Where and why do I have to put the "template" and "typename" keywords?

Asked 13 years ago Active 9 months ago Viewed 217k times

1274

884

In templates, where and why do I have to put `typename` and `template` on dependent names?
What exactly are dependent names anyway?

I have the following code:

```
template <typename T, typename Tail> // Tail will be a UnionNode too.
struct UnionNode : public Tail {
    // ...
    template<typename U> struct inUnion {
        // Q: where to add typename/template here?
        typedef Tail::inUnion<U> dummy;
    };
    template< > struct inUnion<T> {
    };
};
template <typename T> // For the last node Tn.
struct UnionNode<T, void> {
    // ...
    template<typename U> struct inUnion {
        char fail[ -2 + (sizeof(U)%2) ]; // Cannot be instantiated for any U
    };
    template< > struct inUnion<T> {
    };
};
```

The problem I have is in the `typedef Tail::inUnion<U> dummy` line. I'm fairly certain that `inUnion` is a dependent name, and VC++ is quite right in choking on it.
I also know that I should be able to add `template` somewhere to tell the compiler that inUnion is a template-id. But where exactly? And should it then assume that inUnion is a class template, i.e. `inUnion<U>` names a type and not a function?

c++ templates typename c++-faq dependent-name

Share Edit Follow Flag

edited Apr 16, 2020 at 8:06
Arsen Khachaturyan
7,195 ●4 ●36 ●38

asked Mar 4, 2009 at 11:56
MSalters
166k ●8 ●148 ●332

67 Political sensitivities, portability. – MSalters Mar 5, 2009 at 10:34

6 I made your actual question ("Where to put template/typename?") stand out better by putting the final question and code at the beginning and shortened the code horizontally to fit a 1024x screen. – Johannes Schaub - litb Dec 22, 2010 at 10:10

Note that this using typename SomeBase<T>::type; does not work in GCC due to gcc.gnu.org/bugzilla/show_bug.cgi?id=14258. – Justin L. Aug 26, 2011 at 18:58

9 Removed the "dependent names" from the title because it appears that most people who wonder about "typename" and "template" don't know what "dependent names" are. It should be less confusing to them this way. – Johannes Schaub - litb Oct 29, 2011 at 13:53

This helped me with typename: pages.cs.wisc.edu/~driscoll/typename.html – Ciro Santilli 新疆再教育营六四事件法轮功郝海东 Sep 21, 2013 at 18:33

2 @MSalters : boost is quite portable. I'd say only politics is the general reason why boost is often unembraced. The only good reason I know is the increased build times. Otherwise this is all about losing thousands of dollars reinventing the wheel. – v.oddou Nov 28, 2013 at 0:27

@v.oddou: The comment is 4 1/2 years old. – MSalters Nov 28, 2013 at 8:37

@MSalters: boost was not less portable 4 years ago. My comment was not directed at criticizing you. I know this kind of decisions comes generally from external constraints/bosses/publishers/idiots... so no need to feel on the defensive :) – v.oddou Nov 28, 2013 at 8:55

1 Now it appears to me that `char fail[ -sizeof(U) ]; // Cannot be instantiated for any U` will not work, because the `-sizeof(U)` is still always positive, so it may still work for some or all `U`. – Johannes Schaub - litb Jun 1, 2016 at 11:37

@JohannesSchaub-litb: Good point. It's quite likely to fail, though: `-sizeof(U)` likely is a huge positive number, which probably exceeds the largest possible object size. E.g. on a 32 bits system, `-sizeof(char)` would be 4GB - 1 byte. But it's fixable. – MSalters Jun 1, 2016 at 12:26

8 Answers

Active Oldest Votes

1309

(See here also for my C++11 answer)

In order to parse a C++ program, the compiler needs to know whether certain names are types or not. The following example demonstrates that:

```
t * f;
```

How should this be parsed? For many languages a compiler doesn't need to know the meaning of a name in order to parse and basically know what action a line of code does. In C++, the above however can yield vastly different interpretations depending on what `t` means. If it's a type, then it will be a declaration of a pointer `f`. However if it's not a type, it will be a multiplication. So the C++ Standard says at paragraph (3/7):

> Some names denote types or templates. In general, whenever a name is encountered it is necessary to determine whether that name denotes one of these entities before continuing to parse the program that contains it. The process that determines this is called name lookup.

