Why can't I declare a type that derives from a generic type parameter?

devblogs.microsoft.com/oldnewthing/20090814-00

August 14, 2009

Raymond Chen

A lot of questions about C# generics come from the starting point that they are just a cutesy C# name for C++ templates. But while the two may look similar in the source code, they are actually quite different.

C++ templates are macros on steroids. No code gets generated when a template is "compiled"; the compiler merely hangs onto the source code, and when you finally instantiate it, the actual type is inserted and code generation takes place.

```
// C++ template
template<class T>
class Abel
{
public:
 int DoBloober(T t, int i) { return t.Bloober(i); }
};
```

This is a perfectly legal (if strange) C++ template class. But when the compiler encounters this template, there are a whole bunch of things left unknown. What is the return type of `T::Bloober`? Can it be converted to an `int`? Is `T::Bloober` a static method? An instance method? A virtual instance method? A method on a virtual base class? What is the calling convention? Does `T::Bloober` take an `int` argument? Or maybe it's a `double`? Even stranger, it might accept a `Canoe` which gets converted from an `int` by a converting constructor. Or maybe it's a function that takes two parameters, but the second parameter has a default value.

Nobody knows the answers to these questions, not even the compiler. It's only when you decide to instantiate the template

```
Abel<Baker> abel;
```

that these burning questions can be answered, overloaded operators can be resolved, conversion operators can be hunted down, parameters can get pushed on the stack in the correct order, and the correct type of `call` instruction can be generated.

In fact, the compiler doesn't even care whether or not `Baker` has a `Bloober` method, as long as you never call `Abel<Baker>::DoBloober`!

```
void f()
{
 Abel<int> a; // no error!
}
void g()
{
 Abel<int> a;
 a.DoBloober(0, 1); // error here
}
```

Only if you actually call the method does the compiler start looking for how it can generate code for the `DoBloober` method.

C# generics aren't like that.

Unlike C++, where a non-instantiated template exists only in the imaginary world of potential code that could exist but doesn't, a C# generic results in code being generated, but with placeholders where the type parameter should be inserted.

This is why you can use generics implemented in another assembly, even without the source code to that generic. This is why a generic can be recompiled without having to recompile all the assemblies that use that generic. The code for the generic is generated *when the generic is compiled*. By comparison no code is generated for C++ templates until the template is instantiated.

What this means for C# generics is that if you want to do something with your type parameter, it has to be something that the compiler can figure out how to do *without knowing what* `T` *is*. Let's look at the example that generated today's question.

```
class Foo<T>
{
 class Bar : T
 { ... }
}
```

This is flagged as an error by the compiler:

```
error CS0689: Cannot derive from 'T' because it is a type parameter
```

Deriving from a generic type parameter is explicitly forbidden by 25.1.1 of the C# language specification. Consider:

```
class Foo<T>
{
 class Bar : T
 {
   public void FlubberMe()
   {
     Flubber(0);
   }
 }
}
```

The compiler doesn't have enough information to generate the IL for the `FlubberMe` method. One possibility is

```
ldarg.0        // "this"
ldc.i4.0    // integer 0 - is this right?
call T.Flubber // is this the right type of call?
```

The line `ldc.i4.0` is a guess. If the method `T.Flubber` were actually `void Flubber(long l)`, then the line would have to be `ldc.i4.0; conv.i8` to load an 8-byte integer onto the stack instead of a 4-byte integer. Or perhaps it's `void Flubber(object o)`, in which case the zero needs to be boxed.

And what about that call instruction? Should it be a `call` or `callvirt`?

And what if the method returned a value, say, `string Flubber(int i)`? Now the compiler also has to generate code to discard the return value from the top of the stack.

Since the source code for a generic is not included in the assembly, all these questions have to be answered at the time the generic is compiled. Besides, you can write a generic in Managed C++ and use it from VB.NET. Even saving the source code won't be much help if the generic was implemented in a language you don't have the compiler for!

Raymond Chen

Follow