X3J16/96-0103R1
WG21/N0921R1

Dependent Names in Templates

Author: Erwin Unruh
Email: erwin.unruh@mch.sni.de
Siemens Nixdorf information systems

Revision 1: I committed to specify the rules for dependent names at the Stockholm metting. Unfortunately I was not able to specify those changes in a quality I would like them to be. I know the WP changes are not complete and need some more work by the editor. I think the intent is clear.

Based on a new approach for the template compilation model, a change in the rules rgarding dependent names was made. I used this situation to think about the rules and tried to complete them. I send those rules to the reflector.

Unfortunately I was not able to refine the rules in time for the mailing, so I just put the refelctor message in the mailing. If I have time I may rework this paper so we have a better version for the meeting.

Reflector message ext-3611:

> 2. Dependent Names
>
> We propose a simplification of the definition of "dependent name."
> The principal goal is to make this concept more strictly syntactic,
> as suggested in Sean Corfield’s editorial box (Box 28).

I welcome this approach (independent from the rest of the proposal). But I think the rules given are not accurate enough. I will give a more precise set of rules.

When working on this, I made a subtle change in my perception. The usual view is that a type is dependent. I gave up that idea and say that a "type-id" is dependent. It may be questionable whether T in

```
template < typename T > void foo(T);

foo(1);
```

is a type or not. I think it is a placeholder for a type. So the question whether T is int is meaningless for the definition per se. It becomes a meaning when processing an instantiation.
On the same grounds I defined the attribute of being dependent only for expressions, not for their type. The result, of whether a certain piece of code is dependent is not changed.

When drafting the following rules I made a few substantive changes:
- I considered type, value and template parameters.
- The original wording does allow a name to be dependent in one instantiation and not be dependent in another. I follow the rule that a name is dependent in all or no instantiation. So the attribute can be determined looking at the definition alone.
  This makes some programs ill-formed. They are:
  - where the template parameter is a type already used in the template

```
void f(int);
template<class T> void foo(T t){
  f(1.0);       // not dependent
};
void f(double);
foo(1.0);
```

  - where a conversion to a template parameter is used to call a function.

```
class A { };
class B { operator A(); };
template <class T> void foo(T t) {
  B b;
  f(b);
}
void f(A);
foo(A);
```

- I tried to get a somehow minimal set of dependent names. So as example

```
f(sizeof(T))
```

  is not dependent, because the argument type is int.
- I introduced 3 targets of "dependent": a type-id being (type-)dependent, when the represented type depends; an expression being type-dependent, when its type depends; and a constant-expression being value-dependent, when its value depends.

I am using the term "type-id" to describe a syntactic construct describing a type. It is not identical to the syntactic term type-id. The editor is requested to use a more appropiate term for this term.

The rules are :
****

Inside a template some constructs have semantics which are different in differen
t
instantiations. We say that such a construct "depends" on the template parameter
s.
The following constructs can depend:
- a qualified-id which can denote different entities
- a type-id which can denote different tyes.
- an expression which can have a different type.
- a constant-expression which can have different values
- an unqualified name whose binding can change

The exact rules are as follows, where T stands for a type-id representing a type, E stands for an expression, P stands for a parameter of the template, x stands for an identifier and TM stands for a template:

The rules for type-id also cover class-id used for scoping.
A type-id depends on a template parameter P if it is of the form

```
cv T            and T depends on P
T* cv opt       and T depends on P
T&              and T depends on P
T1 T2::* cv opt and T1 or T2 depends on P
T[E]            and T depends on P or E value-depends on P
T (T1, .. Tn) cv opt throw( .. ) opt
                and T or one of T1 .. Tn depends on P

                                Note: exception specification does not give
                                dependency
                                End Note.

TM<P1, .. Pn>   and P is TM or one of P1 to Pn depends on P;

T1 :: T2        and T1 or T2 depends on P

T               and T is P or T is a typedef declared with a type-id which
```

depends on P

A template template argument depends on P if it is either P or of the form
T::TM where T depends on P.
A non-integral non-type template argument depends on P if it is of the form
T::x or &T::x where T depends on P, or when the argument is P
A type template argument depends on P if it (as a type-id) depends on P
An integral non-type template argument depends on P if the argument (as
a constant expression) value-depends on P