How will the compiler find out what a name `t::x` refers to, if `t` refers to a template type parameter? `x` could be a static int data member that could be multiplied or could equally well be a nested class or typedef that could yield to a declaration. **If a name has this property - that it can't be looked up until the actual template arguments are known - then it's called a *dependent name* (it "depends" on the template parameters).**

You might recommend to just wait till the user instantiates the template:

> *Let's wait until the user instantiates the template, and then later find out the real meaning of `t::x * f;`.*

This will work and actually is allowed by the Standard as a possible implementation approach. These compilers basically copy the template's text into an internal buffer, and only when an instantiation is needed, they parse the template and possibly detect errors in the definition. But instead of bothering the template's users (poor colleagues!) with errors made by a template's author, other implementations choose to check templates early on and give errors in the definition as soon as possible, before an instantiation even takes place.

So there has to be a way to tell the compiler that certain names are types and that certain names aren't.

The "typename" keyword

The answer is: *We* decide how the compiler should parse this. If `t::x` is a dependent name, then we need to prefix it by `typename` to tell the compiler to parse it in a certain way. The Standard says at (14.6/2):

> A name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword typename.

There are many names for which `typename` is not necessary, because the compiler can, with the applicable name lookup in the template definition, figure out how to parse a construct itself - for example with `T *f;`, when `T` is a type template parameter. But for `t::x * f;` to be a declaration, it must be written as `typename t::x *f;`. If you omit the keyword and the name is taken to be a non-type, but when instantiation finds it denotes a type, the usual error messages are emitted by the compiler. Sometimes, the error consequently is given at definition time:

```
// t::x is taken as non-type, but as an expression the following misses an
// operator between the two names or a semicolon separating them.
t::x f;
```

The syntax allows `typename` only before qualified names - it is therefor taken as granted that unqualified names are always known to refer to types if they do so.

A similar gotcha exists for names that denote templates, as hinted at by the introductory text.

The "template" keyword

Remember the initial quote above and how the Standard requires special handling for templates as well? Let's take the following innocent-looking example:

```
boost::function< int() > f;
```

It might look obvious to a human reader. Not so for the compiler. Imagine the following arbitrary definition of `boost::function` and `f`:

```
namespace boost { int function = 0; }
int main() {
  int f = 0;
  boost::function< int() > f;
}
```

That's actually a valid *expression*! It uses the less-than operator to compare `boost::function` against zero (`int()`), and then uses the greater-than operator to compare the resulting `bool` against `f`. However as you might well know, `boost::function` in real life is a template, so the compiler knows (14.2/3):

> After name lookup (3.4) finds that a name is a template-name, if this name is followed by a <, the < is always taken as the beginning of a template-argument-list and never as a name followed by the less-than operator.

Now we are back to the same problem as with `typename`. What if we can't know yet whether the name is a template when parsing the code? We will need to insert `template` immediately before the template name, as specified by `14.2/4`. This looks like:

```
t::template f<int>(); // call a function template
```

Template names can not only occur after a `::` but also after a `->` or `.` in a class member access. You need to insert the keyword there too:

```
this->template f<int>(); // call a function template
```

Dependencies

For the people that have thick Standardese books on their shelf and that want to know what exactly I was talking about, I'll talk a bit about how this is specified in the Standard.

In template declarations some constructs have different meanings depending on what template arguments you use to instantiate the template: Expressions may have different types or values, variables may have different types or function calls might end up calling different functions. Such constructs are generally said to *depend* on template parameters.

The Standard defines precisely the rules by whether a construct is dependent or not. It separates them into logically different groups: One catches types, another catches expressions. Expressions may depend by their value and/or their type. So we have, with typical examples appended:

- Dependent types (e.g: a type template parameter `T`)
- Value-dependent expressions (e.g: a non-type template parameter `N`)
- Type-dependent expressions (e.g: a cast to a type template parameter `(T)0`)

Most of the rules are intuitive and are built up recursively: For example, a type constructed as `T[N]` is a dependent type if `N` is a value-dependent expression or `T` is a dependent type. The details of this can be read in section `(14.6.2/1)` for dependent types, `(14.6.2.2)` for type-dependent expressions and `(14.6.2.3)` for value-dependent expressions.

