X3J16/95-0158
WG21/N0758

Template Issues and Proposed Resolutions
Revision 13

John H. Spicer
Edison Design Group, Inc.
jhs@edg.com

July 20, 1995

Revision History

Version 1 (93-0039/N0246) – March 5, 1993: Distributed in Portland and in the post-Portland mailing.
Version 2 (93-0074/N0281) – May 28, 1993: Distributed in pre-Munich mailing. Reflects tentative decisions made in Portland and additional issues added after the Portland meeting. In Portland, the extensions working group reviewed most of the issues from 1.1 to 2.8 and also reviewed 6.3.
Version 3 (93-0123/N0330) – September 28, 1993: Distributed in pre-San Jose mailing. Reflects decisions made in Munich. No new issues were added in this revision.
Version 4 (93-0183/N0330) – November 24, 1993: Distributed in post-San Jose mailing. Reflects decisions made in San Jose. Note that issues that have been closed as a result of formal motions in San Jose will be omitted from subsequent versions of this paper. In San Jose the extensions working group identified a number of issues that required additional work. These issues have not been addressed in this paper but will be addressed in the next revision.
Version 5 (94-0020/N0407) – January 25, 1994: Distributed in the Pre-San Diego mailing. The 41 closed issues have been removed, 20 have been added, and a few existing ones have been updated.
Version 6 (94-0068/N0455) – March 25, 1994: Distributed in the Post-San Diego mailing. Reflects decisions made in San Diego. Note that issues that have been closed as a result of formal motions in San Diego will be omitted from subsequent versions of this paper. In San Diego the extensions working group identified a number of issues that required additional work. These issues have not been addressed in this paper but will be addressed in the next revision.
Version 7 (94-0096/N0483) – June 1, 1994: Distributed in the Pre-Waterloo mailing. The 24 issues closed in version 6 have been removed and 16 new issues have been added.
Version 8 (94-0125/N0512) – November 3, 1994: Distributed in Valley Forge and in the post-Valley Forge mailing. Reflects decisions made in Waterloo. This version contains only issues closed in Waterloo. Version 9 will be distributed at the same time as version 8 and will contain the open issues and new issues.
Version 9 (94-0200/N0587) – November 5, 1994: Distributed in Valley Forge and in the post-Valley Forge mailing. Issues closed in version 8 have been removed and new issues have been added.
Version 10 (94-0212/N0599) – November 25, 1994: Distributed in the post-Valley Forge mailing. Reflects decisions made in Valley Forge. Includes a number of new issues supplied by

Erwin Unruh.
Version 11 (95-0007/N0607) – January 31, 1995: Distributed in the pre-Austin mailing. Includes a few new issues.
Version 12 (95-0101/N0701) – May 28, 1995: Distributed in the pre-Monterey mailing. Reflects decisions made in Austin. 9 issues have been closed, 12 new issues have been added.
Version 13 (95-0158/N0758) – July 20, 1995: Distributed in the post-Monterey mailing. Reflects decisions made in Monterey.

Introduction

This document attempts to clarify a number of template issues that are currently either undefined or incompletely specified. In general, this document addresses smaller issues.

Of the issues that are addressed, some are covered in far more detail than others. Some of the resolutions represent solid proposals while others are more like trial balloons. The more tentative proposals are so designated in the body of the document.

Even those resolutions that represent fairly solid proposals are *only* proposals. This document is not intended as a formal proposal of any specific language changes. Rather, it is intended as to be used as a framework for discussion of these issues. Hopefully this will ultimately result in formal proposals for language changes.

Organization of the Document

The document is organized in sections. Each section consists of a list of questions. Each question has an answer, a status, the version number of the first version of this document that included the question, and the version number of the last change in the question. This allows the reader to skip over questions that have not changed since the last time he or she read the document.

Acknowledgements

I would like to thank Bjarne Stroustrup who contributed greatly by providing issues, reviewing and improving upon proposed resolutions, and providing insights into other language changes that may impact templates. Thank you to Erwin Unruh, who has contributed to many of the issues, and who also contributed the "Erwin Unruh's Issues" section. Thank you to Mike Karasick and Lee Nackman (and possibly others) from IBM who contributed issues concerning name binding and member functions of partial specializations of class templates.

Summary of Issues

Because this is a rather long document this summary is provided to allow the reader to quickly find issues in which he or she may be interested. Note that closed issues have been removed from the body of the paper. Please refer to a previous version of the paper for additional information on these issues.

