ISO Doc No: WG21/N0338
ANSI Doc No: X3J16/93-0131
Date: September 23, 1993
Reply To: Erwin Unruh
erwin.unruh@mch.sni.de

Instantiation of Templates

Erwin Unruh

Siemens Nixdorf Informatiossysteme AG
Department of Software Development Systems
C/C++ Front-End Laboratory SU BS2000 SD 224
Otto-Hahn-Ring 6
D–81739 München
Germany

1 Introduction

A very important point of templates is the determination of the instances. It must be clear, which instances are generated and where the binding points are, which are taken for the name lookup.

I will use the word *instance* to refer to a combination of a template and a specific list of arguments. Two instances are the same, if and only if both the templates and the argument lists are the same.

The main idea is to incorporate the concept of the *lifetime* of an instance. Some instances have a lifetime which is limited to a translation unit. For these the ODR is needed to prevent different realisations. Others have a lifetime which does not end (functions). Their lifetime still has a beginning and ends at the time of linking.

I have divided the possible templates into three groups: class templates without regarding function bodies, functions including member functions and static member of class templates.

Further I look at specializations and instantiation requests. The binding points found must be moved when they do not appear at the global scope. Instantiation is done only at global scope.

I have left some problems to be discussed in San Jose. These are cases where several possibilities seem reasonable. I have noted my preference but I am open to change these points. They are marked with **OPEN**.

2 Class Template Instance

When discussing instantiation, I will separate the class *interface* from the rest of the class. The interface of a class consists of all declarations in the class. It does not contain any definition. The instantiation of member functions and static data member is discussed separately. A nested class is instantiated on its own.

When looking at a class template instance, the first relevant part of the program is the declaration of its template:

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

From now on it is possible to create an instance from this template. When the template name is followed by a specific list of arguments, an instance is refered. A specific instance begins its life at the first place, where it is mentioned. As an example, we can declare a pointer to an instance:

```
A<int> *pa;
```

This is allowed since no class definition is needed to declare a pointer to a class. Even if the template contains more information, the only information about the instance `A<int>` is that it exists. There is no information about its members. Then the body of the template is supplied:

```
template <class T> class A {
    int a;
    T b;
    static int s;
    inline T mfi (void) { return b; }  // mfi = member f inline
    int mfo (void);                    // mfo = member f outline
};
```

This body constitutes the declaration of three more templates for the member `A<..>::s`, `A<..>::mfi` and `A<..>::mfo`. At the point of this definition the only check to be done is the check, whether the parameters to the template correspond to the former declaration. There is no instantiation of `A<int>`. Even if at this place the declaration

```
A<double> *pb;
```

occurs, no member is instantiated. We only know that an instance `A<double>` will exist, but do not do any checking. It is only checked, whether the argument `double` is a valid argument for the parameter `class T`.

At some point in the program the definition of the instance `A<int>` is needed. It is needed when either an object of this type is created or in any other circumstance where a complete class is needed. It is also generated, when the instance itself or a pointer to it is subject to any type conversion, whether as source or target. At this point the interface of the class is instantiated. This instantiation includes the declaration of all data members, function members, default arguments of member functions but excludes the bodies of the member functions.

> **OPEN**: Should an instance be instantiated, if a pointer to it is subject to type conversion?
>
> John Spicer mentioned the problem (as No 2.7), that a conversion from a pointer to a derived class to a pointer to one of its bases needs a fully defined class. I have been more general and included every conversion.

From this point on the instance is fully defined and useable. The class template instance ends its lifetime at the end of the translation unit. (The instance will exist in many translation units, but they are handled independently. The ODR is taken to keep them the same.)