Dependent names

The Standard is a bit unclear about what *exactly* is a *dependent name*. On a simple read (you know, the principle of least surprise), all it defines as a *dependent name* is the special case for function names below. But since clearly `T::x` also needs to be looked up in the instantiation context, it also needs to be a dependent name (fortunately, as of mid C++14 the committee has started to look into how to fix this confusing definition).

To avoid this problem, I have resorted to a simple interpretation of the Standard text. Of all the constructs that denote dependent types or expressions, a subset of them represent names. Those names are therefore "dependent names". A name can take different forms - the Standard says:

> A name is a use of an identifier (2.11), operator-function-id (13.5), conversion-function-id (12.3.2), or template-id (14.2) that denotes an entity or label (6.6.4, 6.1)

An identifier is just a plain sequence of characters / digits, while the next two are the `operator +` and `operator type` form. The last form is `template-name <argument list>`. All these are names, and by conventional use in the Standard, a name can also include qualifiers that say what namespace or class a name should be looked up in.

A value dependent expression `1 + N` is not a name, but `N` is. The subset of all dependent constructs that are names is called *dependent name*. Function names, however, may have different meaning in different instantiations of a template, but unfortunately are not caught by this general rule.

Dependent function names

Not primarily a concern of this article, but still worth mentioning: Function names are an exception that are handled separately. An identifier function name is dependent not by itself, but by the type dependent argument expressions used in a call. In the example `f((T)0)`, `f` is a dependent name. In the Standard, this is specified at `(14.6.2/1)`.

Additional notes and examples

In enough cases we need both of `typename` and `template`. Your code should look like the following

```
template <typename T, typename Tail>
struct UnionNode : public Tail {
    // ...
    template<typename U> struct inUnion {
        typedef typename Tail::template inUnion<U> dummy;
    };
    // ...
};
```

The keyword `template` doesn't always have to appear in the last part of a name. It can appear in the middle before a class name that's used as a scope, like in the following example

```
typename t::template iterator<int>::value_type v;
```

In some cases, the keywords are forbidden, as detailed below

- On the name of a dependent base class you are not allowed to write `typename`. It's assumed that the name given is a class type name. This is true for both names in the base-class list and the constructor initializer list:

  ```
  template <typename T>
  struct derive_from_Has_type : /* typename */ SomeBase<T>::type
  { };
  ```

- In using-declarations it's not possible to use `template` after the last `::`, and the C++ committee said not to work on a solution.

  ```
  template <typename T>
  struct derive_from_Has_type : SomeBase<T> {
     using SomeBase<T>::template type; // error
     using typename SomeBase<T>::type; // typename *is* allowed
  };
  ```

Share Edit Follow Flag

edited May 16, 2020 at 1:23
Gabriel Staples
18.9k ●4 ●121 ●154

answered Mar 5, 2009 at 0:27
Johannes Schaub - litb
478k ●121 ●867 ●1187

28 This answer was copied from my earlier FAQ entry which I removed, because I found that I should better use existing similar questions instead of making up new "pseudo questions" just for the purpose of answering them. Thanks go to @Prasoon, who edited the ideas of the last part (cases where typename/template is forbidden) into the answer. – Johannes Schaub - litb Nov 22, 2010 at 18:29

1 Can you help me when should I use this syntax? this->template f<int>(); I get this error 'template' (as a disambiguator) is only allowed within templates but without the template keyword, it works fine. – balki Feb 2, 2011 at 7:51

2 I asked a similar question today, that was soon marked as duplicate: stackoverflow.com/questions/27923722/.... I was instructed to revive this question instead of creating a new one. I must say I don't agree on them being duplicates but who am I, right? So, is there any reason that `typename` is enforced even when the syntax permits no alternative interpretations other than type-names at this point? – JorenHeit Jan 13, 2015 at 18:39

1 @Pablo you are not missing anything. But still required to write the disambiguation even if the complete line would not anymore be ambiguous. – Johannes Schaub - litb Sep 6, 2018 at 21:47

