
template<class T1, class T2, int I> class A             { }; // #1
template<class T, int I>            class A<T, T*, I>   { }; // #2
template<class T1, class T2, int I> class A<T1*, T2, I> { }; // #3
template<class T>                   class A<int, T*, 5> { }; // #4
template<class T1, class T2, int I> class A<T1, T2*, I> { }; // #5
```

The first declaration declares the primary (unspecialized) class template. The second and subsequent declarations declare partial specializations of the primary template. ]

3 The template parameters are specified in the angle bracket enclosed list that immediately follows the keyword `template`. For partial specializations, the template argument list is explicitly written immediately following the class template name. For primary templates, this list is implicitly described by the template parameter list. Specifically, the order of the template arguments is the sequence in which they appear in the template parameter list. [*Example:* the template argument list for the primary template in the example above is `<T1, T2, I>`. ] [*Note:* the template argument list shall not be specified in the primary template declaration. For example,

```
template<class T1, class T2, int I> class A<T1, T2, I>  { }; // error
```

—*end note*]

4 A class template partial specialization shall be declared in the class or namespace containing the primary template declaration or in a namespace of which the class or namespace containing the primary template declaration is a member. [*Example:*

```
template<class T> struct A {
        template<class T2> struct B { };
};

template<class T> template<class T2>
    struct A<T>::B<T2*> { }; // partial specialization of A<T>::B<T2>
A<short>::B<int*> absip;     // uses partial specialization
```

—*end example*]

5 A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

6 Within the argument list of a class template partial specialization, the following restrictions apply:

— A partially specialized non-type argument expression shall not involve a template parameter of the partial specialization except when the argument expression is a simple *identifier*. [*Example:*

```
template <int I, int J> struct A {};
template <int I> struct A<I+5, I*2> {}; // error

template <int I, int J> struct B {};
template <int I> struct B<I, I> {};     // ok
```

—*end example*]

— The type of a specialized nontype argument shall not be dependent on another type parameter of the specialization. [*Example:*

```
template <class T, T t> struct C {};
template <class T> struct C<T, 1>;  // error
```

—*end example*]

— The argument list of the specialization shall not be identical to the implicit argument list of the primary template.

7 The template parameter list of a specialization shall not contain default template argument values.[2]

### 14.5.4.1 Matching of class template partial specializations [temp.class.spec.match]

1 When a class template is used in a context that requires an instantiation of the class, it is necessary to determine whether the instantiation is to be generated using the primary template or one of the partial specializations. This is done by matching the template arguments of the class template specialization with the template argument lists of the partial specializations.

— If exactly one matching specialization is found, the instantiation is generated from that specialization.

— If more than one matching specialization is found, the partial order rules (14.5.4.2) are used to determine whether one of the specializations is more specialized than the others. If none of the specializations is more specialized than all of the other matching specializations, then the use of the class template is ambiguous and the program is ill-formed.

— If no matches are found, the instantiation is generated from the primary template.

2 A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.8.2). [*Example:*

[2] There is no way in which they could be used.

```
A<int, int, 1>   a1;  // uses #1
A<int, int*, 1>  a2;  // uses #2, T is int, I is 1
A<int, char*, 5> a3;  // uses #4, T is char
A<int, char*, 1> a4;  // uses #5, T1 is int, T2 is char, I is 1
A<int*, int*, 2> a5;  // ambiguous: matches #3 and #5
```

—*end example*]

3 A non-type template argument can also be deduced from the value of an actual template argument of a non-type parameter of the primary template. [*Example:* the declaration of `a2` above. ]

4 In a type name that refers to a class template specialization, (e.g., `A<int, int, 1>`) the argument list must match the template parameter list of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template.

### 14.5.4.2 Partial ordering of class template specializations [temp.class.order]

1 For two class template partial specializations, the first is at least as specialized as the second if, given the following rewrite to two function templates, the first function template is at least as specialized as the second according to the ordering rules for function templates (14.5.5.2):

— the first function template has the same template parameters as the first partial specialization and has a single function parameter whose type is a class template with the template arguments of the first partial specialization, and

— the second function template has the same template parameters as the second partial specialization and has a single function parameter whose type is a class template with the template arguments of the second template specialization.

2 [*Example:*

```
template<int I, int J, class T> class X { };
template<int I, int J>          class X<I, J, int> { }; // #1
template<int I>                 class X<I, I, int> { }; // #2

template<int I, int J> void f(X<I, J, int>); // #A
template<int I>        void f(X<I, I, int>); // #B
```

The partial specialization `#2` is more specialized than the partial specialization `#1` because the function template `#B` is more specialized than the function template `#A` according to the ordering rules for function templates. ]

### 14.5.4.3 Members of class template specializations [temp.class.spec.mfunc]

1 The template parameter list of a member of a class template partial specialization shall match the template parameter list of the class template partial specialization. The template argument list of a member of a class template partial specialization shall match the template argument list of the class template partial specialization. A class template specialization is a distinct template. The members of the class template partial specialization are unrelated to the members of the primary template. Class template partial specialization members that are used in a way that requires a definition shall be defined; the definitions of members of the primary template are never used as definitions for members of a class template partial specialization. An explicit specialization of a member of a class template partial specialization is declared in the same way as an explicit specialization of the primary template. [*Example:*

```
// primary template
template<class T, int I> struct A {
        void f();
};

template<class T, int I> void A<T,I>::f() { }
```

```
// class template partial specialization
template<class T> struct A<T,2> {
        void f();
        void g();
        void h();
};

// member of class template partial specialization
template<class T> void A<T,2>::g() { }

// explicit specialization
template<> void A<char,2>::h() { }

int main()
{
        A<char,0> a0;
        A<char,2> a2;
        a0.f(); // ok, uses definition of primary template's member
        a2.g(); // ok, uses definition of
                // partial specialization's member
        a2.h(); // ok, uses definition of
                // explicit specialization's member
        a2.f(); // ill-formed, no definition of f for A<T,2>
                // the primary template is not used here
}
```

—*end example*]

2 If a member template of a class template is partially specialized, the member template partial specializations are member templates of the enclosing class template; if the enclosing class template is instantiated (14.7.1, 14.7.2), a declaration for every member template partial specialization is also instantiated as part of creating the members of the class template specialization. If the primary member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the partial specializations of the member template are ignored for this specialization of the enclosing class template. If a partial specialization of the member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the primary member template and its other partial specializations are still considered for this specialization of the enclosing class template. [*Example:*

```
template<class T> struct A {
        template<class T2> struct B {};      // #1
        template<class T2> struct B<T2*> {}; // #2
};

template<> template<class T2> struct A<short>::B {}; // #3

A<char>::B<int*>  abcip; // uses #2
A<short>::B<int*> absip; // uses #3
A<char>::B<int>  abci;   // uses #1
```

—*end example*]

### 14.5.5 Function templates [temp.fct]

1 A function template defines an unbounded set of related functions. [*Example:* a family of sort functions might be declared like this:

```
template<class T> class Array { };
template<class T> void sort(Array<T>&);
```

—*end example*]

2 A function template can be overloaded with other function templates and with normal (non-template) functions. A normal function is not related to a function template (i.e., it is never considered to be a specialization), even if it has the same name and type as a potentially generated function template specialization.[3)]

### 14.5.5.1 Function template overloading [temp.over.link]

1 It is possible to overload function templates so that two different function template specializations have the same type. [*Example:*

```
// file1.c
template<class T>
    void f(T*);
void g(int* p) {
    f(p); // call
          // f<int>(int*)
}
```

```
// file2.c
template<class T>
    void f(T);
void h(int* p) {
    f(p); // call
          // f<int*>(int*)
}
```

—*end example*]

2 Such specializations are distinct functions and do not violate the one definition rule (_basic.def.odr_).

3 The signature of a function template specialization consists of the signature of the function template and of the actual template arguments (whether explicitly specified or deduced).

4 The signature of a function template consists of its function signature, its return type and its template parameter list. The names of the template parameters are significant only for establishing the relationship between the template parameters and the rest of the signature. [*Note:* two distinct function templates may have identical function return types and function parameter lists, even if overload resolution alone cannot distinguish them.

