Where Does the WP Require Complete Types

Jerry Schwarz

X3J16/96-0033 WG21/N0851

I intend this paper to examine all sections in the WP that require or ought to require that class types be complete by some point. That is, where a definition must have occurred before that point. If I have omited any (and there is a non-trivial probability that I have) then I hope someone will point it out.

I also propose some changes to these sections of the WP. Most of these proposed changes are editorial, but it is useful to mention them here.

There are a few places where there are substantive issues. I have highlighted those sections containing substantive issues with *****. These occur in sections 3.1, 3.9, 5.1, 5.2.7, 5.5, and 14.3.2. The most delicate issues relate to templates. (Section numbers are those of the September 95 WP.)

I sometimes need a phrase to refer to all the relevant requirements in the WP. I am using "context at which a class type needs to be complete", but this is wordy, a little vague, and might be hard to search for automatically. I would like to replace it with a simple phrase and then go in and use that phrase at each point enumerated here. Unfortunately, I'm not happy with any of the phrases I have invented. Maybe someone else can suggest one.

The proposals in this paper are numbered by the section they would affect and (to allow reference when more than one proposal affects a particular section) a letter.

The notation "pN" where N is a number refers to paragraph N of the section being discussed.

3.1[basic.def] *******

I propose to add to this section a new paragraph.

> (A) A program is ill-formed if the definition of any data object of a non template class gives the object a type that is incomplete at the point of declaration (see 3.9[basic.types]); or if the declaration of a non-static data member of a class gives the member an incomplete type.
>
> (B) Members of a class template may be given incomplete types when the template is defined, but section 14.3.2 requires that certain types be complete at any point of instantiation.
>
> [Example
>
> ```
> class I;
> class A;
> template<class T> class X {
>     T i;
>     A a; // allowed even though A is not dependent
> } ;
> X<I> xi ;
>     // X<I> is "complete", but instantiation of X<T>
>     // requires that I be complete and so the example
>     // is ill-formed.
> class I { } ;
> ```
>
>]

Something similar to 3.1A is currently said in section 3.9, but I

think this section is the more appropriate place to say it, and that my proposed wording is clearer than the existing wording in 3.9.

The deliberate exclusion of members of class templates in 3.1B is substantive, but represents my best reading of the current WP. It says to defer the completeness requirement as much as possible, and allows the declaration of X in the above example.

An interesting consequence is that explicit specializations are treated differently than ordinary classes.

```
class A ;
class B {
    A a ; // ill-formed.  Even if B is never used, the declaration
          // of a member requires that A be complete.
};
template<class T> X ;
template<> X<A> { // explicit specialization
    A a ;  // well-formed.
};
```

This is certainly a change from existing practice, but I think is consistent with the current WP.

3.2[basic.def.odr]

p4 currently states an unclear requirement on when a definition of a class is required. I propose to replace this with an example but no normative text. The example would contain one of each of the cases enumerated in this paper. It is best not to attempt a complete enumeration in normative text. Even if we get it correct today, we might decide to add something tomorrow and forget to change the text in 3.2 An example is a good way to put the list into the WP without risking a contradiction if we have forgotten something.

3.9[basic.types] *******

p6 says "Arrays of unknown size ... are incomplete".

This creates a slight definitional problem because it doesn’t take into account the possibility that the element type might be incomplete. For example

```
class X ;
extern X a[10] ; // type of a is incomplete.
```

I propose to replace this with

> (A) Arrays of unknown size or whose element type is incomplete ... are incomplete

3.9[basic.types] *****

We need to address the question of when template specializations are complete. This paper enumerates contexts where the WP requires some type to be complete or disallows an incomplete type. What we really mean is that at some place in a translation unit the type must be complete or that, if it is a template specialization, it can be instantiated. We could either go through and edit the WP everywhere to say that, or we can try to patch the definition of "complete type".

I propose to do the latter here with a new paragraph:

(B) A class specialization(see 14.3) is regarded as complete if the definition of its template has been seen.

The point of instantiation of a class specialization (see 14.3.2) may create requirement that other classes be complete.

So according to this definition a specialization might be "complete", but still unusable in contexts that normally require a complete type.

For example

```
template<class T> class X ;
template<class T> class Y {
  X<T> x;
};
Y<int> yi ; // Y<int> is "complete" but unusable because X<int> is
            // not complete.
```

3.9[basic.types]

p6 says: "No object shall be created to have incomplete type". I think this statement is vague, since it is unclear exactly what program constructs "create an object". I think its intent is covered by my proposed additions to 3.1 and 14.3.2.