3 @L.F. please create a new C++20 answer, as I did with C++11. – Johannes Schaub - litb Mar 30, 2019 at 9:33

155

C++11

Problem

While the rules in C++03 about when you need `typename` and `template` are largely reasonable, there is one annoying disadvantage of its formulation

```
template<typename T>
struct A {
  typedef int result_type;

  void f() {
    // error, "this" is dependent, "template" keyword needed
    this->g<float>();

    // OK
    g<float>();

    // error, "A<T>" is dependent, "typename" keyword needed
    A<T>::result_type n1;

    // OK
    result_type n2;
  }

  template<typename U>
  void g();
};
```

As can be seen, we need the disambiguation keyword even if the compiler could perfectly figure out itself that `A::result_type` can only be `int` (and is hence a type), and `this->g` can only be the member template `g` declared later (even if `A` is explicitly specialized somewhere, that would not affect the code within that template, so its meaning cannot be affected by a later specialization of `A`!).

Current instantiation

To improve the situation, in C++11 the language tracks when a type refers to the enclosing template. To know that, the type must have been formed by using a certain form of name, which is its own name (in the above, `A`, `A<T>`, `::A<T>`). A type referenced by such a name is known to be the *current instantiation*. There may be multiple types that are all the current instantiation if the type from which the name is formed is a member/nested class (then, `A::NestedClass` and `A` are both current instantiations).

Based on this notion, the language says that `CurrentInstantiation::Foo`, `Foo` and `CurrentInstantiationTyped->Foo` (such as `A *a = this; a->Foo`) are all *member of the current instantiation* **if** they are found to be members of a class that is the current instantiation or one of its non-dependent base classes (by just doing the name lookup immediately).

The keywords `typename` and `template` are now not required anymore if the qualifier is a member of the current instantiation. A keypoint here to remember is that `A<T>` is *still* a type-dependent name (after all `T` is also type dependent). But `A<T>::result_type` is known to be a type - the compiler will "magically" look into this kind of dependent types to figure this out.

```
struct B {
  typedef int result_type;
};

template<typename T>
struct C { }; // could be specialized!

template<typename T>
struct D : B, C<T> {
  void f() {
    // OK, member of current instantiation!
    // A::result_type is not dependent: int
    D::result_type r1;

    // error, not a member of the current instantiation
    D::questionable_type r2;

    // OK for now - relying on C<T> to provide it
    // But not a member of the current instantiation
    typename D::questionable_type r3;
  }
};
```

That's impressive, but can we do better? The language even goes further and *requires* that an implementation again looks up `D::result_type` when instantiating `D::f` (even if it found its meaning already at definition time). When now the lookup result differs or results in ambiguity, the program is ill-formed and a diagnostic must be given. Imagine what happens if we defined `C` like this

```
template<>
struct C<int> {
  typedef bool result_type;
  typedef int questionable_type;
};
```

A compiler is required to catch the error when instantiating `D<int>::f`. So you get the best of the two worlds: "Delayed" lookup protecting you if you could get in trouble with dependent base classes, and also "Immediate" lookup that frees you from `typename` and `template`.

Unknown specializations

In the code of `D`, the name `typename D::questionable_type` is not a member of the current instantiation. Instead the language marks it as a *member of an unknown specialization*. In particular, this is always the case when you are doing `DependentTypeName::Foo` or `DependentTypedName->Foo` and either the dependent type is *not* the current instantiation (in which case the compiler can give up and say "we will look later what `Foo` is) or it *is* the current instantiation and the name was not found in it or its non-dependent base classes and there are also dependent base classes.

Imagine what happens if we had a member function `h` within the above defined `A` class template

```
void h() {
  typename A<T>::questionable_type x;
}
```

In C++03, the language allowed to catch this error because there could never be a valid way to instantiate `A<T>::h` (whatever argument you give to `T`). In C++11, the language now has a further check to give more reason for compilers to implement this rule. Since `A` has no dependent base classes, and `A` declares no member `questionable_type`, the name `A<T>::questionable_type` is *neither* a member of the current instantiation *nor* a member of an unknown specialization. In that case, there should be no way that that code could validly compile at instantiation time, so the language forbids a name where the qualifier is the current instantiation to be neither a member of an unknown specialization nor a member of the current instantiation (however, this violation is still not required to be diagnosed).