```
template<class T> void f();
template<int I> void f(); //ok: overloads the first template
```

—*end note*]

### 14.5.5.2 Partial ordering of function templates [temp.func.order]

1 If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (14.8.2) may associate the function template specialization with more than one function template declaration. *Partial ordering* of overloaded function template declarations is used in the following contexts to select the function template to which a function template specialization refers:

— during overload resolution for a call to a function template specialization (_over.match.best_);

— when the address of a function template specialization is taken;

— when a placement operator delete that is a template function specialization is selected to match a placement operator new (_basic.stc.dynamic.deallocation_, _expr.new_);

— when a friend function declaration (14.5.3), an explicit instantiation (14.7.2) or an explicit specialization (14.7.3) refers to a function template specialization.

2 Given two overloaded function templates, whether one is more specialized than another can be determined by transforming each template in turn and using argument deduction (14.8.2) to compare it to the other.

3 The transformation used is:

— For each type template parameter, synthesize a unique type and substitute that for each occurrence of

[3)] That is, declarations of non-template functions do not merely guide overload resolution of template functions with the same name. If such a non-template function is used in a program, it must be defined; it will not be implicitly instantiated using the function template definition.

that parameter in the function parameter list.

— For each non-type template parameter, synthesize a unique value of the appropriate type and substitute that for each occurrence of that parameter in the function parameter list.

**Box 4**
Erwin mentioned the need to describe the transformation above for template template parameters.

**Box 5**
The rules above need to be augmented to provide partial ordering rules for conversion function templates. 14.5.2 refers to the rules in this paragraph to select the most specialized conversion function template.

4 Using the transformed function parameter list, perform argument deduction against the other function template. The transformed template is at least as specialized as the other if, and only if, the deduction succeeds and the deduced parameter types are an exact match (so the deduction does not rely on implicit conversions).

5 A template is more specialized than another if, and only if, it is at least as specialized as the other template and that template is not at least as specialized as the first. [*Example:*

```
template<class T> struct A { A(); };

template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);

template<class T> void g(T);
template<class T> void g(T&);

template<class T> void h(const T&);
template<class T> void h(A<T>&);

void m() {
        const int *p;
        f(p);  // f(const T*) is more specialized than f(T) or f(T*)
        float x;
        g(x);  // Ambiguous: g(T) or g(T&)
        A<int> z;
        h(z);  // Ambiguous: h(A<T>&) and h(const T&) are not comparable
        const A<int> z2;
        h(z2); // h(const T&) is called because h(A<T>&) is not callable
}
```

*—end example*]

6 The presence of unused ellipsis and default arguments has no effect on the partial ordering of function templates. [*Example:*

```
template<class T> void f(T);          // #1
template<class T> void f(T*, int=1); // #2
template<class T> void g(T);          // #3
template<class T> void g(T*, ...);    // #4

int main() {
        int* ip;
        f(ip); // calls #2
        g(ip); // calls #4
}
```

*—end example*]

## 14.6 Name resolution [temp.res]

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

— The name of the template itself, and names declared within the template itself.

— Names dependent on a *template-argument* (14.6.2).

— Names from scopes which are visible within the template definition.

2 A name used in a template declaration or definition and that is dependent on a *template-argument* is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword `typename`. [*Example:*

```
// no B declared here

class X;

template<class T> class Y {
        class Z; // forward declaration of member class

        void f() {
                X* a1;    // declare pointer to X
                T* a2;    // declare pointer to T
                Y* a3;    // declare pointer to Y<T>
                Z* a4;    // declare pointer to Z
                typedef typename T::A TA;
                TA* a5;   // declare pointer to T's A
                typename T::A* a6;   // declare pointer to T's A
                T::A* a7; // T::A is not a type name:
                // multiply T::A by a7; ill-formed,
                // no visible declaration of a7
                B* a8;    // B is not a type name:
                // multiply B by a8; ill-formed,
                // no visible declarations of B and a8
        }
};
```

—*end example*]

3 A *qualified-name* that refers to a type and that depends on a *template-argument* (14.6.2) shall be prefixed by the keyword `typename` to indicate that the *qualified-name* denotes a type, forming an elaborated-type-specifier (_dcl.type.elab_).

*elaborated-type-specifier:*
  . . .
  `typename` `::`$_{opt}$ *nested-name-specifier identifier*
  `typename` `::`$_{opt}$ *nested-name-specifier identifier* < *template-argument-list* >
  . . .

4 If a specialization of a template is instantiated for a set of *template-argument*s such that the *qualified-name* prefixed by `typename` does not denote a type, the specialization is ill-formed. The usual qualified name lookup (_basic.lookup.qual_) is used to find the *qualified-name* even in the presence of `typename`. [*Example:*

```
struct A {
        struct X { };
        int X;
};
template<class T> void f(T t) {
        typename T::X x; // ill-formed: finds the data member X
                         // not the member type X
}
```

—*end example*]

5 The keyword `typename` shall only be used in template declarations and definitions, including in the return type of a function template or member function template, in the return type for the definition of a member function of a class template or of a class nested within a class template, and in the *type-specifier* for the definition of a static member of a class template or of a class nested within a class template. The keyword `typename` shall only be applied to qualified names, but those names need not be dependent. The keyword `typename` is not permitted in a *base-specifier* or in a *mem-initializer*; in these contexts a *qualified-name* that depends on a *template-argument* (14.6.2) is implicitly assumed to be a type name.

6 Within the definition of a class template or within the definition of a member of a class template, the keyword `typename` is not required when referring to the unqualified name of a previously declared member of the class template that declares a type. The keyword `typename` shall always be specified when the member is referred to using a qualified name, even if the qualifier is simply the class template name. [*Example:*

```
template<class T> struct A {
    typedef int B;
    A::B b;           // ill-formed: typename required before A::B
    void f(A<T>::B); // ill-formed: typename required before A<T>::B
    typename A::B g(); // ok
};
```

The keyword `typename` is required whether the qualified name is `A` or `A<T>` because `A` or `A<T>` are synonyms within a class template with the parameter list `<T>`. ]

7 Knowing which names are type names allows the syntax of every template definition to be checked. No diagnostic shall be issued for a template definition for which a valid specialization can be generated. If no valid specialization can be generated for a template definition, and that template is not instantiated, the template definition is ill-formed, no diagnostic required. [*Note:* if a template is instantiated, errors will be diagnosed according to the other rules in this Standard. Exactly when these errors are diagnosed is a quality of implementation issue. ] [*Example:*

```
int j;
template<class T> class X {
        // ...
        void f(T t, int i, char* p)
        {
                t = i;  // diagnosed if X::f is instantiated
                        // and the assignment to t is an error
                p = i;  // may be diagnosed even if X::f is
                        // not instantiated
                p = j;  // may be diagnosed even if X::f is
                        // not instantiated
        }
        void g(T t) {
                +;      // may be diagnosed even if X::g is
                        // not instantiated
        }
};
```

—*end example*]

8 When looking for the declaration of a name used in a template definition, the usual lookup rules (_basic.lookup.unqual_, _basic.lookup.koenig_) are used for nondependent names. The lookup of names dependent on the template arguments is postponed until the actual template argument is known (14.6.2). [*Example:*

```
#include <iostream>
using namespace std;

template<class T> class Set {
        T* p;
        int cnt;
public:
        Set();
        Set<T>(const Set<T>&);
        void printall()
        {
                for (int i = 0; i<cnt; i++)
                        cout << p[i] << '\n';
        }
        // ...
};
```

in the example, `i` is the local variable `i` declared in `printall`, `cnt` is the member `cnt` declared in `Set`, and `cout` is the standard output stream declared in `iostream`. However, not every declaration can be found this way; the resolution of some names must be postponed until the actual *template-argument*s are known. For example, even though the name `operator<<` is known within the definition of `printall()` and a declaration of it can be found in `<iostream>`, the actual declaration of `operator<<` needed to print `p[i]` cannot be known until it is known what type `T` is (14.6.2). ]

9 If a name does not depend on a *template-argument* (as defined in 14.6.2), a declaration (or set of declarations) for that name shall be in scope at the point where the name appears in the template definition; the name is bound to the declaration (or declarations) found at that point and this binding is not affected by declarations that are visible at the point of instantiation. [*Example:* *