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

// in the following cases no instantiation is done

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

// in the following cases do instantiation

B<int> b;
f(42);
g(42);
void h(B<int> =0);
```

Name binding for classes is done at the point where the interface is instantiated. This is the point, where the definition is needed. It must not be done earlier. Here is an example where this distinction makes a difference:

```
class X;
B<X> *pbx;                    // do NOT instantiate at this point!
class X { ... };
B<X> bx;                      // now do instantiation
```

If instantiation is done too early, the class `X` is not defined. Since the definition is needed to generate the instance, the instantiation will fail. There are more complex examples, where moving the point of instantiation changes the semantics of an instance.

3 Function Template Instance

This section also includes Member Functions.

For function templates there is no difference between determining the parameter of the template and the declaration of the interface. Both are done in a single template declaration. For member functions this is part of the class definition.

```
template <class T> class A {
    int a;
    T b;
    static int s;
    inline T mfi (void) { return b; }  // mfi = member f inline
    int mfo (void);                    // mfo = member f outline
};

template <class T> inline T fi (T a, T b) { return a>b?b:a; }
template <class T> T fo (T a, T b =0 );
```

For normal function templates we have a check at this point. It is checked whether the arguments to the template can be deduced from the argument types of the function. This is done for the template per se, no look is done to any specific instance.

> **OPEN**: To which point should we set the start of the lifetime of a function template instance? It will be somewhere between the template declaration and the usage of the instance in an expression.
>
> If we take the template declaration, we have an inconsistency to the classes. On the other hand every possible template function takes part in overload resolution. So the instances exist, at least virtually.

The next point in the life of an instance is the first time, where it is mentioned directly or indirectly. For member functions this is the time, where the instance of the class definition is created. For the other functions there are several possibilities. One of them is a declaration of the function instance, as in

```
int fo(int, int);                // instance of fo with int
```

The other possibility is the usage of the function in an expression. It does not matter whether the use is a call or the taking of its address.

> **OPEN**: Is it conveniend to have a binding point at the place of declaration?
>
> For classes the binding point is at the point of (real) usage. It should be the same for functions. But then we have a problem with the handling of default arguments.

At this point the declaration for the instance is generated. The arguments of the template are determined from the types of the arguments. Then the default arguments are instantiated. It does not matter, whether the default arguments are used or not.

If an inline functions is used in an expression, also its body is instantiated. It is available only in this translation unit. The call of an inline function needs its definition to be known at the point of the call. Since `inline` has the same semantics as `static`, the same rule applies to static functions.

The body of an extern function is not instantiated at this point. Instead this point is remembered since it is the point of name binding for the instance. The decision whether to generate the instance is defered but the binding refers to this point. Definitions for extern function template instances are not generated during the analysis of a single translation unit.

When the program is complete (i.e. at the point of linking) the definitions of function template instances are generated. The name binding is taken from the remembered point described above. If the binding is different for any two translation units, the program is ill-formed due to violation of the ODR.

If a function is declared both extern and inline, it is handled like an extern function. The inline property is ignored, since it does not have any effect. (It is open, whether such a function is allowed; see also 4.4 from John Spicers list).

> **OPEN**: I do not have a coherent concept for default arguments.
>