Examples and trivia

You can try this knowledge on this answer and see whether the above definitions make sense for you on a real-world example (they are repeated slightly less detailed in that answer).

The C++11 rules make the following valid C++03 code ill-formed (which was not intended by the C++ committee, but will probably not be fixed)

```
struct B { void f(); };
struct A : virtual B { void f(); };

template<typename T>
struct C : virtual B, T {
  void g() { this->f(); }
};

int main() {
  C<A> c; c.g();
}
```

This valid C++03 code would bind `this->f` to `A::f` at instantiation time and everything is fine. C++11 however immediately binds it to `B::f` and requires a double-check when instantiating, checking whether the lookup still matches. However when instantiating `C<A>::g`, the Dominance Rule applies and lookup will find `A::f` instead.

Share Edit Follow Flag

edited May 23, 2017 at 12:18
Community Bot
1 ●1

answered Jul 10, 2013 at 20:02
Johannes Schaub - litb
478k ●121 ●867 ●1187

fyi - this answer is referenced here: stackoverflow.com/questions/56411114/... Much of the code in this answer doesn't compile on various compilers. – Adam Rackis Jun 1, 2019 at 23:41

1 @AdamRackis assuming that the C++ spec hasn't changed changed since 2013 (date that I wrote this answer), then the compilers that you tried your code with simply don't implement this C++11+-feature yet. – Johannes Schaub - litb Jun 2, 2019 at 20:20

118

> **Preface**
>
> This post is meant to be an *easy-to-read* alternative to litb's post.

> The underlying purpose is the same; an explanation to "When?" and "Why?" `typename` and `template` must be applied.

What's the purpose of `typename` and `template`?

`typename` and `template` are usable in circumstances other than when declaring a template.

There are certain contexts in C++ where the compiler must explicitly be told how to treat a name, and all these contexts have one thing in common; they depend on at least one *template-parameter*.

We refer to such names, where there can be an ambiguity in interpretation, as; "*dependent names*".

This post will offer an explanation to the relationship between *dependent-names*, and the two keywords.

A snippet says more than 1000 words

Try to explain what is going on in the following *function-template*, either to yourself, a friend, or perhaps your cat; what is happening in the statement marked (*A*)?

```
template<class T> void f_tmpl () { T::foo * x; /* <-- (A) */ }
```

It might not be as easy as one thinks, more specifically the result of evaluating (*A*) heavily *depends* on the definition of the type passed as template-parameter `T`.

Different `T`s can drastically change the semantics involved.

```
struct X { typedef int       foo;       }; /* (C) --> */ f_tmpl<X> ();
struct Y { static  int const foo = 123; }; /* (D) --> */ f_tmpl<Y> ();
```

The two different scenarios:

- If we instantiate the function-template with type *X*, as in (*C*), we will have a declaration of a *pointer-to int* named *x*, but;
- if we instantiate the template with type *Y*, as in (*D*), (*A*) would instead consist of an expression that calculates the product of *123* multiplied with some already declared variable *x*.

The Rationale

The C++ Standard cares about our safety and well-being, at least in this case.

To prevent an implementation from potentially suffering from nasty surprises, the Standard mandates that we sort out the ambiguity of a *dependent-name* by *explicitly* stating the intent anywhere we'd like to treat the name as either a *type-name*, or a *template-id*.

If nothing is stated, the *dependent-name* will be considered to be either a variable, or a function.

How to handle dependent names?

If this was a Hollywood film, *dependent-names* would be the disease that spreads through body contact, instantly affects its host to make it confused. Confusion that could, possibly, lead to an ill-formed perso-, erhm.. program.