```
void f(char);

template<class T> void g(T t)
{
        f(1);      // f(char)
        f(T(1));   // dependent
        f(t);      // dependent
        dd++;       // not dependent
                    // error: declaration for dd not found
}

void f(int);

double dd;
void h()
{
        g(2);   // will cause one call of f(char) followed
                //  by two calls of f(int)
        g('a'); // will cause three calls of f(char)
}
```

—*end example*]

### 14.6.1 Locally declared names [temp.local]

1 Within the scope of a class template, the unqualified name of the template, when not followed by <, is equivalent to the name of the template followed by the *template-parameter*s enclosed in <>. [*Note:* the equivalence within the scope of a class template between the name of the template and the corresponding *template-id* does not apply when the name of the template is qualified. ] [*Example:* the constructor for `Set` can be referred to as `Set()` or `Set<T>()`. ] Other specializations (14.7.3) of the class can be referred to by explicitly qualifying the template name with the appropriate *template-argument*s. [*Example:*

```
template<class T> class X {
        X* p;           // meaning X<T>
        X<T>* p2;
        X<int>* p3;
};
```

—*end example*]

2 Within the scope of a class template specialization, the name of the specialization, when not followed by <, is equivalent to the name of the specialization followed by the *template-argument*s enclosed in <>. [*Example:*

```
template<class T> class Y;

template<> class Y<int> {
        Y* p;           // meaning Y<int>
        Y<char>* q;     // meaning Y<char>
};
```

—*end example*]

3 The scope of a *template-parameter* extends from its point of declaration until the end of its template. A *template-parameter* hides any entity with the same name in the enclosing scope. [*Note:* this implies that a *template-parameter* can be used in the declaration of subsequent *template-parameter*s and their default arguments but cannot be used in preceding *template-parameter*s or their default arguments. For example,

```
template<class T, T* p, class U = T> class X { /* ... */ };
template<class T> void f(T* p = new T);
```

This also implies that a *template-parameter* can be used in the specification of base classes. For example,

```
template<class T> class X : public Array<T> { /* ... */ };
template<class T> class Y : public T { /* ... */ };
```

The use of a *template-parameter* as a base class implies that a class used as a *template-argument* must be defined and not just declared when the class template is instantiated. ]

4 A *template-parameter* shall not be redeclared within its scope (including nested scopes). A *template-parameter* shall not have the same name as the template name. [*Example:*

```
template<class T, int i> class Y {
        int T;  // error: template-parameter redeclared
        void f() {
                char T; // error: template-parameter redeclared
        }
};

template<class X> class X; // error: template-parameter redeclared
```

—*end example*]

5 In the definition of a member of a class template that appears outside of the class template definition, the name of a member of this template hides the name of a *template-parameter*. [*Example:*

```
template<class T> struct A {
        struct B { /* ... */ };
        void f();
};

template<class B> void A<B>::f() {
        B b;  // A's B, not the template parameter
}
```

—*end example*]

6 In the definition of a member of a class template that appears outside of the namespace containing the class template definition, the name of a *template-parameter* hides the name of a member of this namespace. [*Example:*

```
namespace N {
        class C { };
        template<class T> class B {
                void f(T);
        };
}
template<class C> void N::B<C>::f(C) {
        C b;  // C is the template parameter, not N::C
}
```

—*end example*]

7 In the definition of a class template or in the definition of a member of such a template that appears outside of the template definition, the name of a base class and, if the base class does not depend on a *template-argument*, the name of a base class member hides the name of a *template-parameter* with the same name. [*Example:*

```
struct A {
        struct B { /* ... */ };
        int a;
        int Y;
};

template<class B, class a> struct X : A {
        B b;  // A's B
        a b;  // error: A's a isn't a type name
};
```

—*end example*]

### 14.6.2 Dependent names [temp.dep]

1 Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct *depends* on the template arguments. In particular, types and expressions may depend on the type and or value of template arguments and this determines the context for name lookup for certain names. Expressions may be *type-dependent* (on the type of a template argument) or *value-dependent* (on the value of a non-type template argument). In an expression of the form:

*postfix-expression* `(` *expression-list*$_{\mathrm{opt}}$ `)`

where the *postfix-expression* is an *identifier*, the *identifier* denotes a *dependent name* if and only if any of the expressions in the *expression-list* is a type-dependent expression (14.6.2.2). If an operand of an operator is a type-dependent expression, the operator also denotes a dependent name. Such names are unbound and are looked up at the point of the template instantiation (14.6.4.1) in both the context of the template definition and the context of the point of instantiation.

2 [*Example:*

```
template<class T> struct X : B<T> {
        typename T::A* pa;
        void f(B<T>* pb) {
                static int i = B<T>::i;
                pb->j++;
        }
};
```

the base class name `B<T>`, the type name `T::A`, the names `B<T>::i`, `pa`, `pb` and `pb->j` explicitly depend on the *template-argument*. This shows a typical dependent operator call:

```
class Horse { /* ... */ };

ostream& operator<<(ostream&,const Horse&);

void hh(Set<Horse>& h)
{
        h.printall();
}
```

In the call of `Set<Horse>::printall()`, the meaning of the `<<` operator used to print `p[i]` in the definition of `Set<T>::printall()` (14.6), is

```
operator<<(ostream&,const Horse&);
```

This function takes an argument of type `Horse` and is called from a template with a *template-parameter* `T` for which the *template-argument* is `Horse`. Because this function depends on a *template-argument*, the call is well-formed. Some calls that depend on a *template-argument* type `T` are:

1) The function called has a parameter that depends on `T` according to the type deduction rules (14.8.2). For example, `f(T)`, `f(Array<T>)`, and `f(const T*)`.

2) The type of the actual argument depends on `T`. For example, `f(T(1))`, `f(t)`, `f(g(t))`, and `f(&t)` assuming that `t` has the type `T`.

**Box 6**

[Josee: I would like to remove this bullet. Erwin indicated that this bullet is not needed anymore. The example below is also wrong: it does not show an example of bullet 3). Any objections to removing this bullet? ]

3) A call is resolved by the use of a conversion to `T` without either an argument or a parameter of the called function being of a type that depends on `T` as specified in (1) and (2). For example,

```
struct B { };
struct T : B { };
struct X { operator T(); };

void f(B);

void g(X x)
{
        f(x);  // meaning f( B( x.operator T() ) )
               // so the call f(x) depends on T
}
```

This ill-formed template instantiation uses a function that does not depend on a template-argument:

```
template<class T> class Z {
public:
        void f() const
        {
                g(1); // g() not found in Z's context.
                      // ill-formed, even if g is declared at
                      // the point of instantiation.  This
                      // could be diagnosed either here or
                      // at the point of instantiation.
        }
};
```

```
void g(int);
void h(const Z<Horse>& x)
{
        x.f(); // error: g(int) called by g(1) does not depend
               // on template-argument ``Horse''
}
```

The call `x.f()` gives rise to the specialization:

```
void Z<Horse>::f() { g(1); }
```

The call `g(1)` would call `g(int)`, but since that call does not depend on the *template-argument* `Horse` and because `g(int)` wasn't in scope at the point of the definition of the template, the call `x.f()` is ill-formed. ]

3 In the definition of a class template or in the definition of a member of such template that appears outside of the template definition, if a base class of this template depends on a *template-argument*, the base class scope is not examined during name look up until the class template is instantiated. [*Example:*

```
typedef double A;
template<class T> B {
        typedef int A;
};
template<class T> struct X : B<T> {
        A a; // a has type double
};
```

The type name `A` in the definition of `X<T>` binds to the typedef name defined in the global namespace scope, not to the typedef name defined in the base class `B<T>`. ]

4 If a base class is a dependent type, a member of that class cannot hide a name declared within a template, or a name from the template's enclosing scopes. [*Example:*

