Why do I have to access template base class members through the this pointer?

Asked 13 years, 11 months ago Modified 5 years, 2 months ago Viewed 34k times

269

If the classes below were not templates I could simply have `x` in the `derived` class. However, with the code below, I *have to* use `this->x`. Why?

```
template <typename T>
class base {

protected:
    int x;
};

template <typename T>
class derived : public base<T> {

public:
    int f() { return this->x; }
};

int main() {
    derived<int> d;
    d.f();
    return 0;
}
```

c++ templates inheritance c++-faq

Share
Improve this question
Follow

edited Jun 15, 2016 at 14:13
Destructor
14.4k ● 11 ● 68 ● 130

asked Jan 10, 2011 at 1:42
Ali
58.3k ● 31 ● 173 ● 272

@Ed Swangren: Sorry, I missed it among the offered answers when posting this question. I had been looking for the answer for a long time before that. – Ali Jan 10, 2011 at 1:58

8 This happens because of the two-phase name lookup (which not all compilers use by default) and dependent names. There are 3 solutions to this problem, other than prefixing the `x` with `this->`, namely : **1)** Use the prefix `base<T>::x`, **2)** Add a statement `using base<T>::x`, **3)** Use a global compiler switch that enables the permissive mode. The pros & cons of these solutions are described in stackoverflow.com/questions/50321788/…
– George Robinson May 14, 2018 at 13:53

3 Answers

Sorted by: Highest score (default)

363

Short answer: in order to make `x` a dependent name, so that lookup is deferred until the template parameter is known.

Long answer: when a compiler sees a template, it is supposed to perform certain checks immediately, without seeing the template parameter. Others are deferred until the parameter is known. It's called two-phase compilation, and MSVC doesn't do it but it's required by the standard and implemented by the other major compilers. If you like, the compiler must compile the template as soon as it sees it (to some kind of internal parse tree representation), and defer compiling the instantiation until later.

The checks that are performed on the template itself, rather than on particular instantiations of it, require that the compiler be able to resolve the grammar of the code in the template.

In C++ (and C), in order to resolve the grammar of code, you sometimes need to know whether something is a type or not. For example:

```
#if WANT_POINTER
    typedef int A;
#else
    int A;
#endif
static const int x = 2;
template <typename T> void foo() { A *x = 0; }
```

if A is a type, that declares a pointer (with no effect other than to shadow the global `x`). If A is an object, that's multiplication (and barring some operator overloading it's illegal, assigning to an rvalue). If it is wrong, this error must be diagnosed *in phase 1*, it's defined by the standard to be an error *in the template*, not in some particular instantiation of it. Even if the template is never instantiated, if A is an `int` then the above code is ill-formed and must be diagnosed, just as it would be if `foo` wasn't a template at all, but a plain function.

Now, the standard says that names which *aren't* dependent on template parameters must be resolvable in phase 1. `A` here is not a dependent name, it refers to the same thing regardless of type `T`. So it needs to be defined before the template is defined in order to be found and checked in phase 1.

`T::A` would be a name that depends on T. We can't possibly know in phase 1 whether that's a type or not. The type which will eventually be used as `T` in an instantiation quite likely isn't even defined yet, and even if it was we don't know which type(s) will be used as our template parameter. But we have to resolve the grammar in order to do our precious phase 1 checks for ill-formed templates. So the standard has a rule for dependent names - the compiler must assume that they're non-types, unless qualified with `typename` to specify that they *are* types, or used in certain unambiguous contexts. For example in `template <typename T> struct Foo : T::A {};`, `T::A` is used as a base class and hence is unambiguously a type. If `Foo` is instantiated with some type that has a data member `A` instead of a nested type A, that's an error in the code doing the instantiation (phase 2), not an error in the template (phase 1).

But what about a class template with a dependent base class?

```
template <typename T>
struct Foo : Bar<T> {
    Foo() { A *x = 0; }
};
```

Is A a dependent name or not? With base classes, *any* name could appear in the base class. So we could say that A is a dependent name, and treat it as a non-type. This would have the undesirable effect that *every name* in Foo is dependent, and hence *every type* used in Foo (except built-in types) has to be qualified. Inside of Foo, you'd have to write:

```
typename std::string s = "hello, world";
```

because `std::string` would be a dependent name, and hence assumed to be a non-type unless specified otherwise. Ouch!

A second problem with allowing your preferred code (`return x;`) is that even if `Bar` is defined before `Foo`, and `x` isn't a member in that definition, someone could later define a specialization of `Bar` for some type `Baz`, such that `Bar<Baz>` does have a data member `x`, and then instantiate `Foo<Baz>`. So in that instantiation, your template would return the data member instead of returning the global `x`. Or conversely if the base template definition of `Bar` had `x`, they could define a specialization without it, and your template would look for a global `x` to return in `Foo<Baz>`. I think this was judged to be just as surprising and distressing as the problem you have, but it's *silently* surprising, as opposed to throwing a surprising error.