Template Parameters

1.1 Can template parameters have default arguments? (closed in version 4)

1.2 Where can default arguments for template parameters be specified? (closed in version 4)

1.3 Can a type parameter be used in the type declaration of a nontype parameter? (closed in version 4)

1.4 Can a nontype parameter as used above have a default argument? (closed in version 4)

1.5 Should it be possible to redeclare a template parameter name to mean something else inside a template definition? (closed in version 4)

1.6 Can the name of a nontype parameter be omitted? (closed in version 4)

1.7 Can the name of a type parameter be omitted? (closed in version 4)

1.8 Can a typedef appear in a template declaration? (closed in version 4)

1.9 Can a nontype parameter have a reference type? (closed in version 4)

1.10 Are qualifiers allowed on nontype parameters? (closed in version 4)

1.11 May a template parameter have the same name as the class template with which it is associated? (closed in version 4)

Class Template References

2.1 Can a nontype parameter that is not a reference be used as an lvalue or have its address taken? (closed in version 4)

2.2 Can the class template name be used as a synonym for the current instantiation inside a class template? (closed in version 4)

2.3 Can a class template have a template parameter as a base class? (closed in version 4)

2.4 Can a local type be used as a type argument of a class template? (closed in version 4)

2.5 Can a character string be a nontype argument? (closed in version 4)

2.6 Can any conversions be done on nontype actual arguments of class templates? (closed in version 6)

2.7 What causes a template class to be instantiated? (closed in version 4)

2.8 How can a class template name be used within the definition of the template? (closed in version 6)

2.9 The previous rule makes possible runaway recursive instantiations. How should an implementation prevent this? (closed in version 5)

2.10 At what point are names injected? (closed in version 6)

2.11 Does an array parameter decay to a pointer type? (closed in version 6)

2.12 What can be used as an actual argument for a parameter that is a reference? (closed in version 4)

2.13 Can template parameters be used in elaborated type specifiers? (closed in version 4)

2.14 Can a class template or function template be declared as a friend of a class? (closed in version 6)

2.15 Can template arguments be supplied in explicit destructor calls? (closed in version 4)

2.16 What happens if the same name is used for a template parameter of an out-of-class definition of a member of a class template and a member of the class? (closed in version 6)

2.17 What happens if the name of a template parameter of a class template is also the name of a member of one of its base classes? (closed in version 6)

2.18 When must a type used within a template be completed? (closed in version 6)

2.19 Must a specialization declaration precede the use of a class template in a context that requires only an incomplete type? (closed in version 6)

2.20 Proposal to defer error checking for `operator ->`. (closed in version 6)

2.21 When are names considered known in a template dependent base class? (closed in version 6)

2.22 Proposed revision to rules for explicit instantiation of all class members. (closed in version 8)

2.23 How does name injection interact with the semantics of friend declarations? (withdrawn - last in version 10)

2.24 Class template partial specialization clarification. (closed in version 13)

2.25 May a nested class within a class template be defined outside of the template? (closed in version 13)

2.26 Question: May a class nested within a template be declared as a template friend? (closed in version 13)

2.27 May a friend function be defined in a template friend declaration? (closed in version 13)

Function Templates

3.1 Can function templates have default function parameters? (closed in version 4)

3.2 Can the parameters with default arguments involve template parameters in their types? (closed in version 5)

3.3 Can a local type be used as a type argument of a template function? (closed in version 4)

3.4 Can any conversions be done when matching arguments to function templates? (closed in version 5)

3.5 The WP requires that every template parameter be used in an argument type of a function template. What constitutes a "use" of a template parameter in an argument type? (closed in version 4)

3.6 Can unnamed types be used as template arguments? (closed in version 4)

3.7 Can template parameters be used in qualified names in function template declarations? (closed in version 12)

3.8 Can a noninline function template be instantiated when referenced? (closed in version 4)

3.9 A proposal to allow conversions in function template calls. (closed in version 6)

3.10 What happens when the explicit specification of function template arguments results in an invalid type? (closed in version 6)