```
struct A {
        struct B { /* ... */ };
        int a;
        int Y;
};

int a;

template<class T> struct Y : T {
        struct B { /* ... */ };
        B b;                      // The B defined in Y
        void f(int i) { a = i; }  // ::a
        Y* p;                     // Y<T>
};

Y<A> ya;
```

The members `A::B`, `A::a`, and `A::Y` of the template argument `A` do not affect the binding of names in `Y<A>`. ]

#### 14.6.2.1 Dependent types [temp.dep.type]

1 A type is dependent if it is

— a template parameter,

— a *qualified-id* whose *nested-name-specifier* contains a *class-name* that names a dependent type or whose *unqualified-id* names a dependent type,

— a cv-qualified type where the cv-unqualified type is dependent,

— a compound type constructed from any dependent type,

— an array type constructed from any dependent type or whose size is specified by a constant expression that is value-dependent,

— a *template-id* in which either the template name is a template parameter or any of the template arguments is a dependent type or an expression that is type-dependent or value-dependent.

### 14.6.2.2 Type-dependent expressions [temp.dep.expr]

1 Except as described below, an expression is type-dependent if any subexpression is type-dependent.

2 `this` is type-dependent if the class type of the enclosing member function is dependent (14.6.2.1).

3 An *id-expression* is type-dependent if it contains:

— an *identifier* that was declared with a dependent type,

— a *template-id* that is dependent,

— a *conversion-function-id* that specifies a dependent type,

— a *nested-name-specifier* that contains a *class-name* that names a dependent type.

4 Expressions of the following forms are type-dependent only if the type specified by the *type-id*, *simple-type-specifier* or *new-type-id* is dependent, even if any subexpression is type-dependent:

*simple-type-specifier* `(` *expression-list*$_{opt}$ `)`
`::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id* *new-initializer*$_{opt}$
`::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$
`dynamic_cast <` *type-id* `> (` *expression* `)`
`static_cast <` *type-id* `> (` *expression* `)`
`const_cast <` *type-id* `> (` *expression* `)`
`reinterpret_cast <` *type-id* `> (` *expression* `)`
`(` *type-id* `)` *cast-expression*

5 Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

*literal*
*postfix-expression* `.` *pseudo-destructor-name*
*postfix-expression* `->` *pseudo-destructor-name*
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`typeid (` *expression* `)`
`typeid (` *type-id* `)`
`::`$_{opt}$ `delete` *cast-expression*
`::`$_{opt}$ `delete [ ]` *cast-expression*
`throw` *assignment-expression*$_{opt}$

### 14.6.2.3 Value-dependent expressions [temp.dep.constexpr]

1 Except as described below, a constant expression is value-dependent if any subexpression is value-dependent.

2 An *identifier* is value-dependent if it is:

— a name declared with a dependent type,

— the name of a non-type template parameter,

— a constant with integral or enumeration type and is initialized with an expression that is value-dependent.

3 Expressions of the following form are value-dependent if the *unary-expression* is type-dependent or the *type-id* is dependent (even if `sizeof` *unary-expression* and `sizeof (` *type-id* `)` are not type-dependent):

`sizeof` *unary-expression*
`sizeof (` *type-id* `)`

4 Expressions of the following form are value-dependent if either the *type-id* or *simple-type-specifier* is dependent or the *expression* or *cast-expression* is value-dependent:

*simple-type-specifier* `(` *expression-list*$_{opt}$ `)`
`static_cast <` *type-id* `> (` *expression* `)`
`const_cast <` *type-id* `> (` *expression* `)`
`reinterpret_cast <` *type-id* `> (` *expression* `)`
`(` *type-id* `)` *cast-expression*

### 14.6.2.4 Dependent template arguments [temp.dep.temp]

1 A template template argument is dependent if it names a template argument or is a *qualified-id* where the *nested-name-specifier* contains a *class-name* that names a dependent type.

2 A non-integral non-type template argument is dependent if it has either of the following forms

*qualified-id*
`&` *qualified-id*

and the *nested-name-specifier* specifies a *class-name* that names a dependent type.

3 A type template argument is dependent if the type it specifies is dependent.

4 An integral non-type template argument is dependent if the constant expression it specifies is value-dependent.

### 14.6.3 Non-dependent names [temp.nondep]

1 Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used. [*Example:*

```
void g(double);
void h();

template<class T> class Z {
public:
        void f() {
                g(1); // calls g(double)
                h++;  // ill-formed: cannot increment function;
                      // this could be diagnosed either here or
                      // at the point of instantiation
        }
};

void g(int); // not in scope at the point of the template
             // definition, not considered for the call g(1)
```

—*end example*]

### 14.6.4 Dependent name resolution [temp.dep.res]

1 In resolving dependent names, names from the following sources are considered:

— 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 [temp.point]

1 For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template argument, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

2 For a class template specialization, a class member template specialization, or a specialization for a class member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization, if the context from which the specialization is referenced depends on a template argument, and if the specialization is not instantiated previous to the instantiation of the enclosing template, the point of instantiation is immediately before the point of instantiation of the enclosing template. Otherwise, the point of instantiation for such a specialization immediately precedes the namespace scope declaration or definition that refers to the specialization.

3 If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.

4 The instantiation context of an expression that depends on the template arguments is the set of declarations with external linkage visible at the point of instantiation of the template specialization.

5 A specialization for a function template, a member function template, or of a member function or static data member of a class template may have multiple points of instantiations within a translation unit. A specialization for a class template has at most one point of instantiation within a translation unit. A specialization for any template may have points of instantiation in multiple translation units. If two different points of instantiation give a template specialization different meanings according to the one definition rule (_basic.def.odr_), the program is ill-formed, no diagnostic required.

### 14.6.4.2 Candidate Functions [temp.dep.candidate]

1 For a function call that depends on a template argument, if the function name is an *unqualified-id*, the candidate functions are found using the usual lookup rules (_basic.lookup.unqual_, _basic.lookup.koenig_) except that:

— For the part of the lookup using unqualified name lookup (_basic.lookup.unqual_), only function declarations with external linkage from the template definition context are found.

— For the part of the lookup using associated namespaces (_basic.lookup.koenig_), only function declarations with external linkage found in either the template definition context or the template instantiation context are found.

If the call would be ill-formed or would find a better match had the lookup within the associated namespaces considered all the function declarations with external linkage introduced in those namespaces in all translation units, not just considered those declarations found in the template definition and template instantiation contexts, then the program has undefined behavior. *

### 14.6.4.3 Conversions [temp.dep.conv]

1 Any standard conversion sequence (_over.ics.scs_) may be applied to an argument in a function call that depends on a template argument. A user-defined conversion sequence (_over.ics.user_) may be applied to an argument in a function call that depends on a template argument, but the user-defined conversion in this sequence shall either be a conversion function that is a member function of the class type of the argument, or shall be a constructor of the class type that is the target type of the user-defined conversion sequence. The user-defined conversion function thus selected shall be found either in the template definition context or in the template instantiation context. [*Note:* The set of candidate functions is formed first, before conversions are considered, so the possible conversions do not affect the set of candidate functions. ]

### 14.6.5 Friend names declared within a class template [temp.inject]

1 Friend classes or functions can be declared within a class template. When a template is instantiated, the names of its friends are treated as if the specialization had been explicitly declared at its point of instantiation.

2 The names of friend functions of a class template specialization are found by the usual lookup rules, including the rules for associated namespaces (_basic.lookup.koenig_).[4] [*Example:*

```
template<typename T> class number {
        //...
        friend number<T> gcd(const number<T>& x,
                             const number<T>& y) { ... }
        //...
};

void g()
{
        number<double> a, b;
        //...
        a = gcd(a,b);   // looks inside number<double> for gcd
}
```

—*end example*]

### 14.7 Template specialization [temp.spec]

1 A function instantiated from a function template is called an instantiated function. A class instantiated from a class template is called an instantiated class. A member function, a member class, or a static data member of a class template instantiated from the member definition of the class template is called, respectively, an instantiated member function, member class or static data member. A member function instantiated from a member function template is called an instantiated member function. A member class instantiated from a member class template is called an instantiated member class. The act of instantiating a function, a class, a member of a class template or a member template is referred to as template instantiation. An explicit specialization may be declared for a function template, a class template, a member of a class template or a member template. An explicit specialization declaration is introduced by `template<>`. In an explicit specialization declaration for a class template, a member of a class template or a class member template, the name of the class that is explicitly specialized shall be a *template-id*. In the explicit specialization declaration for a function template or a member function template, the name of the function or member function explicitly specialized may be a *template-id*. [*Example:*