A *dependent-name* is **any** name that directly, or indirectly, depends on a *template-parameter*.

```
template<class T> void g_tmpl () {
   SomeTrait<T>::type                   foo; // (E), ill-formed
   SomeTrait<T>::NestedTrait<int>::type bar; // (F), ill-formed
   foo.data<int> ();                         // (G), ill-formed
}
```

We have four *dependent* names in the above snippet:

- **E)**
  - **"type"** depends on the instantiation of `SomeTrait<T>`, which include `T`, and;
- **F)**
  - **"NestedTrait"**, which is a *template-id*, depends on `SomeTrait<T>`, and;
  - **"type"** at the end of (*F*) depends on *NestedTrait*, which depends on `SomeTrait<T>`, and;
- **G)**
  - **"data"**, which looks like a *member-function template*, is indirectly a *dependent-name* since the type of *foo* depends on the instantiation of `SomeTrait<T>`.

Neither of statement (*E*), (*F*) or (*G*) is valid if the compiler would interpret the *dependent-names* as variables/functions (which as stated earlier is what happens if we don't explicitly say otherwise).

The solution

To make `g_tmpl` have a valid definition we must explicitly tell the compiler that we expect a type in (*E*), a *template-id* and a *type* in (*F*), and a *template-id* in (*G*).

```
template<class T> void g_tmpl () {
   typename SomeTrait<T>::type foo;                            // (G), legal
   typename SomeTrait<T>::template NestedTrait<int>::type bar; // (H), legal
   foo.template data<int> ();                                  // (I), legal
}
```

Every time a *name* denotes a type, **all** *names* involved must be either *type-names* or *namespaces*, with this in mind it's quite easy to see that we apply `typename` at the beginning of our fully *qualified name*.

`template` however, is different in this regard, since there's no way of coming to a conclusion such as; *"oh, this is a template, then this other thing must also be a template"*. This means that we apply `template` directly in front of any *name* that we'd like to treat as such.

Can I just stick the *keywords* in front of any name?

> *"Can I just stick `typename` and `template` in front of any name? I don't want to worry about the context in which they appear..."* - `Some C++ Developer`

The rules in the Standard states that you may apply the keywords as long as you are dealing with a *qualified-name* (*K*), but if the name isn't *qualified* the application is ill-formed (*L*).

```
namespace N {
  template<class T>
  struct X { };
}
```

```
         N::         X<int> a; // ...  legal
typename N::template X<int> b; // (K), legal
typename template    X<int> c; // (L), ill-formed
```

Note: Applying `typename` or `template` in a context where it is not required is not considered good practice; just because you can do something, doesn't mean that you should.

Additionally there are contexts where `typename` and `template` are **explicitly** disallowed:

- **When specifying the bases of which a class inherits**

  Every name written in a derived class's *base-specifier-list* is already treated as a *type-name*, explicitly specifying `typename` is both ill-formed, and redundant.

  ```
                     // .------- the base-specifier-list
  template<class T> // v
  struct Derived      : typename SomeTrait<T>::type /* <- ill-formed */ {
    ...
  };
  ```

- **When the *template-id* is the one being referred to in a derived class's *using-directive***

  ```
  struct Base {
    template<class T>
    struct type { };
  };

  struct Derived : Base {
    using Base::template type; // ill-formed
    using Base::type;          // legal
  };
  ```

Share Edit Follow Flag

edited Jul 7, 2020 at 7:32
rayryeng
99.5k ●21 ●180 ●185

answered Jun 7, 2014 at 20:28
Filip Roséen - refp
59.8k ●19 ●144 ●189

29

This answer is meant to be a rather short and sweet one to answer (part of) the titled question. If you want an answer with more detail that explains why you have to put them there, please go here

The general rule for putting the `typename` keyword is mostly when you're using a template parameter and you want to access a nested `typedef` or using-alias, for example:

```
template<typename T>
struct test {
    using type = T; // no typename required
    using underlying_type = typename T::type // typename required
};
```

Note that this also applies for meta functions or things that take generic template parameters too. However, if the template parameter provided is an explicit type then you don't have to specify `typename`, for example:

```
template<typename T>
struct test {
    // typename required
    using type = typename std::conditional<true, const T&, T&&>::type;
    // no typename required
    using integer = std::conditional<true, int, float>::type;
};
```

The general rules for adding the `template` qualifier are mostly similar except they typically involve templated member functions (static or otherwise) of a struct/class that is itself templated, for example:

Given this struct and function:

```
template<typename T>
struct test {
    template<typename U>
    void get() const {
        std::cout << "get\n";
    }
};

template<typename T>
void func(const test<T>& t) {
    t.get<int>(); // error
}
```

Attempting to access `t.get<int>()` from inside the function will result in an error:

```
main.cpp:13:11: error: expected primary-expression before 'int'
     t.get<int>();
           ^
main.cpp:13:11: error: expected ';' before 'int'
```

Thus in this context you would need the `template` keyword beforehand and call it like so:

`t.template get<int>()`

That way the compiler will parse this properly rather than `t.get < int`.