3.11 How do default arguments work when using new explicit specialization declarations? (closed in version 6)

3.12 How do old style specialization declarations interact with new style ones? (closed in version 6)

3.13 Revisiting default arguments. (closed in version 12)

3.14 What are the rules regarding use of the inline keyword in function template declarations? (closed in version 10)

3.15 How may elaborated type specifiers be used in function template declarations? (closed in version 8)

3.16 Clarification of template parameter deduction rules. (closed in version 8)

3.17 How may an overloaded function name be used as a function template argument in a context that requires parameter deduction? (closed in version 8)

3.18 Must a function template declaration be visible when an instance of the template is called? (closed in version 8) item[3.19] What are the rules regarding the deduction of template template parameters? (closed in version 8)

3.20 How are type/expression ambiguities resolved in explicitly qualified function template calls? (closed in version 10)

3.21 May template functions with the same signature coexist with one another? May a template function with a given signature coexist with a nontemplate function with the same signature. (closed in version 12)

3.22 Proposed rules for selecting between overloaded function templates (closed in version 12)

3.23 Binding of function and array types to template dependent reference parameters.

3.24 Clarification regarding nontype parameters deduced from array bounds. (closed in version 13)

3.25 Can a type parameter be deduced from the type of a nontype parameter? (closed in version 13)

3.26 What is the type of a constant deduced from an array bound? (closed in version 13)

3.27 Clarification of rules regarding expressions used as nontype arguments. (closed in version 13)

Member Function Templates

4.1 Are inline member functions that are not used by a given class template instance instantiated? (closed in version 4)

4.2 Can a noninline member function or a static data member be instantiated when referenced? (closed in version 4)

4.3 Must the template parameter names in a member function definition match the names used in the class definition? (closed in version 4)

4.4 What are the rules regarding use of the inline keyword in member function declarations? (closed in version 6)

4.5 How are default arguments for parameters of member functions of class templates handled? (closed in version 4)

4.6 Can a class template member function be redeclared outside of the class? (closed in version 6)

4.7 Can a member function of a class specialization be instantiated from a member function of the class template? (closed in version 8)

4.8 Can a template member function be declared in a specialization declaration? (closed in version 8)

4.9 Can a member function defined in a class template definition be specialized? (closed in version 8)

4.10 How are members of class templates declared and defined? (closed in version 13)

4.11 How are members functions of a partial specialization of a class template defined? (closed in version 13)

Class Template Specific Declarations and Definitions

5.1 Can you create a specific definition of a class template for which only a declaration has been seen? (closed in version 4)

5.2 Can you declare an incompletely defined object type that is a specific definition of a class template? (closed in version 4)

5.3 Can the class template name be used as a synonym for the current specific definition inside the specific definition? (closed in version 4)

5.4 Can a specific definition of a class template be a local class? (closed in version 4)

Other Issues

6.1 Should classes used as template arguments have external linkage? (closed in version 4)

6.2 When must errors in template definitions be issued and when must they not be issued? (closed in version 4)

6.3 What kinds of types may be used in a function template declaration while still being able to deduce the template argument types? (closed in version 4)

6.4 Can a static data member of a class template be declared with an incomplete array type? (closed in version 4)

6.5 How should template arguments that contain "`>`" be parsed? (closed in version 4)

6.6 Can template versions of `operator new` and `operator delete` be declared? (closed in version 4)

6.7 How can a name that is undefined at the point of its use in a template declaration be determined to be a type or nontype? (closed in version 4)

6.8 May template declarations be given a linkage specification other than C++. (closed in version 6)

6.9 Should there be a translation limit that specifies a minimum depth of recursive instantiation that must be supported? (closed in version 6)