```
template<class T = int> struct A
{
        static int x;
};
template<class U> void g(U) { }

template<> struct A<double> { };  // specialize for T == double
template<> struct A<> { };        // specialize for T == int
template<> void g(char) { }       // specialize for U == char
                                  // U is deduced from the parameter type
template<> void g<int>(int) { }   // specialize for U == int
template<> int A<char>::x = 0;    // specialize for T == char
template<> int A<>::x = 1;        // specialize for T == int
```

—*end example*]

[4] Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.

2 An instantiated template specialization can be either implicitly instantiated (14.7.1) for a given argument list or be explicitly instantiated (14.7.2). A specialization is a class, function, or class member that is either instantiated or explicitly specialized (14.7.3). A template that has been used in a way that requires a specialization of its definition causes the specialization to be implicitly instantiated unless a declaration for the explicit specialization appears before the specialization is used.

3 No program shall explicitly instantiate any template more than once, both explicitly instantiate and explicitly specialize a template, or specialize a template more than once for a given set of *template-arguments*. An implementation is not required to diagnose a violation of this rule.

4 Each class template specialization instantiated from a template has its own copy of any static members. [*Example:*

```
template<class T> class X {
        static T s;
        // ...
};
template<class T> T X<T>::s = 0;
X<int> aa;
X<char*> bb;
```

`X<int>` has a static member `s` of type `int` and `X<char*>` has a static member `s` of type `char*`. ]

**Box 7**

core issue 881: What class-key can be used in the declarations of explicit instantatiations, explicit specializations and partial specializations? Especially wrt the `union` class-key.

### 14.7.1 Implicit instantiation [temp.inst]

1 Unless a class template specialization has been explicitly instantiated (14.7.2) or explicitly specialized (14.7.3), the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program. The implicit instantiation of a class template specialization does not cause the implicit instantiation of the definitions of the class member functions, member classes, static data members or member templates. Unless a member of a class template or a member template has been explicitly instantiated or explicitly specialized, the specialization of the member is implicitly instantiated when the specialization is referenced in a context that requires the member definition to exist; in particular, the initialization (and any associated side-effects) of a static data member does not occur until the static data member is itself used in a way that requires the definition of the static data member to exist. Unless a function template specialization has been explicitly instantiated or explicitly specialized, the function template specialization is implicitly instantiated when the specialization is referenced in a context that requires a function definition to exist.

2 [*Example:*

```
template<class T> class Z {
public:
        void f();
        void g();
};
```

```
void h()
{
        Z<int> a;      // instantiation of class Z<int> required
        Z<char>* p;    // instantiation of class Z<char> not
                       // required
        Z<double>* q; // instantiation of class Z<double>
                       // not required

        a.f();  // instantiation of Z<int>::f() required
        p->g(); // instantiation of class Z<char> required, and
                // instantiation of Z<char>::g() required
}
```

Nothing in this example requires `class Z<double>`, `Z<int>::g()`, or `Z<char>::f()` to be implicitly instantiated. ]

3 A class template specialization is implicitly instantiated if the class type is used in a context that requires a completely-defined object type or if the completeness of the class type affects the semantics of the program; in particular, if an expression whose type is a class template specialization is involved in overload resolution, pointer conversion, pointer to member conversion or is used as the operand of a delete expression, the class template specialization is implicitly instantiated. [*Example:*

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

void f(void*);
void f(B<int>*);

void g(D<int>* p, D<char>* pp, D<double> ppp)
{
        f(p); // instantiation of D<int> required: call f(B<int>*)

        B<char>* q = pp; // instantiation of D<char> required:
                         // convert D<char>* to B<char>*

        delete ppp; // instantiation of D<double> required
}
```

—*end example*]

4 If the overload resolution process can determine the correct function to call without instantiating a class template definition, it is unspecified whether that instantiation actually takes place. [*Example:*

```
template <class T> struct S {
        operator int();
};

void f(int);
void f(S<int>&);
void f(S<float>);

void g(S<int>& sr) {
        f(sr);  // instantiation of S<int> allowed but not required
                // instantiation of S<float> allowed but not required
};
```

—*end example*]

5 If an implicit instantiation of a class template specialization is required and the template is declared but not defined, the program is ill-formed. [*Example:*

```
template<class T> class X;

X<char> ch; // error: definition of X required
```

—*end example*]

6 If a function template or a member function template specialization is used in a way that involves overload resolution, a declaration of the specialization is implicitly instantiated (14.8.3).

7 An implementation shall not implicitly instantiate a function template, a member template, a non-virtual member function, a member class or a static data member of a class template that does not require instantiation. It is unspecified whether or not an implementation implicitly instantiates a virtual member function of a class template that does not require specialization.

8 Implicitly instantiated class and function template specializations are placed in the namespace where the template is defined. Implicitly instantiated specializations for members of a class template are placed in the namespace where the enclosing class template is defined. Implicitly instantiated member templates are placed in the namespace where the enclosing class or class template is defined. [*Example:*

```
namespace N {
        template<class T> class List {
        public:
                T* get();
                // ...
        };
}

template<class K, class V> class Map {
        N::List<V> lt;
        V get(K);
        //  ...
};

void g(Map<char*,int>& m)
{
        int i = m.get("Nicholas");
        // ...
}
```

a call of `lt.get()` from `Map<char*,int>::get()` would place `List<int>::get()` in the namespace `N` rather than in the global namespace. ]

9 [*Note:* 14.6.4.1 defines the point of instantiation of a template specialization. ]

10 There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations, which could involve more than one template. The result of an infinite recursion in instantiation is undefined. [*Example:*

```
template<class T> class X {
        X<T>* p; // ok
        X<T*> a; // implicit generation of X<T> requires
                 // the implicit instantiation of X<T*> which requires
                 // the implicit instantiation of X<T**> which ...
};
```

—*end example*]

### 14.7.2 Explicit instantiation [temp.explicit]

1 A class, a function or member template specialization can be explicitly instantiated from its template. The member function, member class or static data member of a class template can be explicitly instantiated from the member definition associated with its class template.

2 The syntax for explicit instantiation is:

> *explicit-instantiation*:
> `template` *declaration*

If the explicit instantiation is for a class, a function or a member template specialization, the *unqualified-id* in the *declaration* shall be either a *template-id* or, where all template arguments can be deduced, a *template-name*. [*Note:* the declaration may declare a *qualified-id*, in which case the *unqualified-id* of the *qualified-id* must be a *template-id*. ] If the explicit instantiation is for a member function, a member class or a static data member of a class template specialization, the name of the class template specialization in the *qualified-id* for the member *declarator* shall be a *template-id*. [*Example:*

```
template<class T> class Array { void mf(); };
template class Array<char>;
template void Array<int>::mf();

template<class T> void sort(Array<T>& v) { /* ... */ }
template void sort(Array<char>&); // argument is deduced here

namespace N {
        template<class T> void f(T&) { }
}
template void N::f<int>(int&);
```

—*end example*]

3 A declaration of a function template shall be in scope at the point of the explicit instantiation of the function template. A definition of the class or class template containing a member function template shall be in scope at the point of the explicit instantiation of the member function template. A definition of a class template or class member template shall be in scope at the point of the explicit instantiation of the class template or class member template. A definition of a class template shall be in scope at the point of an explicit instantiation of a member function or a static data member of the class template. A definition of a member class of a class template shall be in scope at the point of an explicit instantiation of the member class. If the *declaration* of the explicit instantiation names an implicitly-declared special member function (_special_), the program is ill-formed.