Share Edit Follow Flag

edited May 23, 2017 at 12:10
Community Bot
1 ●1

answered Jun 6, 2014 at 22:23
Rapptz
20.1k ●4 ●70 ●86

1 This is, to me at least, indeed the **most efficient answer** with clear code examples. It should come before the thorough and detailed explanation by J. Schaub (@litb). – diemo Nov 8, 2021 at 10:24

20

```
typedef typename Tail::inUnion<U> dummy;
```

However, I'm not sure you're implementation of inUnion is correct. If I understand correctly, this class is not supposed to be instantiated, therefore the "fail" tab will never avtually fails. Maybe it would be better to indicates whether the type is in the union or not with a simple boolean value.

```
template <typename T, typename TypeList> struct Contains;

template <typename T, typename Head, typename Tail>
struct Contains<T, UnionNode<Head, Tail> >
{
    enum { result = Contains<T, Tail>::result };
};

template <typename T, typename Tail>
struct Contains<T, UnionNode<T, Tail> >
{
    enum { result = true };
};

template <typename T>
struct Contains<T, void>
{
    enum { result = false };
};
```

PS: Have a look at Boost::Variant

PS2: Have a look at typelists, notably in Andrei Alexandrescu's book: Modern C++ Design

Share Edit Follow Flag

answered Mar 4, 2009 at 13:37
Luc Touraille
75.7k ●15 ●85 ●135

inUnion<U> would be instantiated, if you for instance tried to call Union<float,bool>::operator=(U) with U==int. It calls a private set(U, inUnion<U>* = 0). – MSalters Mar 5, 2009 at 10:52

And the work with result=true/false is that I'd need boost::enable_if< >, which is incompatible with our current OSX toolchain. The separate template is still a good idea, though. – MSalters Mar 5, 2009 at 10:55

Luc means the typedef Tail::inUnion<U> dummy; line. that will instantiate Tail. but not inUnion<U>. it gets instantiated when it needs the full definition of it. that happens for example if you take the sizeof, or access a member (using ::foo). @MSalters anyway, you've got another problem: – Johannes Schaub - litb Mar 5, 2009 at 14:56

-sizeof(U) is never negative :) because size_t is an unsigned integer type. you will get some very high number. you probably want to do sizeof(U) >= 1 ? -1 : 1 or similar :) – Johannes Schaub - litb Mar 5, 2009 at 14:58

... then char f[sizeof(U) >= 1 ? -1 : 1] or -sizeof(U) is never valid. i read it long time ago but today morning i found the paragraph again: 14.6/7 . it's not required to refuse it but it may do that. however if you just put only the declaration of the template, it's all fine. – Johannes Schaub - litb Mar 5, 2009 at 15:09

3

C++20 aka C++2a

As outlined in this Proposal, C++20 / C++2a has further relaxed the requirements for the `typename` keyword. In particular, `typename` may now be omitted in all those places, where syntactically only a type is legal. So, if an unknown token must be a type, C++20 will actually treat it as a type. For backwards compatibility, `typename` may still be used, though.

In particular, most `using` and `typedef` declarations can now be written without `typename`. `typename` can also be omitted in the declaration of method return types (including trailing return types), in the declaration of method and lambda parameters and in the type argument to `static_cast`, `const_cast`, `dynamic_cast` and `reinterpret_cast`.

One notable exception, where `typename` is still required, is in the argument list of instantiations of user or library defined templates: Even, if that particular argument was declared to be a type, the `typename` keyword is still required. So `static_cast<A::B>(arg)` is legal in C++20, but `my_template_class<A::B>(arg)` is ill-formed, if A is a dependant scope and `my_template_class` expects a type.

