Doc No: SC22/WG21/N1945=06-0015
Project: JTC1.22.32
Date: 2006-02-24
Reply To: Anthony Williams
Just Software Solutions Ltd
Email: anthony@justsoftwaresolutions.co.uk

Names, Linkage, and Templates

1 Background

My previous paper on this topic (N1427) met a lukewarm reception when I presented it to the EWG at the Oxford meeting in 2003 — some people were strongly for this proposal, and others not so. Brief discussions with a selection of compiler writers indicated to me that they all thought this was implementable, though for some this was more of an issue than others. Subsequently, I understand that this was further discussed at the Redmond meeting in 2004, and lacking sufficient support was put aside in favour of other proposals.

The issue was raised again in discussions on the cxxstd-core reflector, particularly the thread following cxxstd-core-10977, in August/September 2005. Whereas the original proposal was purely to cover the use of local types with templates, these messages indicate to me that the issue is wider than that, and includes unnamed types (such as enums) declared at namespace scope. I believe this led to a brief discussion on the matter at the October 2005 meeting.

This proposal also has direct bearing on Core Issue 488, where the current proposed resolution would make passing a local type as an argument to template function a type deduction failure.

There are two key drivers behind this proposal: consistency, and ease of use.

1.1 Consistency

From the point of view of a C++ programmer, there is little difference between a class declared at function scope, and a class declared at namespace or class scope; it is thus surprising to find that there is such a difference in behaviour — the class defined at namespace scope can be used as a template argument, whereas the class defined at function scope cannot.

```
template<typename T>
void foo(T const& t){}

struct X{};

int main()
{
  struct Y{};
  foo(X()); // well-formed
  foo(Y()); // ill-formed
}
```

Likewise, it is surprising to find that an unnamed enum declared at namespace scope is excluded from template type deduction, whereas a named enum is permitted.

```
template<typename T>
void foo(T const& t){}

enum X { x };
enum { y };

int main()
{
  foo(x); // well-formed
  foo(y); // ill-formed
}
```

Under this proposal, both these examples would be well-formed.

1.2 Ease of Use

Much of the functionality provided by the Standard Library, and popular third-party libraries such as Boost, is provided by means of templates, whether they be containers such as `std::vector`, algorithms such as `std::for_each`, or general purpose utilities such as `tr1::shared_ptr`.

There are many cases where a class is only needed for the scope of a function, whether it is because it is used to group data, or to provide a predicate for the application of an algorithm, or whatever. It would therefore be most convenient to define such a class at function scope, rather than class or namespace scope. Indeed, if the function in question is a member function, then there may be impediments to defining the class at namespace scope, due to use of private members of the enclosing class, which would force the new class to be defined at class scope. If this class definition is shared between translation units, then this causes additional coupling between them due to the exposure of what is essentially an implementation detail of a member function.

It is for such reasons that programmers often resort to using `for` loops in favour of algorithms from the Standard Library, as the overhead is just too great for all but trivial cases.

Under this proposal, these problems would evaporate, and local types and unnamed types would work with template-based algorithms and containers. There would be no substantive difference compared to namespace- and class-scope types, or named types, in these circumstances.

2 Proposal Summary

The key part of the proposal is that the restrictions on which types can be used as template type parameters from 14.3.1 paragraph 2 be lifted. In order to facilitate this, it is proposed that all local types and unnamed types have a “name for linkage purposes” created by the compiler, much the same as unnamed namespaces have a unique name that is created by the compiler. Many of those entities which currently fall under the category of “no linkage” in section 3.5 of the C++ Standard, would now qualify for external linkage:

— Unnamed enumerations

— Enumerators belonging to unnamed enumerations

— Names declared in a local scope

In order to avoid unnecessary complexity, local types have the same linkage as their enclosing function, and unnamed types have the same linkage they would have if they were named. In particular, this means that types declared within namespace-scope `static` functions still have no linkage — if this is inconvenient for the programmer, the function can easily be moved inside an unnamed namespace instead. As this use of `static` is deprecated in Standard C++ anyway, going to a lot of trouble for such a corner case does not seem justified.

3 Required Changes

3.5 Program and Linkage [basic.link]

Add a new paragraph after paragraph 4:

> An unnamed class (or enum) defined at namespace scope, and not covered by paragraph 4, is assigned an identifier by the implementation for linkage purposes, which is not the same as any other identifier in the program. This identifier, and the class or enumeration it identifies, shall have external linkage.

Add two new paragraphs following the existing paragraph 5:

> Inside a function which itself has external linkage, a name has external linkage if it is the name of:
>
> — a named class (or enum) defined at block scope; or
>
> — an unnamed class (or enum) defined in a typedef declaration at block scope, in which the class (or enum) has the typedef name for linkage purposes.

and

> Unnamed classes and enumerations not covered by the preceding paragraphs are assigned an identifier by the implementation, which shall not be the same as any other identifier in the program. This identifier, and the class or enumeration so-named, shall have external linkage if, and only if, it would have had external linkage when specified directly in the source. [Example:

```
template<typename T>
void f(T t);

void g()
{
  enum { x }; // equivalent to enum unique_1 { x };
  // g() has external linkage,
  // therefore unique_1, and x, have external linkage
  f(x); // well-formed
}

static void h()
{
  enum { y }; // equivalent to enum unique_2 { y };
  // h() has internal linkage,
  // therefore unique_2, and y, have no linkage
  f(y); // ill-formed
}

enum { z }; // equivalent to enum unique_3 { z };
// unique_3 and z have external linkage

void j()
{
  f(z); // well-formed
}
```

—end example] [Note: Where such a unique identifier is assigned at namespace scope, the same definition in another translation unit will yield a distinct name.]

Change paragraph 8 from:

Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage. A name with no linkage (notably, the name of a class or enumeration declared in a local scope (3.3.2)) shall not be used to declare an entity with linkage. If a declaration uses a typedef name, it is the linkage of the type name to which the typedef refers that is considered. [Example:

```
void f()
{
  struct A { int x; }; // no linkage
  extern A a; // ill-formed
  typedef A B;
  extern B b; // ill-formed
}
```

—end example] This implies that names with no linkage cannot be used as template arguments (14.3).

to:

Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage. A name with no linkage shall not be used to declare an entity with linkage. If a declaration uses a typedef name, it is the linkage of the type name to which the typedef refers that is considered. [Example:

```
static void f() // internal linkage
{
  struct A { int x; }; // no linkage
  extern A a; // ill-formed
  typedef A B;
  extern B b; // ill-formed
}
```

—end example]

14.3.1 Template type arguments [temp.arg.type]

Modify paragraph 2 to allow local types and unnamed types to be template arguments, and remove the example:

[Remove: A local type,] a type with no linkage***[Remove: , an unnamed type]*** or a type compounded from any of these types shall not be used as a template-argument for a template type-parameter. ***[Remove: [Example:***

```
template <class T> class X { /* ... */ };
void f()
{
  struct S { /* ... */ };
  X<S> x3; // error: local type
           // used as template-argument
  X<S*> x4; // error: pointer to local type
            // used as template-argument
}
```

—end example]] [Note: a template type argument may be an incomplete type (3.9).]

4 Acknowledgements

Thanks to those who have read and commented on drafts of this proposal.