6.10 Can a single template declaration declare more than one thing? (closed in version 6)

6.11 Can a storage class be specified in a template parameter declaration? (closed in version 6)

6.12 Can an incomplete type be used as a template argument? (closed in version 6)

6.13 Can a template nontype parameter have a void type? (closed in version 6)

6.14 Can a nontype parameter be a floating point type? (closed in version 6)

6.15 What kind of expressions may be used as nontype template arguments?

6.16 Can a template parameter be used in an explicit destructor call? (closed in version 6)

6.17 Can pointer to member types be used as nontype parameters? (closed in version 8)

6.18 Issues regarding declarations of specializations. (closed in version 12)

6.19 Clarification of explicit designation of a name as a type. (closed in version 8)

6.20 Template compilation model proposal. (withdrawn - last in version 7)

6.21 How is a dependent name known to be a template? (closed in version 12)

6.22 Interaction of templates and namespaces. (closed in version 10)

6.23 Floating point template parameters revisited. (closed in version 10)

6.24 May function types be used as template parameters? (closed in version 12)

6.25 WP clarification: overloaded functions as template arguments (closed in version 10)

6.26 WP clarification: access checking an template arguments (closed in version 10)

6.27 Name binding problems (closed in version 12)

6.28 Can a user-specialization be provided for an `operator ->` that cannot be instantiated? (closed in version 13)

6.29 How are names from template dependent base classes to be used? (withdrawn, last in version 12)

Erwin Unruh's Issues

7.1 Type deduction for conversion operators (closed in version 12)

7.2 How does type deduction interact with overloading (closed in version 13)

7.3 How does type deduction interact with conversions

7.4 What is the point of instantiation really?

7.5 Short addition to 3.17 (closed in version 13)

7.6 Type deduction with several results (closed in version 13)

Nontype Parameters for Function Templates

A proposal for nontype parameters for function templates as required by the `Bitset` class. (closed in version 4)

Class Template References

2.24 Class template partial specialization clarification.

Status: Approved in Monterey.

Answer: As is the case with complete specializations, a class template partial specialization must be declared before its first use in every translation unit in which it is referenced.

Version added: 12
Version updated: 12

2.25 Question: May a nested class within a class template be defined outside of the template?

Status: Approved in Monterey.

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

template <class T> struct A<T>::B {
        // Class definition
};
```

Answer: Classes that are members of templates may be defined outside of the class. To maintain semantic equivalence with members defined inside the class, the rules for instantiation of member classes have been changed.

Member template classes follow the same instantiation rules as nonmember template classes: they are only instantiated when an instantiation is required.

Version added: 12
Version updated: 13

2.26 Question: May a class nested within a template be declared as a template friend?

Status: Approved in Monterey.

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

struct A<char> {
struct B {};  // Not a friend because A<char> is explicitly
                        // specialized
};

class C {
        template <class T> friend struct A<T>::B;
};
```

Answer: Yes. Note that the friendship is only extended to `B`s that are members of generated `A`s.

Version added: 12
Version updated: 12

2.27 Question: May a friend function be defined in a template friend declaration?

Status: Approved in Monterey.

```
class A {
        template <class T> friend void f(A, T){}
};
```

Answer: Yes.

This was prohibited by the original template friend proposal from an earlier version of this paper. This was originally done to simplify both the definition and implementation of the feature. However, because the declaration is permitted to make use of names from the enclosing class, there actually is no such simplification and the rule is an unnecessary and pointless restriction.

Version added: 12
Version updated: 12

Function Templates

3.23 Binding of function and array types to template dependent reference parameters.

Status: Open

WP 14.10.2 [temp.deduct] says that array and function types do not decay when binding to a parameter that is a reference. The problem with this is it permits array types to be used in places where the template writer had not intended them to be used. For example, the HP STL distribution includes a `max` template that is defined as:

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

This works well for most types, but fails for array types such as string literals.

```
int main()
{
        char* x;
        x = max("hello", "there");  // T is char[6]
        x = max("hi", "there");     // fails because T is char[3]
                                    // and char[6]
}
```

What is intended is that the resulting function parameter type for `const T&` is `const char*&`. What happens with the current WP wording is that the resulting function parameter is `const char (&)[6]`. This causes a problem: the length of the two strings

must be identical for type deduction to succeed, and the return type will end up being a reference to array of the same size.

Answer: The proposed solution is to revise the deduction rules to say that an array or function type can only bind to a parameter that is declared with a reference to array or function type, as in the example that appears below.

More specifically, assuming `P` is the parameter type and `A` is the argument type: If `P` is a reference to an array type and `A` is an array type, or `P` is a reference to function type and `A` is a function type, and if the values of the all of the template parameters referenced by `P` can be deduced from `A`, then the original type of `A` is used for type deduction. Otherwise,

- if `A` is an array type, the result of the array to pointer decay is used in place of `A` for type deduction; otherwise,
- if `A` is a function type, the result of the function to pointer decay is used in place of `A` for type deduction.

```
template <class T, int I1, int I2>
T* f(T (&t1)[I1], T (&t2)[I2]);

int main()
{
        char* x;
        x = f("hello", "there");
}
```

This still permits binding of array types, but only in cases where that is explicitly indicated by the template writer.