> There are several possibilities how to handle default arguments. They may be checked at the first declaration, at the binding point or at demand. In the last case an unused default argument will not be checked.

3.1 Virtual Member Function

Virtual member functions are a more difficult problem. Some of their calling points are determined at run time. So it is not possible for a compiler to statically check, whether a virtual function is called. The call may be in a different translation unit, where the template is not present. To solve (part of) the problem I state the following rule:

A virtual function is considered "used" at the point, where its class is instantiated. This is also the point of name binding.

This rule has the drawback, that all names used by virtual functions must be available at the point of the class instantiation. This is different from the rules for non-virtual functions.

> **OPEN**: The point of the first static use of a virtual function is an additional binding point. If the name binding is different at this point, the program is ill-formed due to violation of the ODR.
>
> This rule may help people to keep their programs correct, when they add or remove the `virtual` specifier.

> **OPEN**: The compiler should be given the option to optimise away an instance of a virtual function if he can determine that it is never used. If such an optimisation is done, even the error diagnostics should not be mandatory. (See section 8)

4 Instance for Static Class Member

Static data member of classes are at most handled like non-inline static member functions. The only difference is that the body to lay down is the call to the constructor for that member. The instantiation is delayed until program completion time.

The binding point for static class members is determined as for member functions. It is the point of usage in an expression.

> **OPEN**: Their is an additional point of name binding at the point of instantiation of the class.
>
> We have a problem, when a static member of a class instance is never used. There is no point of name binding. If it is not instantiated, its constructor is never called.
>
> For normal classes a definition of a static member must be present. So we should set the same requirement for templates.

5 Moving the Points

It is not recommended that an instantiation occurs in the middle of another declaration. To avoid that, all usage points mentioned above are moved out of declarations. They are moved to the point *before the first global declaration that mentions it*.

> **OPEN**: Do we move an instantiation out of a namespace declaration?
>
> The template itself may be inside the namespace. So moving it out of it may be wrong. The simple rule should be changed to say, out of which scopes the instantiation is moved really. This can be very important, since there are possibilities to add names to the global scope.

The compiler is not obliged to really do the move. The analysis and instantion has do occur *as if* the usage was just before the declaration.

As one instantiation can cause the instantiation of another template, the instantiations at any one point (which may be before a global declaration and subsume the usages from a big function) are ordered. This ordering should ensure that no instantiation needs an instance which is generated later. This can result in some instantiation split into two parts: first the pure mentioning of its existence and afterwards instantiating the interface. For inline functions there may be a third part: the instantiation of its body. To generate the ordering, the interface of a function can be split. The second part will contain the default arguments. In the first part only the presence of default arguments is given.

If such an ordering is not possible, there is a recursion in instantiation and the program is ill-formed.

The rules and the wording finally accepted in the working paper must be choosen very carefully. I do not know, whether there is a possibility of a program changing semantics due to a different ordering of instantiations.

> **OPEN**: Maybe there is a problem, when an instance uses another instance and causes it to be instantiated. Where is the binding point?
>
> The instantiation of a template function instance is outside of any translation unit. It is formed somewhere between the translation of the last unit and the start of running the program. So we have difficulties to find the real binding point for it.

To clarify the description here are two examples:

