

Dependent scope and nested templates

Asked 10 years, 10 months ago   Modified 10 years, 10 months ago   Viewed 15k times

▲ When I compile this:

```
15 #ifndef BTREE_H
    #define BTREE_H
    #include <QList>

    template <class T, int degree>
    class btree
    {
    public:
        class node
        {
        public :
            node();
        private:
            node* parent;
            QList<T> values;
            QList<node*> children;
        };
    public:
        btree();
        void insert(const T& value);
        node* findLeaf(const T& value);
        void performInsertion(const T& value, node& place);
        //
        node* root;
    };
#endif // BTREE_H
```

Implementation of findLeaf is this:

```
template <class T, int degree>
btree<T,degree>::node* btree<T,degree>::findLeaf(const T &value)
{
    if(root == NULL)
        return root;
}
```


This error occurs:

error: need 'typename' before 'btree<T, degree>::Node' because 'btree<T, degree>' is a dependent scope

C++ templates nested-class

Share Edit Follow Flag

asked Jul 4, 2011 at 12:20

 sorush-t  
9,886 ● 16 ● 82 ● 168

▲ For the unwary: the error probably occurs at the definition of `findLeaf`. – [Matthieu M.](#) Jul 4, 2011 at 12:22

▲ I'm trying to paste my code but so doesn't show that! – [sorusht-r](#) Jul 4, 2011 at 12:23

5 ▲ I wish SO would put line numbers next to the code, to make references easier. – [Matthieu M.](#) Jul 4, 2011 at 12:25

2 Answers

Sorted by: Highest score (default) ▾

▲ No, this has not to do with C++'s grammar, but with lazy instantiation of C++ templates and two phase lookup.

43 In C++, a dependant name is a name or symbol whose meaning *depends* on one or more template parameters:

```
template <typename T>
struct Foo {
    Foo () {
        const int x = 42;
        T::Frob (x);
    }
};
```

By parsing that snippet alone, without knowin all future values of T, no C++ compiler can deduce whether `Frob` in `T` is a function name, a type name, something else, or whether it exists at all.

To give an example for why this is relevant, imagine some types you will substitute for T:

```
struct Vietnam {
    typedef bool Frob; // Frob is the name of a type alias
};

struct Football {
    void Frob (int) {} // Frob is a function name
};

struct Bigfoid {}; // no Frob at all!
```

Put those into our Foo-template:

```
int main () {
    Foo<Vietnam> fv; // Foo::Foo would declare a type
    Foo<Football> ff; // Foo::Foo would make a function call
    Foo<Bigfoid> ffw; // Foo::Foo is not defined at all
}
```

Relevant in this is the concept of **two-phase lookup**. In the first phase, non-dependent code is parsed and compiled:

```
template <typename T>
struct Foo {
    Foo () {
        const int x = 42; // does not depend on T
        T::Frob (x); // full check skipped for second phase, only rudimentary
        // checking
    }
};
```

This first phase is what lets compilers emit error messages in the template definition itself.

The second phase would trigger errors of your template in conjunction with the then-known type T.

Some early C++ compilers would only parse templates once you instantiate them; with those compilers, disambiguation wasn't needed, because at the point of instantiation, the template arguments are known. The problem with this one-phase lookup is that many errors in the template *itself* won't be detected at all or only late in the compile, because templates are by default instantiated lazily, i.e. only parts of a class-template are expanded that are actually used, plus it gives you more cryptic error messages that possibly root in the template-argument.

So in order for two-phase lookup to work, you must help the compiler. In this case, you must use `typename` in order to tell the compiler that you mean a type:


```
template <typename T>
struct Foo {
    Foo () {
        const int x = 42;
        typename T::Frob (x);
    }
};
```

The compiler now knows that x is a variable of type Frob.)