(C) I propose to delete this sentence or replace it with a reference to 3.1 and 14.3.2.

3.9[basic.types]

p7 says a "... classes that have been declared but not defined are called incomplete types".

I propose to expand on this slightly

(D) Within a class'es definition it is regarded as complete within function bodies, default arguments and constructor class'es ctor-initializers (including such things in nested classes.) Otherwise it is regarded as incomplete within its own definition.

This description is taken from 3.3.6 where it is used to describe the scope of members.

4.1[conv.lvalue]

This section makes ill-formed a program that requires an lvalue to rvalue conversion of an expression with incomplete type.

5[expr]

p10 says "Whenever an lvalue expression appears as an operand of an operator that expects an rvalue ..."

What seems to be implicit is that all operands are rvalues unless otherwise noted. It is editorial to make this implicit assumption explicit, and I suggest below some places where such explicit statements would clarify that a context requires a completed class type.

5.1[expr.prim] *********

p8 describes the scope operator(::). But it doesn't require the

class before the :: to be complete. I propose to add

> (A) A program is ill formed if the nested-name-specifier names an incomplete class.

Note that this taken together with 3.9D implies

```
class X {
    enum E { z = 0 } ;
    int a[z] ; // well formed
    int b[X::z] ; // ill-formed
};
```

I'm not sure whether this is a change. It is certainly a substantive issue. A special exception to 5.1A could be made for this purpose, but I would prefer to keep the simple rule.

5.2.1[expr.sub]

WP requires complete type for "element".

No exception is made for the common idiom &x[0].

5.2.2[expr.call]

p2 requires a complete type for return type. As written it applies only to explicit calls and not to implicitly called conversions. See my proposal for 13.3

This section says nothing about arguments, that seems to be covered by my proposed words in in 12.8, but I suggest adding something to p3 anyway. Specifically:

> (A) If an argument is copied (see 12.8) it shall have a complete type.

I deliberately am not proposing to take this into account in determining viable functions. I think it would be a mistake to allow completing a type to change overload resolution.

5.2.3[expr.type.conv]

With regard to T(e1,....) the WP says "the type shall be a class with a suitably declared constructor". This seems to require the type to be completed, although I don't think it would hurt to change "class" to "complete class".

However with regards to T() it doesn't say anything. I propose to add

> (A) If T is a class type it shall be complete.

5.2.4[expr.ref]

WP requires type of object in "." or "->" to be complete.

5.2.6[expr.dynamic.cast]

In dynamic_cast<T>(v) the WP requires T to be a ptr to or ref of a complete class type, and v to be a ptr to or lvalue of a complete class type.

5.2.7[expr.typeid] *********

I'm not sure what the intention is here. Do we allow typeid's for incomplete type? The WP currently doesn't explicitly require this

although it hints at it. It says you do certain things if certain types are polymorphic. It's possible that you can check the runtime data structures to determine if the type is polymorphic, but otherwise we would need to require a complete type. So as a straw proposal I have

> (A) The type of the expression shall not be an incomplete class type.

That is, I don't feel strongly about this one way or the other, but I think we should make an explicit decision and if we decide not to make this change then I believe a footnote is desirable to indicate that this is an explicit decision.

5.2.8[expr.static.cast]

Static cast's do not impose any requirements on completeness, but the WP does allow certain conversions to be applied to "class types" which obviously require knowledge of the definitions. I propose to change these references from "class type" to "complete class type".

It isn't clear when v is subject to rvalue conversions. p5 clearly assumes that it isn't always subject to such conversions. The case that is relevant here is casting to void. I propose a footnote to p4.

> (A) Casting to void does not subject the operand to rvalue conversions.

5.3.1[expr.unary.op]

unary * requires a complete type.

No exception is made here for the common idiom

```
class X ;
X* f() ;
X& xr = &f();
```

However, Josee tells me that such an exception was accepted in Tokyo and will be in the next version of the WP.

5.3.3[expr.sizeof]

The WP forbids the operand to have incomplete type, or that when applied to a typeid, the type to be incomplete.

5.3.4[expr.new]

The WP requires a complete type for new expressions.

5.3.5[expr.delete]

The WP explicitly allows deleting of pointer to incomplete type.

5.4[expr.cast]

The WP applies semantic requirements of other forms of cast.

This section also contains the same glitch as 5.2.8. Namely when describing possible conversions there is a reference to "class type" that ought to read "complete class type"

> (A) change "class type" to "complete class type" as appropriate

5.5[expr.mptr.oper] ********

This deals with using pointer to members. The WP does not require that the type of the object be complete. This imposes some severe constraints on the representation of pointer to members. I am not certain whether all existing implementations can satisfy this constraint.