Note that this illustrates another clarification that needs to be made. Major array bounds are part of the parameter type when the parameter is a reference. Consequently, nontype template parameters may be deduced from a major array bound in such cases.

Version added: 12
Version updated: 12

3.24 Clarification regarding nontype parameters deduced from array bounds.

Status: Approved in Monterey.

A nontype parameter may be deduced from an array bound that is part of the type of the argument. Typically, the major array component decays into a pointer, and cannot participate in deduction. When an argument is bound directly to a reference, the decay does not occur and the template parameter may be deduced from the major bound.

```
template <int I> void f(int (&t1)[I]);  // I can be deduced
template <int I> void g(int t1[I]);     // I cannot be deduced
template <int I> void h(int t1[2][I]);  // I can be deduced

int main()
{
        int n[5];
        int m[2][6];
```

```
        f(n);
        g<5>(n);  // I can be explicitly specified
        h(m);
}
```

Version added: 12
Version updated: 12

3.25 Question: Can a type parameter be deduced from the type of a nontype parameter?

Status: Approved in Monterey.

Answer: This should go without saying, but just in case... `T` cannot be deduced in the following example, but `f` could still be called by explicitly specifying the value of `T`.

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

Version added: 12
Version updated: 12

3.26 Question: What is the type of a constant deduced from an array bound?

Status: Approved in Monterey.

The purpose of this example is to illustrate that the type of a constant as used in a function argument type must match its declared type in order for the constant to be deduced. A constant value from an array bound is not considered to have any particular type; it will match any integral type. (For additional information about conversion of nontype template arguments, see 2.6 in N0455)

```
template <int I> struct A {};
template <short S> struct B {};

template <int I> void f(int i[2][I], A<I>);  // I can be deduced
template <short S> void g(int i[2][S], B<S>);// S can be deduced
template <short S> void h(A<S>, B<S>);  // S cannot be deduced

int main()
{
        int n[2][5];
        A<5> a5;
        B<5> b5;
        f(n, a5);
        g(n, b5);
        h(a5, b5);  // Error - S in A<S> has wrong type
        h<5>(a5, b5);  // OK
}
```

Version added: 12
Version updated: 12

3.27 Clarification of rules regarding expressions used as nontype arguments.

Status: Approved in Monterey.

14.10.2 paragraph 9 says "Nontype parameters shall not be used in expressions in the function declaration". This rule predates explicit specification of function template parameters. The rule should be revised to say that nontype parameters cannot be deduced when used in expressions in the function declaration.

As with other cases in template argument deduction, the arguments are processed independently of one another, so argument deduction fails on call #1 below. Note that such a failure does not by itself result in an error. Only if there is no other function that can satisfy the function parameters is there an error.

```
template <int I> struct A {};
template <int I> void f(A<I>, A<I+1>);

int main()
{
        A<1> a1;
        A<2> a2;
        f(a1, a2);  // #1 Error - no function matches the call
        f<1>(a1, a2);  // #2 OK
}
```

Version added: 12
Version updated: 12

Member Function Templates

4.10 Question: How are members of class templates declared and defined?

Status: Approved in Monterey as modified and with the addition of the `template <>` syntax as described in 95-0143/N0743.

Answer:

1. A specialization of a member function of a template class (but not a member template) is declared or defined using the normal function member function syntax (i.e., without use of the `<>` used in template function specializations). Note that some additional specialization syntax such as `template <>` or `specialise` would be useful in contexts like this to make specializations easier to spot.
2. A member template is defined outside of the class definition in much the same way as a nontemplate member, but the template parameter list of the class is followed by a second template declaration instead of being followed by a normal function declarator. Default arguments may not be supplied in these declarations.
3. A member template of a class template may be specialized for a given instance of the class as indicated in the example below. Default arguments may not be supplied in these declarations.

4. A specific instance of a member template may be specified using the normal template function specialization syntax. The <> is required. Default arguments may not be supplied in these declarations.