An expression type-depends on a template parameter P if it is of the form

this and the class type of the member function depends on P
T::x and T depends on P
x and x is declared with a type-id which depends on P
operator T and T depends on P
E1[E2] and E1 or E2 depends on P
E(E1, .. En) and E or one of E1 .. En depends on P
T(E1, .. En) and T depends on P
(T)E and T depends on P
static_cast<T>(E) and T depends on P
const_cast<T>(E) and T depends on P
reinterpret_cast<T>(E) and T depends on P
dynamic_cast<T>(E) and T depends on P
new T (E1, .. En) and T depends on P
new (E1, .. En) T (EE1, .. EEn)
and T depends on P
Note:
Whether a cast is dependent depends solely on the type being cast to,
not whether the expression being cast depends. The same is true for new-
expressions, where the resulting type does not depend on placement
expressions or the arguments to the constructor.
End Note
E.x
E->x and E depends on P
E. template opt T::x
E-> template opt T::x and E depends on P or T depends on P
E++ and E depends on T
E-- and E depends on T
op E and E depends on P and op is one of + -, (tilde), !, *, &,
++, --
E1 op E2 and E1 or E2 depends on P and op is one of +, -, *, /, %,
^, &, | =, <, > +=, -=, *=, /=, %=, ^=, &=, |=, <<, >>, >>=,
<<=, ==, !=, <=, >=, &&, ||, (comma), ->*
E ? E1 : E2 and E1 or E2 depends on P
(E) and E depends on P

Note:
The following forms of expressions never type-depend on a P:
E. pseudo-destructor-call because its result is void
E-> pseudo-destructor-call because its result is void
literal because its type is fixed
typeid(T)
typeid(E) because its tye is typeinfo&
delete E
delete [] E because its type is void
End Note.

An constant-expression value-depends on a template parameter P if it is of
the form

T::x and T depends on P
x and x is P (where P is a value-parameter)
x and x is declared with a type-id which depends on P
x and x is an integral constant initialized with an expression
which value-depends on P

```
T(E)
(T)E
static_cast<T>(E)   and T depends on P or E value-depends on P
const_cast<T>(E)   and T depends on P or E value-depends on P
reinterpret_cast<T>(E)   and T depends on P or E value-depends on P
dynamic_cast<T>(E)   and T depends on P or E value-depends on P
sizeof (T)      and T depends on P
sizeof E        and E type-depends on P
op E            and E value-depends on P
E1 op E2        and E1 or E2 value-depends on P
E ? E1 : E2     and E or E1 or E2 value-depends on P

Note:
An expression of the form
        offsetof(T,x)
depends on P if T or x depend on P. However, offsetof is not part of the
expression syntax and so special rule for it does not exist.
End Note.
Note:
The rules specify that "sizeof(T) ? 2 : 2" is considered value-dependend,
even if the value will always be 2.
End Note.
************************************************************

still open: when is a non-integral template-value-argument dependent?

OPEN: The above rules do not cover implicit scoping, as are classes,
functions and data member of a dependent template class. See

        template<class T> class C {
                class D{};
                void f();
                int i;
                void foo(){
                        C c;    // dependent ?
                        i++;    // dependent ?
                        f();    // dependent ?
                }
        };
(After a second thought I do doubt whether these should be considered
dependent. I am not sure!)


When scanning a template definition, lookup all names present. If a
non-qualified name appears in the position

        name ( E1, .. En )

within an expression
and the lookup resolves to a set which only contains functions
        (the set may be empty)
and one of the expressions E1 to En type-depends on a parameter P of
        the template
then the name is looked up again in the context of the instantiation.


I leave open the semantics of the second lookup and the relationship of
first and second lookup. They may follow the WP rules or some of the
newly proposed rules.


The list for value-dependency  is shorter than for type-dependency because
a whole set of constructs are not allowed in constant expressions. The
intent is that if an expression type-depends on P and is a valid constant
expression, than it also value-depends on P.
```

The rules clearly favour the first lookup. If the first lookup finds a type or a variable, a second lookup is not tried. This does even hold if that variable does not have an operator().

They have the big advantage (especially for compiler vendors) that the set of names which may be dependent is clearly defined at the point of the template definition.

I know that the rules above need work to be formed into standardeese. I also think that there are a few mistakes in the rules.

Andy: (Or whoever does the editing): Check, whether the using of "depend on template parameter" is consistent and where argument is used instead of parameter.

Working paper changes:

in 14.6 [temp.res]

- delete the first sentence of paragraph 2 (two and a half line)
- add a reference to 14.6.2 [temp.dep] after the word "depends" in the former second sentence of paragraph 2.
- remove Box 20.
- replace paragraphs 7 and 8 by

When looking for the declaration of a name used in a template definition, the usual lookup rules (_??_) are applied.
A qualified name which depends on a template parameter is not looked up within the template definition.

If one operand of an operator type-depends on a template parameter, the lookup is restricted to the operators found during "scoped lookup" and lookup in the namespace of the class of a non-dependend operand. An additional lookup will be done during instantiation (see [temp.???])

If an id-expression is used as the function (???) and the lookup resolves to a (possibly empty) set which only contains functions, and one of the arguments type-depends on a template parameter, an additional lookup will be done during instantiation (see [temp.???])

in 14.6.2 [temp.dep]

- put the text between the two starred lines at the beginning of that subsection
- remove paragraphs 2 through 6 (keeping the examples in paragraphs 2 and 4 may be a good idea)
- remove paragraph 1 (its semantics are now covered by 14.6.4 as introduced by John Wilkinsons wording in X3J16/96-0155 = WG21/N0973)