As a straw proposal I propose adding

(A) The type T shall be a complete class type.

5.7[expr.add]

p1 and p2 require pointers to completely defined object types in the relevant situations.

No exception is arithmetic involving a constant 0.

5.17[expr.ass]

There is no explicit statement that the type of an assignment cannot be an incomplete class type. It may be implied because the rhs is a rvalue. But that isn't clearly stated either. So I propose adding (to p3).

(A) The left operand shall not be of incomplete class type.

6.2[stmt.expr]

The WP does not require that the expression have a complete type. I propose to add.

(A) The expression is evaluated as an rvalue.

This imposes (somewhat indirectly) the requirement that it not be an incomplete class type.

7.1.5.2[dcl.type.simple]

This is where the WP allows qualified names to designate types.

For the purposes of this discussion I propose to add

(A) A program is ill-formed if the nested-name-specifier is present and names an incomplete class.

Note that this section needs to be expanded that it contains a description of the lookup. (The discussion in 5.1 can presumably be used as a model)

10[class.derived]

The WP requires that the base class be "previously defined".

(A) I think this should be changed to "complete class" for consistency.

13.3 [over.match]

This sections discusses the implicit calling of functions. I propose to add a new paragraph:

(A) If overload resolution succeeds then the return type of the selected function and the type of any argument that is copied shall be complete.

I don't think this is addressed by the requirements in 5.2.2 because those apply only to explicit function calls.

I do not propose to take this requirement into account when determining viable functions. I don't think whether a class has been completed should effect the result of overload resolution.

14.3.2 [temp.point]

This section is the critical one for understanding how the requirements for complete types interact with templates. I apologize that I haven't been following the discussion of the "extensions" group in as great a detail as I might so some of the issues I raise here may already have been resolved.

p1 says that the point of instantiation is determine by "the first use of a template requiring its definition." But an examination of the enumeration in this paper will reveal a great many points where a class must be complete but the type is not named. I think this requires some editorial work, something along the lines of

(A) the first program context that requires a generated class to be a complete type.

But I'm not sure about the vocabulary here. The section refers repeatedly to point of instantiation for a template, where I would have thought it should be point of instantiation for a specialization.

14.3.2 [temp.point] *****

By virtue of making a context a point of instantiation we may generate more "needs" for instantiations. I think p9 is trying to deal with that issue. It says "The point of instantiation for a template used inside another template ..." But this seems to create too many points of instantiation.

```
template<class T> class X ;
template<class T> class Y { X<T>* p ; } ;
Y< int > yx ; // should this be a point of instantiation for X<int>
```

By p12 if the above is a point of instantiation for X<int> then the program is ill-formed. Unless I'm misreading it, the sentence quoted above would create a point of instantiation for X<int>. I'm not sure whether this is deliberate, but I propose a substantive change to

(B) If within a generated class definition, a context requires the instantiation of another template then the point of instantiation for the second template is ....

This is still a little vague, as it isn't clear exactly what contexts within a definition might require instantiation. In particular my understanding is that the bodies of inclass definitions are not instantiated at this point. I propose to clarify this with

(C) The contexts that might require instantiation of another template are:
    a) the base-clause of the template declaration.
    b) declarations of non-static data members.

c) any expressions occurring outside the function-body and ctor-initializer of function definitions, and outside the declarations of members classes

(c) includes default arguments, constant expressions in arrays and non-type template arguments. I think that is an exhaustive list, but I've phrased (c) in a roundabout fashion so as to cover myself if I've forgotten anything.

I have deliberately excluded the bodies of member function definitions (and ctor-initializers of constructors.) My understanding of the current WP is that member functions have their own point of instantiation and so any requirements imposed by their bodies are not automatically propagated by the instantiation of the class. Specifically p6 says "An implementation shall not instantiate a function, nonvirtual member function, class or member class that does not require instantiation.

For the same reason I have also excluded declarations of nested (member) classes.

I also propose to add a new paragraph.

> (D) At the point of instantiation of a template, the type of all nonstatic data members and of all base classes shall be complete object types.

This is required because the proposed words in 3.1A and 3.1B do not cover these members. It isn't clear to me whether the words in 10 would cover the base class case, but it doesn't hurt to be explicit here.

Note that this proposal has the consequence.

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

X<char> xc ; // ill-formed, A is not complete
class A { } ;
X<int> xi ;  // ok, it is completed later.
```

A question remains in my mind about static data members. The WP seems to contemplate their definition being instantiated, (in a fashion similar to template functions), and at that point they will need to have a complete type, but it isn't clear to me where to say that.