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

// Specialization - #1
void A<int>::f(int);

// Out of class definition - #2
template <class T> template <class X> void A<T>::g(T,X) {}

// Specialization of member template - #3
template <class X> void A<int>::g(int,X);

// Specialization of an instance of a member template
void A<int>::g<>(int, char);
```

Default argument rationale: It has been tentatively decided that default arguments may only appear on the initial declaration of a template. This rules out the use of default arguments on the definition or specialization of a member template (#2 and #3 above). It has further been decided that default arguments are not permitted on specialization of function templates. This rules out the use of default arguments on the declaration of a specialization of an instance of a member template (#4 above).

A specialization of a member function of a template class is different from either of these cases. It doesn't declare a template, nor is a member function of a class template really a function template (i.e., member functions of class templates do not participate in the function template matching process and in template argument deduction).

Version added: 11
Version updated: 11

4.11 Question: How are member functions of a partial specialization of a class template defined?

Status: Approved in Monterey.

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

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

template <class T> struct A<T, 1> {
        void g();
        void h();
```

```
};

template <class T> void A<T,1>::g(){}

A<int, 0> a0;  // Uses primary template
A<int, 1> a1;  // Uses partial specialization

void A<int, 0>::f(){}  // Specializes member of primary template
void A<int, 1>::g(){}  // Specializes member of partial specialization

int main()
{
        a0.f();  // OK
        a1.f();  // Error - A<int,1> has no member f
        a1.h();  // Error at link - no definition of A<int,1>::h
                 // provided.  The one from the primary template
                 // is not used.
}
```

Answer: The definition of a member function of a class template partial specialization uses a template parameter list that matches the template parameter list of the partial specialization with which it is associated (but, as always, the names of the parameters are not significant). The template argument used in the declarator must also match the template argument list in the definition of the partial specialization of the class template.

A complete specialization is automatically associated with the appropriate specialization by the implementation just as any other reference to an instance of the class template.

Version added: 12
Version updated: 12

Other Issues

6.28 Question: Can a user-specialization be provided for an `operator ->` that cannot be instantiated?

Status: Approved in Monterey, but then made obsolete by motion 13 in Monterey that relaxed the rules for the return type of `operator ->` in general.

This is a clarification. `operator->` can be declared in a class template provided it is not actually used or instantiated for instances for which the return type is invalid.

This issue clarifies that user specializations are also not permitted.

```
template <class T> struct A {
        T* operator->();
};
int* A<int>::operator->(){}  // Error
```

Answer: No.

Version added: 12
Version updated: 12

Erwin Unruh's Issues

Many thanks to Erwin Unruh who provided the following issues in finished Latex form! These issues were added to this document in version 10.

7.2 How does type deduction interact with overloading (ext-2320, Erwin Unruh)

Status: Approved in Monterey.

After determining the candidate functions, for each template function type deduction takes place. That type deduction can have different results:

1. A single function is chosen.
   In this case that function is the candidate replacing the template (It will also be a viable function).
2. No match is found.
   In this case there is no viable function instance.
3. Several functions (finite number) may be chosen.

```
template<class T> class A{};

template<class T> int f( A<T> );

class B : public A<char>, public A<int> {};

B b;

int i = f(b);
```

   In this example, both bases produce each a viable function from the template. The result is that this call is ambigous.
   There may be situations where two viable functions may be generated from a template, one of them better than the other. (One possibility may be a specialization which is derived from another specialization of the same template). Another sensible case would be where both functions are worse than a non-template function (add `f(B)` to the example).
   Resolution: No function produced from this template is considered for overloading.
4. The type deduction fails.
   This can be the case when an infinite number of viable functions is generated, a template argument cannot be deduced or a conflict between deductions arises (3.16).
   In theses cases no reasonable function can be chosen from that template. (See also 7.6)
   Resolution: No function produced from this template is considered for overloading.

Version added: 10
Version updated: 13

7.3 How does type deduction interact with conversions (ext-2320, Erwin Unruh)

Status: Open

At the moment I see the following problems, where templates enter the discussion of conversions.

1. Conversion from pointer to derived to pointer to base:

   Both the derived and the base class could be template classes. At this point there should be no big problem. Both classes must be complete to allow such a conversion. The base class must be instantiated for the derived class to be defined. The derived class must be instantiated whenever a pointer to it is subject to a conversion. (see point 2.7)

2. Conversion of pointers to member

   When solving the conversion of template arguments we left out member pointer. So pointer to member conversions cannot interfere with template type deduction. So source and target of such a conversion are fixed and it can be checked whether the types are completely defined.

   Proposal: When a pointer to a member of a template class may be the target of a conversion, that class will be instantiated.

3. Constructor templates

   This does have a very neat solution after the proposal for the section 13 is accepted (94-0080). Here the overload resolution goes back to itself whenever a user defined conversion comes into play. So the conversions itself are described using overload resolution. In this context it is relatively easy to incorporate constructor templates into that scheme.