A few examples:

```
class A { public: typedef int type; static const int val { 1 }; };
class B { public: typedef float type; static const int val { 2 }; };
template<typename T> class C {};
template<int I> class D {};
template<typename T> class X {
    T::type v;                                  // OK
    T::type f(T::type arg) { return arg; }      // OK
    T::type g(double arg) { return static_cast<T::type>(arg); } // OK
    // C<T::type> c1;                           // error
    D<T::val> d;                                // OK (as has always been)
    C<typename T::type> c2;                     // OK (old style)
    typedef T::type mytype;                     // OK
    using mytypeagain = T::type;                // OK
    C<mytype> c3;                               // OK (via typedef / using)
};
X<A> xa;
X<B> xb;
```

Share Edit Follow Flag

edited Apr 25, 2021 at 23:38
MSalters
166k ●8 ●148 ●332

answered Apr 25, 2021 at 7:23
Kai Petzke
1,380 ●13 ●24

As a DR for C++20, the `template` parser guide was made optional in the same contexts. – Davis Herring Aug 3, 2021 at 9:59

2

I am placing JLBorges's excellent response to a similar question verbatim from cplusplus.com, as it is the most succinct explanation I've read on the subject.

> In a template that we write, there are two kinds of names that could be used - dependant names and non- dependant names. A dependant name is a name that depends on a template parameter; a non-dependant name has the same meaning irrespective of what the template parameters are.
>
> For example:
>
> ```
> template< typename T > void foo( T& x, std::string str, int count )
> {
>     // these names are looked up during the second phase
>     // when foo is instantiated and the type T is known
>     x.size(); // dependant name (non-type)
>     T::instance_count ; // dependant name (non-type)
>     typename T::iterator i ; // dependant name (type)
>
>     // during the first phase,
>     // T::instance_count is treated as a non-type (this is the default)
>     // the typename keyword specifies that T::iterator is to be treated as a type.
>
>     // these names are looked up during the first phase
>     std::string::size_type s ; // non-dependant name (type)
>     std::string::npos ; // non-dependant name (non-type)
>     str.empty() ; // non-dependant name (non-type)
>     count ; // non-dependant name (non-type)
> }
> ```
>
> What a dependant name refers to could be something different for each different instantiation of the template. As a consequence, C++ templates are subject to "two-phase name lookup". When a template is initially parsed (before any instantiation takes place) the compiler looks up the non-dependent names. When a particular instantiation of the template takes place, the template parameters are known by then, and the compiler looks up dependent names.
>
> During the first phase, the parser needs to know if a dependant name is the name of a type or the name of a non-type. By default, a dependant name is assumed to be the name of a non-type. The typename keyword before a dependant name specifies that it is the name of a type.

Summary

Use the keyword typename only in template declarations and definitions provided you have a qualified name that refers to a type and depends on a template parameter.

Share Edit Follow Flag

edited Jun 20, 2020 at 9:12
Community Bot
1 ●1

answered Aug 11, 2018 at 12:20
KeyC0de
4,047 ●6 ●39 ●55

1

Dependent name is a name depends on template parameters, we need to instruct compiler in order to compile the template class/function properly before actually instiatiate them.

- typename -> tell compiler the dependent name is an actual type

  ```
  template <class T>
  struct DependentType
  {
    typename T::type a;
    using Type=typename T::type;
  };
  ```

- template -> tell compiler the dependent name is a template function/class

  ```
  template <class T>
  struct DependentTemplate
  {
    // template function
    template <class U>
    static void func() {}

    // template class
    template <class U>
    struct ClassName{};
  };


  template <class T1, class T2>
  void foo()
  {
    // 3 ways to call a dependent template function
    DependentTemplate<T1>::template func<T2>();
    DependentTemplate<T1>().template func<T2>();
    (new DependentTemplate<T1>())->template func<T2>();

    // You need both typename and template to reference a dependent template class
    typename DependentTemplate<T1>::template ClassName<T2> obj;
    using Type=typename DependentTemplate<T1>::template ClassName<T2>;
  }
  ```

Share Edit Follow Flag

edited May 15, 2021 at 0:40

answered May 15, 2021 at 0:07
Baiyan Huang
5,998 ●6 ●40 ●69