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

5 An explicit instantiation of a class or function template specialization is placed in the namespace in which the template is defined. An explicit instantiation for a member of a class template is placed in the namespace where the enclosing class template is defined. An explicit instantiation for a member template is placed in the namespace where the enclosing class or class template is defined. [*Example:*

```
namespace N {
        template<class T> class Y { void mf() { } };
}

template class Y<int>; // error: class template Y not visible
                       // in the global namespace

using N::Y;
template class Y<int>; // ok: explicit instantiation in namespace N

template class N::Y<char*>; // ok: explicit instantiation in namespace N
template void N::Y<double>::mf(); // ok: explicit instantiation
                                  //     in namespace N
```

—*end example*]

6 A trailing *template-argument* can be left unspecified in an explicit instantiation of a function template specialization or of a member function template specialization provided it can be deduced from the type of a function argument (14.8.2). [*Example:*

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v);

// instantiate sort(Array<int>&) - template-argument deduced
template void sort<>(Array<int>&);
```

—*end example*]

7 The explicit instantiation of a class template specialization implies the instantiation of all of its members not previously explicitly specialized in the translation unit containing the explicit instantiation. *

8 The usual access checking rules do not apply to explicit instantiations. [*Note:* In particular, the template arguments and names used in the function declarator (including parameter types, return types and exception specifications) may be private types or objects which would normally not be accessible and the template may be a member template or member function which would not normally be accessible. ]

### 14.7.3 Explicit specialization [temp.expl.spec]

1 An explicit specialization of any of the following:

— function template

— class template

— member function of a class template

— static data member of a class template

— member class of a class template

— member class template of a class template

— member function template of a class template

can be declared by a declaration introduced by `template<>`; that is:

*explicit-specialization*:
    `template < >` *declaration*

[*Example:*

```
template<class T> class stream;

template<> class stream<char> { /* ... */ };
```

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

template<> void sort<char*>(Array<char*>&) ;
```

Given these declarations, `stream<char>` will be used as the definition of streams of `char`s; other streams will be handled by class template specializations instantiated from the class template. Similarly, `sort<char*>` will be used as the sort function for arguments of type `Array<char*>`; other `Array` types will be sorted by functions generated from the template. ]

2 An explicit specialization shall be declared in the namespace of which the template is a member, or, for member templates, in the namespace of which the enclosing class or enclosing class template is a member. An explicit specialization of a member function, member class or static data member of a class template shall be declared in the namespace of which the class template is a member. Such a declaration may also be a definition. If the declaration is not a definition, the specialization may be defined later in the namespace in which the explicit specialization was declared, or in a namespace that encloses the one in which the explicit specialization was declared.

3 A declaration of a function template or class template being explicitly specialized shall be in scope at the point of declaration of an explicit specialization. [*Note:* a declaration, but not a definition of the template is required. ] The definition of a class or class template shall be in scope at the point of declaration of an explicit specialization for a member template of the class or class template. [*Example:*

```
template<> class X<int> { /* ... */ }; // error: X not a template

template<class T> class X;

template<> class X<char*> { /* ... */ }; // fine: X is a template
```

—*end example*]

4 A member function, a member class or a static data member of a class template may be explicitly specialized for a class specialization that is implicitly instantiated; in this case, the definition of the class template shall be in scope at the point of declaration of the explicit specialization for the member of the class template. If such an explicit specialization for the member of a class template names an implicitly-declared special member function (_special_), the program is ill-formed.

5 A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the class template; instead, the member of the class template specialization shall itself be explicitly defined. In this case, the definition of the class template explicit specialization shall be in scope at the point of declaration of the explicit specialization of the member. The definition of an explicitly specialized class is unrelated to the definition of a generated specialization. That is, its members need not have the same names, types, etc. as the members of the a generated specialization. Definitions of members of an explicitly specialized class are defined in the same manner as members of normal classes, and not using the explicit specialization syntax. [*Example:*

```
template<class T> struct A {
        void f(T) { /* ... */ }
};

template<> struct A<int> {
        void f(int);
};
```

```
void h()
{
        A<int> a;
        a.f(16);  // A<int>::f must be defined somewhere
}

// explicit specialization syntax not used for a member of
// explicitly specialized class template specialization
void A<int>::f() { /* ... */ }
```

—*end example*]

6 If a template, a member template or the member of a class template is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs. If the program does not provide a definition for an explicit specialization, the program is ill-formed, no diagnostic required; an implicit instantiation is never generated for an explicit specialization that is declared but not defined. [*Example:*

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

void f(Array<String>& v)
{
        sort(v); // use primary template
                 // sort(Array<T>&), T is String
}

template<> void sort<String>(Array<String>& v); // error: specialization
                                                // after use of primary template
template<> void sort<>(Array<char*>& v); // fine sort<char*> not yet used
```

—*end example*] [*Note:* if the implicit instantiation of an exported template refers to an explicit specialization, a declaration for this explicit specialization shall appear in the translation unit where the exported template is defined, before the definition of the exported template.

```
// translation unit #1
template<class T> struct A { };
export template<class T> void f(T) { A<T> a; }

// translation unit #2
template<class T> struct A { };

// not visible during instantiation of f(int)
// ill-formed program: f<int>(int) used the implicit instantiation of
// A<int> and an explicit specialization exists
template<> struct A<int> { };

template<class T> void f(T);
void g() { f(1); }
```

]

7 A template explicit specialization is in the scope of the namespace in which the template was defined. [*Example:*

```
namespace N {
        template<class T> class X { /* ... */ };
        template<class T> class Y { /* ... */ };

        template<> class X<int> { /* ... */ }; // ok: specialization
                                               //     in same namespace
        template<> class Y<double>;            // forward declare intent to
                                               // specialize for double
}

template<> class N::Y<double> { /* ... */ }; // ok: specialization
                                             //     in same namespace
```

—*end example*]

8 A *template-id* that names a class template explicit specialization that has been declared but not defined can be used exactly like the names of other incompletely-defined classes (_basic.types_). [*Example:*

```
template<class T> class X; // X is a class template
template<> class X<int>;

X<int>* p; // ok: pointer to declared class X<int>
X<int> x; // error: object of incomplete class X<int>
```

—*end example*]

9 A trailing *template-argument* can be left unspecified in the *template-id* naming an explicit function template specialization provided it can be deduced from the function argument type. [*Example:*

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v);

// explicit specialization for sort(Array<int>&)
// with deduces template-argument of type int
template<> void sort(Array<int>&);
```

—*end example*]

10 It is possible for a specialization with a given function signature to be instantiated from more than one function template. In such cases, explicit specification of the template arguments must be used to uniquely identify the function template specialization being specialized. [*Example:*

```
template <class T> void f(T);
template <class T> void f(T*);
template <>        void f(int*);        // Ambiguous
template <>        void f<int>(int*);   // OK
template <>        void f(int);         // OK
```

—*end example*]

11 A function with the same name as a template and a type that exactly matches that of a template specialization is not an explicit specialization (14.5.5).

12 An explicit specialization of a function template is inline only if it is explicitly declared to be, and independently of whether its function template is. [*Example:*

```
template<class T> void f(T) { /* ... */ }
template<class T> inline T g(T) { /* ... */ }

template<> inline void f<>(int) { /* ... */ } // ok: inline
template<> int g<>(int) { /* ... */ } // ok: not inline
```

—*end example*]

13 An explicit specialization of a static data member of a template is a definition if the declaration includes an initializer; otherwise, it is a declaration. [*Note:* there is no syntax for the definition of a static data member of a template that requires default initialization. *

```
template<> X Q<int>::x;
```

This is a declaration regardless of whether X can be default initialized (_dcl.init_). ]

14 A member or a member template of a class template may be explicitly specialized for a given implicit instantiation of the class template, even if the member or member template is defined in the class template definition. An explicit specialization of a member or member template is specified using the template specialization syntax. [*Example:* *

```
template<class T> struct A {
        void f(T);
        template<class X> void g(T,X);
        void h(T) { }
};

// specialization
template<> void A<int>::f(int);

// out of class member template definition
template<class T> template<class X> void A<T>::g(T,X) { }

// member template partial specialization
template<> template<class X> void A<int>::g(int,X);

// member template specialization
template<> template<>
        void A<int>::g(int,char);        // X deduced as char
template<> template<>
        void A<int>::g<char>(int,char);  // X specified as char

// member specialization even if defined in class definition
template<> void A<int>::h(int) { }
```

—*end example*]

15 A member or a member template may be nested within many enclosing class templates. If the declaration of an explicit specialization for such a member appears in namespace scope, the member declaration shall be preceded by a `template<>` for each enclosing class template that is explicitly specialized. [*Example:*

```
template<class T1> class A {
        template<class T2> class B {
                void mf();
        };
};
template<> template<> A<int>::B<double> { };
template<> template<> void A<char>::B<char>::mf() { };
```

—*end example*]

16 In an explicit specialization declaration for a member of a class template or a member template that appears in namespace scope, the member template or some of the enclosing class templates may remain unspecialized; in such cases, the keyword `template` followed by a *template-parameter-list* shall be provided instead of the `template<>` preceding the explicit specialization declaration of the member. The types of the *template-parameters* in the *template-parameter-list* shall be the same as those specified in the primary template definition. [*Example:*

```
template<class T1> class A {
        template<class T2> class B {
                void mf();
        };
};
template<> template<class X> A<int>::B<X> { };
template<class Y> template<> void A<Y>::B<double>::mf() { };
```

—*end example*]

17 A specialization of a member function template or member class template of a non-specialized class template is itself a template.

