How do you understand dependent names in C++

Asked 12 years, 5 months ago Active 2 years, 3 months ago Viewed 8k times

22

14

I come across this term "dependent names" typically in the context of templates. However, I rarely touch the latter. Thus naturally would like to know more about the concept of dependent names.

How do you understand it in the context of templates and outside of them? example are critically encouraged!

c++

Share Edit Follow Flag

asked Oct 6, 2009 at 20:12
vehomzzz
39.8k 71 175 213

see also the answer at stackoverflow.com/a/613132/1558890 – Jim Garrison Jun 19, 2013 at 1:17

2 Answers

Active Oldest Votes

19

Dependent names are characterized by a **dependency** on a template argument. Trivial example:

```
#include <vector>

void NonDependent()
{
  //You can access the member size_type directly.
  //This is precisely specified as a vector of ints.

  typedef std::vector<int> IntVector;
  IntVector::size_type i;

  /* ... */
}

template <class T>
void Dependent()
{

  //Now the vector depends on the type T.
  //Need to use typename to access a dependent name.

  typedef std::vector<T> SomeVector;
  typename SomeVector::size_type i;

  /* ... */
}

int main()
{
  NonDependent();
  Dependent<int>();
  return 0;
}
```

EDIT: As I mentioned in the comment below, this is an example of a peculiar situation regarding the use of dependent names which appears quite frequently. Sometimes the rules governing the use of dependent names are not what one might instinctively expect.

For instance, if you have a dependent class which derives from a depenent base, but within an scope in which a name from the base class apparently doesn't depent on the template, you might get a compiler error just like below.

```
#include <iostream>

template <class T>
class Dependent
{
protected:
  T data;
};

template <class T>
class OtherDependent : public Dependent<T>
{
public:
  void printT()const
  {
    std::cout << "T: " << data << std::endl; //ERROR
  }
};

int main()
{
  OtherDependent<int> o;
  o.printT();
  return 0;
}
```

This error happens because the compiler will not lookup name `data` inside the base class template since it doesn't dependent on `T` and, consequently, it is not a depedent name. The ways to fix are using `this` or explicitly telling the dependent base class template:

```
std::cout << "T: " << this->data << std::endl; //Ok now.
std::cout << "T: " << Dependent<T>::data << std::endl; //Ok now.
```

or placing `using` declaration:

```
template <class T>
class OtherDependent : public Dependent<T>
{
    using Dependent<T>::data; //Ok now.
    ....
};
```

Share Edit Follow Flag

edited Nov 17, 2019 at 18:49
4LegsDrivenCat
866 1 12 22

answered Oct 6, 2009 at 20:44
Leandro T. C. Melo
3,874 20 22

You should make explicit that 'typename' is required because the dependent name is a type and not a static variable. With dependent names that are not types, 'typename' would not be required. – David Rodríguez - dribeas Oct 6, 2009 at 21:42

Yes, that's one example of a dependent name. Actually, one I see many beginers confused with. I also edited the post with another situation concerning dependent names that I think happen quite often. – Leandro T. C. Melo Oct 7, 2009 at 11:37

+1: I never realised that the second example was caused by the name being classified as not dependent. – Troubadour Oct 8, 2009 at 7:04

1 Placing `using Dependent<T>::data` on `OtherDependent` definition will do the work too. – PaperBirdMaster Feb 27, 2013 at 11:59

2 @LeandroT.C.Melo shouldn't `name` in "compiler will not lookup name inside" be `data`? – Koldar Sep 6, 2019 at 12:07

13

A dependent name is essentially a name that depends on a template argument.

When using templates there is a distinction between the point of definition of the template and the point of instantiation i.e. where you actually use the template. Names that depend on a template don't get bound until the point of instantiation whereas names that don't get bound at the point of definition.

A simple example would be:

```
template< class T > int addInt( T x )
{
    return i + x.toInt();
}
```

where a declaration or definition of `i` would need to appear *before* the definition given above since `i` does not depend on the template argument `T` and is therefore bound at the point of definition. The definition of the `toInt` member of the as-yet-unknown-type `x` variable only has to appear before the `addInt` function is actually used somewhere as it is a dependent name (technically the point of instantiation is taken as the nearest enclosing global or namespace scope just before the point of use and so it has to be available before that).

Share Edit Follow Flag

edited Oct 6, 2009 at 22:02

answered Oct 6, 2009 at 20:23
Troubadour
13k 2 35 55

I'll just point out that a lot of compilers seem not to enforce the dependent names rule, but that once you break the rule, there can be big variations in what you end up needing to do to make things work. I've had *BIG* issues because of this in the past - bordering on throw-it-out-and-start-again big. Annoying as I *thought* I'd written portable code - and the code concerned had been used in Borland C++ 5 then two (maybe three) versions of MS VC++ before I hit a problem. Basically, beware. – user180247 Oct 6, 2009 at 21:26

I don't think that the 'technically' comment at the end is 100% correct but rather misleading. 'Technically', toInt is a member function of the type of x (argument to addInt), so it must be declared inside that class, that might or not be in the 'nearest global or namespace scope from the point of use': `namespace A { struct B { int toInt(); }; } void f() { A::B b; addInt( b ); }` <- the enclosing namespace at the place of call is the global namespace but the code is correct as th type of the 'b' variable was qualified. – David Rodríguez - dribeas Oct 6, 2009 at 21:40

@dribeas: You're quite correct. I wrote something different to what I was thinking! I've edited the answer to clarify that the enclosing namespace is relevant to the meaning of "point of instantiation" i.e. it happens a bit earlier that the actual point of use. Thanks. – Troubadour Oct 6, 2009 at 22:04

I would still remove the namespace part. The dependent name must be available before the use of the template. In your particular example, with the dependent name being a member function namespaces don't affect, the class can be defined in any other namespace (if the variable definition is qualified). Even if instead of a member you have a free function, ADL would kick in and it will search within the closest namespace to the type of x, not the point of instantiation of the template. – David Rodríguez - dribeas Oct 7, 2009 at 11:14

@dribeas: I don't quite follow now. The namespace part is simply there to point out the subtle difference between the point of instantiation and the point of use i.e if I call `addInt` within a member of some other class then the point of instantiation of `addInt` is not in that class's member at the point of use where you might naturally expect it to be. This is to prevent picking up a binding for dependent names from within that member. Had `toInt` been an ordinary function then any locally defined `toInt` function (i.e. defined within that member) would be ignored. – Troubadour Oct 8, 2009 at 7:02