Share Edit Follow Flag

edited Jul 4, 2011 at 13:20

answered Jul 4, 2011 at 12:58

 Sebastian Mach  
37.4k ● 6 ● 88 ● 128

▲ Great answer. Thank you very much. – [Drew Noakes](#) Jan 22, 2013 at 23:10

▲ Now I understand what is the rationale behind the error message, thank you. – [Rasoul](#) Oct 15, 2013 at 12:52

▲ Great explanation! Hope all answers on SO are like this answer! – [Anh Tuan](#) May 13, 2015 at 2:10

▲ I wonder why this isn't the answer selected by the user who asked the question. – [Mostafa Talebi](#) Oct 13, 2016 at 23:36 ✓

▲ The C++ grammar is horrendous, and as such it is not possible, when given a template class, to know whether the `node` you refer to is a variable/constant or a type.

26 The Standard therefore mandates that you use `typename` before types to remove this ambiguity, and treats all other usages as if it was a variable.

✓ Thus

```
template <typename T, int degree>
typename btree<T,degree>::node* btree<T,degree>::findLeaf(T const& value)
```

is the correct signature for the definition.

Share Edit Follow Flag

answered Jul 4, 2011 at 12:25

 Matthieu M.  
267k ● 42 ● 407 ● 676

▲ Now compiler can't find my function: 'no 'btree<T, degree>::node' 'btree<T, degree>::findLeaf(const T&)' member function declared in class 'btree<T, degree>' – [sorusht-r](#) Jul 4, 2011 at 12:28

▲ Sorry, this works correctly. I typed a character wrong... Thank you – [sorusht-r](#) Jul 4, 2011 at 12:33

▲ @phresnel: I find this keyword redundant in many cases, the very fact that the compiler helpfully provide it in its error message is a red herring. It does not make sense for a function signature to contain a value, whether in the return type or as an argument type. And yet you are forced to write `typename`. I admit that there may be convoluted situations (due to the horrendous grammar) where `typename` would be necessary, but most of the time, it's not. Example of "horrendousness": `template <typename T, size_t N> T* begin(T (&array)[N]) { return array; }` (to avoid the pointers to function) – [Matthieu M.](#) Jul 4, 2011 at 14:25

▲ ... This is barely understandable, and will in fact trump most readers. Why are not array declared as `T[N]& array`? Or pointer to functions as `typedef void (*f)(); func_ptr; f No`, instead we have a mishmash inherited from C, and throwing templates in the mix make parsing extremely difficult even for compilers, so us poor humans... – [Matthieu M.](#) Jul 4, 2011 at 14:27

▲ @Matthieu M.: I do not disagree that the grammar is poor in several aspects. However, the typename thing is not rooted in that. In C++, function signatures can always define and override default arguments in the parameter list. In metaprogramming, function signatures can be and are often based on values, e.g. `enable_if<is_foo<T>::value>::bar()`. Ignoring some other aspects, I think it would be more confusing than necessary to only require typename-disambiguation within template-argument-lists, but not on the outside. ... – [Sebastian Mach](#) Jul 4, 2011 at 14:44

▲ ... My personal opinion about this, anyways: Writing library code, or generally, code to be used by other programmers, has completely different requirements than writing application code. Writing libraries is often not ought to be easy, using them is. And one might expect a different degree of experience from an elder programmer: authoring library code is what templates are for; therefore, templates are typically not for beginners (for my part, when I am in "library writing mode", I rarely forget when a typename is needed, and when not) – [Sebastian Mach](#) Jul 4, 2011 at 14:49 ✓

▲ @phresnel: I agree that consistency is good... however since it's only required when moving the definition outside the class (and not inside...), it really makes moving definition outside the class really difficult (that and the double template for template methods inside template classes). And as a consequence, people often just define all in the class, because it's easier, and even though it means groking the class afterward will be more difficult. – [Matthieu M.](#) Jul 4, 2011 at 14:50

▲ @phresnel: I share your opinion about library vs applicative code, but still would prefer C++ to be easier for novices :) – [Matthieu M.](#) Jul 4, 2011 at 14:52

▲ It is sort of related to grammar. If the grammar required, say `node` when accessing a member type, instead of `node`, then the `typename` would be implicit in that. – [Claudio](#) Sep 9, 2014 at 17:43