18 An explicit specialization declaration shall not be a friend declaration.

19 Default function arguments shall not be specified in a declaration or a definition for one of the following explicit specializations:

— the explicit specialization of a function template;

— the explicit specialization of a member function template;

— the explicit specialization of a member function of a class template where the class template specialization to which the member function specialization belongs is implicitly instantiated. [*Note:* default function arguments may be specified in the declaration or definition of a member function of a class template specialization that is explicitly specialized. ]

### 14.8 Function template specializations [temp.fct.spec]

1 A function instantiated from a function template is called a function template specialization; so is an explicit specialization of a function template. Template arguments can either be explicitly specified when naming the function template specialization or be deduced (14.8.2) from the function arguments in a call to the function template specialization.

2 Each function template instantiated from a template has its own copy of any static variable. [*Example:*

```
template<class T> void f(T* p)
{
        static T s;
        // ...
};

void g(int a, char* b)
{
        f(&a);  // call f<int>(int*)
        f(&b);  // call f<char*>(char**)
}
```

Here `f<int>(int*)` has a static variable `s` of type `int` and `f<char*>(char**)` has a static variable `s` of type `char*`. ]

### 14.8.1 Explicit template argument specification [temp.arg.explicit]

1 Template arguments can be specified when referring to a function template specialization by qualifying the function template specialization name by the list of *template-argument*s exactly as *template-argument*s are specified in uses of a class template specialization. [*Example:*

```
template<class T> void sort(Array<T>& v);
void f(Array<dcomplex>& cv, Array<int>& ci)
{
        sort<dcomplex>(cv); // sort(Array<dcomplex>&)
        sort<int>(ci);      // sort(Array<int>&)
}
```

and

```
template<class U, class V> U convert(V v);

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

—*end example*]

2 Trailing arguments that can be deduced (14.8.2) may be omitted from the list of explicit *template-argument*s. [*Example:*

```
template<class X, class Y> X f(Y);
void g()
{
        int i = f<int>(5.6); // Y is deduced to be double
        int j = f(5.6);      // ill-formed: X cannot be deduced
}
```

—*end example*]

3 Implicit conversions (_conv_) will be performed on a function argument to bring it to the type of the corresponding function parameter if the parameter type is fixed by an explicit specification of a *template-argument*. [*Example:*

```
template<class T> void f(T);

class Complex {
        // ...
        Complex(double);
};

void g()
{
        f<Complex>(1); // ok, means f<Complex>(Complex(1))
}
```

—*end example*]

4 [*Note:* because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates. ]

### 14.8.2 Template argument deduction [temp.deduct]

1 Template arguments that can be deduced from the function arguments of a call need not be explicitly specified. [*Example:*

```
void f(Array<dcomplex>& cv, Array<int>& ci)
{
        sort(cv);   // call sort(Array<dcomplex>&)
        sort(ci);   // call sort(Array<int>&)
}
```

and

```
void g(double d)
{
        int i = convert<int>(d);   // call convert<int,double>(double)
        int c = convert<char>(d);  // call convert<char,double>(double)
}
```

—*end example*] [*Note:* if a *template-parameter* is only used to represent a function template return type, its corresponding *template-argument* cannot be deduced and the *template-argument* must be explicitly specified. ]

2 Type deduction is done for each function template argument that is not explicitly specified. The type of the parameter of the function template (call it `P`) is compared to the type of the corresponding argument of the call (call it `A`), and an attempt is made to find a type for the template type argument, a template for the template template argument or a value for the template non-type argument, that will make `P` after substitution of the deduced type or value (call that the deduced `A`) compatible with the call argument. Type deduction is done independently for each parameter/argument pair, and the deduced template argument types, templates and values are then combined. If type deduction cannot be done for any parameter/argument pair, or if for any parameter/argument pair the deduction leads to more than one possible set of deduced types, templates or values, or if different parameter/argument pairs yield different deduced types, templates or values for a given template argument, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails.

3 If `P` is not a reference type:

— if `A` is an array type, the pointer type produced by the array-to-pointer standard conversion (_conv.array_) 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 (_conv.func_) 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 in place of `P` for type deduction.

4 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`.

— If `P` is a pointer or pointer to member type, `A` can be another pointer or pointer to member type that can be converted to the deduced `A` via a qualification conversion (_conv.qual_).

— 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 cannot be done otherwise. If they yield more than one possible deduced `A`, the type deduction fails. When deducing arguments in the context of taking the address of an overloaded function (_over.over_), these inexact deductions are not considered.

5 [*Example:* here is an example in which different parameter/argument pairs produce inconsistent template argument deductions:

```
template<class T> void f(T x, T y) { /* ... */ }
struct A { /* ... */ };
struct B : A { /* ... */ };
int g(A a, B b)
{
        f(a,b);  // error: T could be A or B
        f(b,a);  // error: T could be A or B
        f(a,a);  // ok: T is A
        f(b,b);  // ok: T is B
}
```

6 Here is an example where two template arguments are deduced from a single function parameter/argument pair. This can lead to conflicts that cause type deduction to fail:

```
template <class T, class U> void f(  T (*)( T, U, U )  );

int g1( int, float, float);
char g2( int, float, float);
int g3( int, char, float);

void r()
{

        f(g1);          // ok: T is int and U is float
        f(g2);          // error: T could be char or int
        f(g3);          // error: U could be char or float
}
```

7 Here is an example where a qualification conversion applies between the argument type on the function call and the deduced template argument type:

```
template<class T> void f(const T*) {}
int *p;
void s()
{
        f(p);  // f(const int *)
}
```

8 Here is an example where the template argument is used to instantiate a derived class type of the corresponding function parameter type:

```
template <class T> struct B { };
template <class T> struct D : public B<T> {};
struct D2 : public B<int> {};
template <class T> void f(B<T>&){}
void t()
{
        D<int> d;
        D2     d2;
        f(d);  // calls f(B<int>&)
        f(d2); // calls f(B<int>&)
}
```

—*end example*]

9 A template type argument `T`, a template template argument `TT` or a template non-type argument `i` can be deduced if `P` and `A` have one of the following forms:

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

where `(T)` represents argument lists where at least one argument type contains a `T`, and `()` represents argument lists where no parameter contains a `T`. Similarly, `<T>` represents template argument lists where at least one argument contains a `T`, `<i>` represents template argument lists where at least one argument contains an `i` and `<>` represents template argument lists where no argument contains a `T` or an `i`.

10 In a type which contains a *nested-name-specifier*, template argument values cannot be deduced for template parameters used within the *nested-name-specifier*. [*Example:*