```
template <class T> class B;
template <class T> class A {
    B<T> b;
};
template <class T> class B {
    A<T> *pa;
};

A<int> vara;
```

At this declaration the following instantiations will be done (in this order):

1. The template instance `A<int>` is set to be existent.
2. The template instance `B<int>` is instantiated completely.
3. The template instance `A<int>` is instantiated completely.
4. The variable `vara` is defined.

Here the instance `A<int>` was split into two parts. This splitting was necessary since there is a circular dependency between `A` and `B`. This dependency is broken with the pointer declaration in `B`.

```
template <class T> T g(T, T= f(0) );
template <class T> inline T f(T a) { return a==0 ? 0 : g(a,a); }

int var = g(5);
```

Here the order of instantiations may be:

1. The prototype of the instance `g<int>` is generated, excluding the value of the default argument.
2. The instance `f<int>` is completely instantiated, including the body.
3. The default argument of `g<int>` is added.

6 Specialization

A specialization has to occur after the first mentioning of the template. It is an open question (No. 5.2 from J. Spicers list) whether a specialization is allowed before the interface definition of the template.

This is a change to the behavior of most of the present compilers. They interpret any function, which has the same name as the template and suitable parameters, as a specializations. My rule requires, that the template declaration is available in the translation unit in which the specialization is defined.

> **OPEN**: A specialization must have been declared at every point it is used.
>
> This requirement is too strict. One serious use of specialization is to build a system with templates and replace the critical function with spezialisations. It may be conveniend to correct false semantics or to improve a function which is heavily used. It would hinder the programmer if he has to declare such a function and recompile half of his program.

A specialisation must occur before the instantiation is done. This results in several regions for the different templates:

- For class templates it must occur before the first *object declaration* of that class is done.
- For inline function templates it must occur before the function is first used in an expression.
- For non-inline functions it may occur anywhere. Since those functions are instantiated at program completion, any definition in any translation unit can serve as a specialization.
- For static class members the specialization can be anywhere in the program. These objects are also instantiated at program completion time.

7 Instantiation Request

In my last paper (93-0067=N0274) I have asked for an *instantiation request*. Bjarne has also written a short note regarding the usability of such a construct. Reasons to incorporate such a request can be found there.

The version of request I propose in this paper differs from that in my earlier paper. I have adopted the very moderate version of Bjarne.

The instantiation of an entity can be requested. Requests are resolved for single instances, so the class interface is requested independently from its member functions.

The primary result of an instantiation request is as follows:

- For a class: the interface of the class is instantiated, as if an object of that class is declared/defined.
- For an inline function: interface and body of the function are instantiated. Both are checked for errors.
- For a non-inline function: the interface of the function is instantiated. This includes the default arguments. The point of request is noted as one of the binding points for that instance.

A program may contain several requests for the same instance. A request is not required for an instance to be generated.

> **OPEN**: If a program contains an instantiation request and a specialization for the same instance, the program is ill-formed.
>
> This rule would allow the compiler to generate code at the point of the request. It is still responsible to avoid multiple instantiations,

but may do optimisations. So it may inline normal functions, as no specialization is possible.

It is even a possible feature to the programmer. He also can be sure that no specilization will take control over his template function.

OPEN: If an instance is requested, the compiler may check it. If the instantiation is ill-formed it may issue a diagnostic, not regarding whether a specialization is present in the program or not.

If the former point is not taken, this possibility gives the compiler the right to check the instantiation and issue the errors occuring. It also gives the writer of the class the evidence that his template will work for the disired parameters, whether it is specialized or not.

OPEN: All definitions needed to analyse the template instance must be available at the point of an instantiation request. This includes the definiton of the template itself, of all the types used as its parameters and other classes, if their definiton is needed.

This point was chosen to allow the compiler a check of the template without looking at any other translation unit. It is needed to gain the benefit of *check at translation*. If it is not fulfilled, the requested instance can only be check at the time of linking, where all the definitions are available.

OPEN: The name binding from the point of an instantiation request takes precedence over normal name binding points. It is no error to have a conflict between them. The binding resolution of two request must be identical (ODR).

This rule will allow the user to help the compiler. If the binding rules for a specific instance are very weird, the programmer can issue an instantiation request and be sure of the name binding.

It is also questionable, since the presence of an instantiation request not only effects the validity of a program, it also can change the semantics. This may be the case when one use-point and one request-point are present with different name bindings.

Most compilers will generate code for requested instantiations directly. This code can be used to build libraries and shorten compile time. This possible kind of usage is not inside the scope of the standard.

It may be convenient to have a syntax which groups some instantiation requests. As an example a class interface can be grouped with all its inline member functions. A grouping of all member functions and static members of a class may be of equal worth.

8 Error Checking

Checking is done only at the point of instantiation. If some part of a template or an instance is ill-formed a diagnostic may not be issued before the instantiation of that part is done. This is covered as No 6.2 by J. Spicers list and should be decided there.

If the ODR is not checked by a compiler and an instantiation request is present for an instance, the compiler must use the name binding from the point of the request (any one if several are present).

9 Related Work

The following documents have related work in them:

1. Stroustrup, 93-0081=N0288, Major Template Issues, section (I=instantiation)
2. Spicer, 93-0074=N0281, Template Issues and Proposed Resolutions,

   major points in sections 2.7, 3.8, 4.1, 4.2, 6.2

   minor points in sections 2.9, 4.4, 5.1, 5.2, 5.4
3. Unruh, 93-0067=N0274, Instantiation Request

From this list, the arguments in (1) and (3) are incorporated in this paper, from (2) the questions in sections 2.9, 4.4, 5.4 and 6.2 remain open.