To avoid these problems, the standard in effect says that dependent base classes of class templates just aren't considered for search unless explicitly requested. This stops everything from being dependent just because it could be found in a dependent base. It also has the undesirable effect that you're seeing - you have to qualify stuff from the base class or it's not found. There are three common ways to make `A` dependent:

- `using Bar<T>::A;` in the class - `A` now refers to something in `Bar<T>`, hence dependent.
- `Bar<T>::A *x = 0;` at point of use - Again, `A` is definitely in `Bar<T>`. This is multiplication since `typename` wasn't used, so possibly a bad example, but we'll have to wait until instantiation to find out whether `operator*(Bar<T>::A, x)` returns an rvalue. Who knows, maybe it does...
- `this->A;` at point of use - `A` is a member, so if it's not in `Foo`, it must be in the base class, again the standard says this makes it dependent.

Two-phase compilation is fiddly and difficult, and introduces some surprising requirements for extra verbiage in your code. But rather like democracy it's probably the worst possible way of doing things, apart from all the others.

You could reasonably argue that in your example, `return x;` doesn't make sense if `x` is a nested type in the base class, so the language should (a) say that it's a dependent name and (2) treat it as a non-type, and your code would work without `this->`. To an extent you're the victim of collateral damage from the solution to a problem that doesn't apply in your case, but there's still the issue of your base class potentially introducing names under you that shadow globals, or not having names you thought they had, and a global being found instead.

You could also possibly argue that the default should be the opposite for dependent names (assume type unless somehow specified to be an object), or that the default should be more context sensitive (in `std::string s = "";`, `std::string` could be read as a type since nothing else makes grammatical sense, even though `std::string *s = 0;` is ambiguous). Again, I don't know quite how the rules were agreed. My guess is that the number of pages of text that would be required, mitigated against creating a lot of specific rules for which contexts take a type and which a non-type.

Share

Improve this answer

Follow

edited Oct 4, 2019 at 13:18

community wiki
13 revs, 2 users 99%
Steve Jessop

2 Ooh, nice detailed answer. Clarified a couple of things I've never bothered to look up. :) +1 – Stack Overflow is garbage Jan 10, 2011 at 3:41

30 @jalf: is there such a thing as the C++QTWBFAETYNSYEWTKTAAHMITTBGOW - "Questions that would be frequently asked except that you're not sure you even want to know the answer and have more important things to be getting on with"? – Steve Jessop Jan 10, 2011 at 3:51

7 extraordinary answer, wonder if the question could fit in the faq. – Matthieu M. Jan 10, 2011 at 8:19

1 Whoa, can we say encyclopaedic? *highfive* One subtle point, though: "If Foo is instantiated with some type that has a data member A instead of a nested type A, that's an error in the code doing the instantiation (phase 2), not an error in the template (phase 1)." Might be better to say that the template isn't malformed, but this could still be a case of an incorrect assumption or logic bug on the part of the template writer. If the flagged instantiation was actually the intended usecase, then template would be wrong. – Ionoclast Brigham Jul 12, 2013 at 19:53

1 @SteveJessop the base class is only not searched for unqualified name lookup. Said another way, unqualified name lookup, whether for dependent names or not, will never look into dependent base classes. – Johannes Schaub - litb Sep 15, 2013 at 13:42

18

(Original answer from Jan 10, 2011)

I think I have found the answer: GCC issue: using a member of a base class that depends on a template argument. The answer is not specific to gcc.

Update: In response to mmichael's comment, from the draft N3337 of the C++11 Standard:

> **14.6.2 Dependent names [temp.dep]**
> [...]
> 3 In the definition of a class or class template, if a base class depends on a template-parameter, the base class scope is not examined during unqualified name lookup either at the point of definition of the class template or member or during an instantiation of the class template or member.

Whether *"because the standard says so"* counts as an answer, I don't know. We can now ask why the standard mandates that but as Steve Jessop's excellent answer and others point out, the answer to this latter question is rather long and arguable. Unfortunately, when it comes to the C++ Standard, it is often nearly impossible to give a short and self-contained explanation as to why the standard mandates something; this applies to the latter question as well.

Share
Improve this answer
Follow

edited May 23, 2017 at 11:54
Community Bot
1 ● 1

answered Jan 10, 2011 at 1:48
Ali
58.3k ● 31 ● 173 ● 272

16

The `x` is hidden during the inheritance. You can unhide via:

```
template <typename T>
class derived : public base<T> {
```

```
public:
    using base<T>::x;             // added "using" statement
    int f() { return x; }
};
```

Share Improve this answer Follow

answered Jan 10, 2011 at 1:47

chrisaycock
37.8k ● 15 ● 92 ● 127

31 This answer doesn't explain *why* it's hidden. – jamesdlin Jan 10, 2011 at 2:49

I get `base<T> is not a namespace or unscoped enum` – JDługosz Sep 29, 2020 at 15:31