4. Conversion templates

   The usage of conversion templates is discussed in Point 7.1. There is however an additional problem in the declaration matching. Consider the following example:

   ```
   class A {
       operator int();
   };
   class B : public A {
       template <class T> operator T ();
   };
   class C : public B {
       operator char ();
   };
   ```

   The question is whether the template hides the conversion in the base class and whether a declaration in the derived class may hide (an instance of) the template. The problem arises since the return type of the conversion operator is considered its name.

   Proposal: The template conversion does hide only the conversions which have an exact match. A program is ill-formed, if a conversion template is potentially hiding (or being hidden by) a conversion for which the type deduction can not be done.

   To elaborate that rule: If there is a potential hiding between a template and a normal conversion, try type deduction. If that results in a match, fine. If the result is that there is definitely no match (`int` vs. `T*`), fine. Otherwise, there is a problem!

Hiding between two template conversions should be discussed when the topic of partial specialization is resolved. Is it allowed to have two template conversions in the same class ?

Version added: 10
Version updated: 10

7.4 What is the point of instantiation really? (ext-2547, Erwin Unruh)

Status: Tentatively approved in Monterey.

Answer: The point of instantiation is the point of use, except that local scopes are not considered for name lookup and name injection.

Discussion: The present rules for the template name binding have a uncomfortable bit. Consider the following example:

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

void h()
{
    extern void g(char);
    f('a');        // error
}

// \#1

void g(int i)
{
    f(i);          // error ??
}
```

With the present rules both instantiations fail. The first `f<char>` should fail, since no `g` is in (global) scope at the point of instantiation and the local one is ignored (with very good reason).

The second however is not so clear cut. The WP says the point of instantiation is #1 and there is no `g` in scope. On the other hand one could argue that the function `g` is known at the call as it is not local.

This topic is currently (Nov. 1994) still under discussion and should be reviewed in a later version. It also interacts with the problem of name injection.

Version added: 10
Version updated: 10

7.5 Short addition to 3.17 (ext-2455, Erwin Unruh)

Status: Approved in Monterey.

I want to add a new point to the example: It reads now

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

void g(int, int);     // g1
void g(int, char);    // g2

template <class T> void h(int, T);

int main()
{
    f(g);    // O.k. chooses g1
    f(h);    // ??
}
```

The call with g is no problem. The function g2 can be no match, since its second parameter does not match the second parameter of the function pointer.

The call with h looks similar. Here is only one function, which can be the argument to the call. This is the case since the template parameters involve different parameters of the function pointer. But to solve this problem we have to do type deduction on both f and h in parallel. This looks very strange.

So I propose to add a new rule (or a variant of the one in spicer's list):

> If a template argument is deduced from a function parameter of a pointer to function type, the function argument must be an expression of a single type, or a name of an overloaded function which does not contain template functions, otherwise type deduction fails (see 7.2).

There is a short note in 3.17 which supports this view. It reads: "... (i.e., in which no type deduction is required)."

Version added: 10
Version updated: 10

7.6 Type deduction with several results (ext-2436, Erwin Unruh)

Status: Approved in Monterey.

After the adoption of 3.9 (conversion of template arguments) and 3.16 (deducing from several arguments) we have a strange situation which is not handled by the rules.

See the following example:

```
template <class T> class B {};

class D : public B<int>, public B<float> {};

template <class T> int f ( B<T> );
template <class T> int g ( B<T> , B<T> );

D d;

int i = f(d);
int j = g(d,d);
```

Let's first look at f. The argument is a class which does not match. It has two base classes which can be reached and which would match. So there are two instances of the template which can be used for overload resolution. (In this case they are ambiguous, but another parameter could have solved the ambiguity).

The function g gets more interesting. For both parameters we can deduce the argument type for the template. Counting all tuples they colud be:

1. int and int
2. int and float
3. float and int
4. float and float

Only the first and last possibilities really lead to correct functions.

If we are going to allow several functions, we have to describe a complete algorithm of how to find the viable functions.

I propose to have a simple rule, which may be different from ordinary functions. The rule is: Type deduction from a single argument must lead to at most one choice of a template argument. If two different argument values can be deduced, type deduction fails (See also 7.2).

I am not very firm on this conclusion and would like to hear more arguments!

Version added: 10
Version updated: 10