```
template<int i, typename T>
T deduce(typename A<T>::X x,      // T is not deduced here
                  T       t,      // but T is deduced here
         typename B<i>::Y y);     // i is not deduced here
A<int> a;
B<77>  b;

int    x = deduce<77>(a.xm, 62, y.ym);
// T is deduced to be int, a.xm must be convertible to
// A<int>::X
// i is explicitly specified to be 77, y.ym must be convertible
// to B<77>::Y
```

—*end example*] When a template parameter is used in this context, an argument value that has been explicitly specified, or deduced from other arguments is used. If the value cannot be deduced elsewhere, and is not explicitly specified, the program is ill-formed. Conversions (_conv_) will be performed on a function argument that corresponds with a function parameter that contains only non-deducible template parameters and explicitly specified template parameters (14.8.1). These forms can be used in the same way as `T` is for further composition of types. [*Example:*

```
X<int> (*)(char[6])
```

is of the form

*class-template-name*`<T> (*)(`*type*`[i])`

which is a variant of

*type* `(*)(T)`

where type is `X<int>` and `T` is `char[6]`. ]

11 Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

12 A template type argument cannot be deduced from the type of a non-type *template-argument*. [*Example:*

```
template<class T, T i> void f(double a[10][i]);
int v[10][20];
f(v); // error: argument for template-parameter T cannot be deduced
```

—*end example*]

13 [*Note:* except for reference and pointer types, a major array bound is not part of a function parameter type and cannot be deduced from an argument:

```
template<int i> void f1(int a[10][i]);
template<int i> void f2(int a[i][20]);
template<int i> void f3(int (&a)[i][20]);

void g()
{
        int v[10][20];
        f1(v);     // ok: i deduced to be 20
        f1<20>(v); // ok
        f2(v);     // error: cannot deduce template-argument i
        f2<10>(v); // ok
        f3(v);     // ok: i deduced to be 10
}
```

—*end note*]

14 If, in the declaration of a function template with a non-type *template-parameter*, the non-type *template-parameter* is used in an expression in the function parameter-list, the corresponding *template-argument* shall always be explicitly specified because type deduction would otherwise always fail for such a *template-argument*. [*Example:*

```
template<int i> class A { /* ... */ };
template<short s> void g(A<s+1>);
void k() {
  A<1> a;
  g(a);        // error: deduction fails for expression s+1
  g<0>(a);     // ok
}
```

—*end example*]

15 If, in the declaration of a function template with a non-type *template-parameter,* the non-type *template-parameter* is used in an expression in the function parameter-list and, if the corresponding *template-argument* is deduced, the *template-argument* type shall match the type of the *template-parameter* exactly, except that a *template-argument* deduced from an array bound may be of any integral type.[5] [*Example:*

```
template<int i> class A { /* ... */ };
template<short s> void f(A<s>);
void k1() {
  A<1> a;
  f(a);        // error: deduction fails for conversion from int to short
  f<1>(a);     // ok
}
```

[5] Although the *template-argument* corresponding to a *template-parameter* of type `bool` may be deduced from an array bound, the resulting value will always be `true` because the array bound will be non-zero.

```
template<const short cs> class B { };
template<short s> void h(B<s>);
void k2() {
  B<1> b;
  g(b);        // ok: cv-qualifiers are ignored on template parameter types
}
```

—*end example*]

16 A *template-argument* can be deduced from a pointer to function or pointer to member function argument if the set of overloaded functions does not contain function templates and at most one of a set of overloaded functions provides a unique match. [*Example:*

```
template<class T> void f(void(*)(T,int));
template<class T> void foo(T,int);
void g(int,int);
void g(char,int);

void h(int,int,int);
void h(char,int);
int m()
{
        f(&g);    // error: ambiguous
        f(&h);    // ok: void h(char,int) is a unique match
        f(&foo);  // error: type deduction fails because foo is a template
}
```

—*end example*]

17 If function *template-argument*s are explicitly specified in a call, they shall be specified in declaration order of their corresponding *template-parameter*s. Trailing arguments can be left out of a list of explicit *template-arguments*. [*Example:*

```
template<class X, class Y, class Z> X f(Y,Z);
        void g()
        {
                f<int,char*,double>("aa",3.0);
                f<int,char*>("aa",3.0); // Z is deduced to be double
                f<int>("aa",3.0);       // Y is deduced to be char*, and
                                        // Z is deduced to be double
                f("aa",3.0);            // error: X cannot be deduced
        }
```

—*end example*]

18 A template *type-parameter* cannot be deduced from the type of a function default argument. [*Example:*

```
template <class T> void f(T = 5, T = 7);
void g()
{
        f(1);     // ok: call f<int>(1,7)
        f();      // error: cannot deduce T
        f<int>(); // ok: call f<int>(5,7)
}
```

—*end example*]

19 The *template-argument* corresponding to a template *template-parameter* is deduced from the type of the *template-argument* of a class template specialization used in the argument list of a function call. [*Example:*

```
template <template X<class T> > struct A { };
template <template X<class T> > void f(A<X>) { }
template<class T> struct B { };
A<B> ab;
f(ab); // calls f(A<B>)
```

—*end example*]

20 If trailing *template-argument*s are left unspecified in a function template explicit instantiation or explicit specialization (14.7.2, 14.7.3), the template arguments can be deduced from the function parameters according to the rules specified in this subclause. [*Note:* a default *template-argument* cannot be specified in a function template declaration or definition; therefore default *template-argument*s cannot be used to influence template argument deduction. ]

**Box 8**
core issue 842: Template argument deduction rules for template conversion functions are missing.

### 14.8.3 Overload resolution [temp.over]

1 A function template can be overloaded either by (non-template) functions of its name or by (other) function templates of the same name. When a call to that name is written (explicitly, or implicitly using the operator notation), template argument deduction (14.8.2) is performed for each function template to find the template argument values (if any) that can be used with that function template to instantiate a function template specialization that can be invoked with the call arguments. For each function template, if the argument deduction succeeds, the deduced *template-argument*s are used to instantiate a single function template specialization which is added to the candidate functions set to be used in overload resolution. If, for a given function template, argument deduction fails, no such function is added to the set of candidate functions for that template. The complete set of candidate functions includes all the function templates instantiated in this way and all of the non-template overloaded functions of the same name. The function template specializations are treated like any other functions in the remainder of overload resolution, except as explicitly noted.[6)]

2 [*Example:*

```
template<class T> T max(T a, T b) { return a>b?a:b; }

void f(int a, int b, char c, char d)
{
        int m1 = max(a,b);  // max(int a, int b)
        char m2 = max(c,d); // max(char a, char b)
        int m3 = max(a,c);  // error: cannot generate max(int,char)
}
```

3 Adding the non-template function

```
int max(int,int);
```

to the example above would resolve the third call, by providing a function that could be called for `max(a,c)` after using the standard conversion of `char` to `int` for `c`.

6) The parameters of function template specializations contain no template parameter types. The set of conversions allowed on deduced arguments is limited, because the argument deduction process produces function templates with parameters that either match the call arguments exactly or differ only in ways that can be bridged by the allowed limited conversions. Non-deduced arguments allow the full range of conversions. Note also that _over.match.best_ implies that a non-template function will be given preference over a template instantiation with the same parameter types.

4 Here is an example involving conversions on a function argument involved in *template-argument* deduction:

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

void g(B<int>& bi, D<int>& di)
{
        f(bi);  // f(bi)
        f(di);  // f( (B<int>&)di )
}
```

5 Here is an example involving conversions on a function argument not involved in *template-parameter* deduction:

```
template<class T> void f(T*,int);  // #1
template<class T> void f(T,char);  // #2

void h(int* pi, int i, char c)
{
        f(pi,i);  // #1: f<int>(pi,i)
        f(pi,c);  // #2: f<int*>(pi,c)

        f(i,c);   // #2: f<int>(i,c);
        f(i,i);   // #2: f<int>(i,char(i))
}
```

—*end example*]

6 Only the signature of a function template specialization is needed to enter the specialization in a set of candidate functions. Therefore only the function template declaration is needed to resolve a call for which a template specialization is a candidate. [*Example:*

```
template<class T> void f(T);    // declaration

void g()
{
        f("Annemarie"); // call of f<const char*>
}
```

The call of `f` is well-formed even if the template `f` is only declared and not defined at the point of the call. The program will be ill-formed unless a specialization for `f<const char*>`, either implicitly or explicitly generated, is present in some translation unit. ]