Doc No:	SC22/WG21/N2187=07-0047
Project:	JTC1.22.32
Date:	2007-03-06
Reply To:	Anthony Williams
	Just Software Solutions Ltd
	Email: anthony@justsoftwaresolutions.co.uk

Names, Linkage, and Templates (rev 1)

1 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.

2 Implementation Experience

Since my previous paper on this topic (N1945) was written, several compiler vendors have experimented with implementations of this feature, and the Microsoft Visual C++ 2005 compiler provides this feature as part of the Microsoft Extensions mode (with the /Ze compiler switch). Therefore there is now the possibility of real experience of code written using this feature, and compiler vendors should be in a better position to discuss the consequences for implementations.

3 Motivation

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

3.1 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.

3.2 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.

3.3 Examples

3.3.1 Comparators for associative containers

One scenario where local classes would be beneficial is when a function needs to sort the values in a container in a particular order, which is only needed for that function. For example, consider a `Payload` class defined as

```
// payload.h
class Payload
{
public:
    bool compareName(Payload const &rhs);
    friend bool operator<(Payload const &lhs,Payload const &rhs);
};
std::ostream & operator<<(std::ostream &os,Payload const &rhs);
```

where the provided implementation of `operator<` sorts by some attribute not shown, other than the “name”. In order to sort a collection of `Payload` instances in name order rather than the default order, a custom comparator must be provided:

```
void printByName(std::set<Payload> const &s)
{
   class ByName
   {
   public:
      bool operator()(Payload const &lhs,Payload const &rhs)
      {
         return lhs.compareName(rhs);
      }
   };
   std::set<Payload, ByName> s2(s.begin(),s.end());
   std::copy(s2.begin(),s2.end(),
             std::ostream_iterator<Payload>(std::cout,"\n"));
}
```

In C++03 this is not well-formed code, since the comparator is a local class. Under this proposal it would become well-formed. The C++03 solution is to move the comparator to namespace scope, but this requires ensuring that its name doesn't clash with anything else. Using an anonymous namespace can restrict the scope of potential name clashes to the specific translation unit, but for many applications this can still be a large body of code.

```
namespace
{
   class SomeTUUniqueName
   {
   public:
      bool operator()(Payload const &lhs,Payload const &rhs)
      {
         return lhs.compareName(rhs);
      }
   };
}

void printByName(std::set<Payload> const &s)
{
   std::set<Payload, SomeTUUniqueName> s2(s.begin(),s.end());
   std::copy(s2.begin(),s2.end(),
             std::ostream_iterator<Payload>(std::cout,"\n"));
}
```

The use of a namespace helper class splits the implementation across the source file, and makes it necessary to pick a different name for each such class in the translation unit. If there are several functions that require their own sorting, this can quickly become tedious.

Though it is important to provide good names for local classes, just as it is with any class, the reduced scope means that there is more context surrounding its use: `ByName` can be sufficient for a local class, whereas a namespace-scope class might be better named `ComparePayloadByName`.

3.3.2 Callbacks

Another scenario in which local classes would be useful is for a callback. There are many examples of functions where the caller must provide a callback which is invoked at specific points during the processing. This could be a function called to update progress on a time-consuming operation, or registering a callback to be invoked in response to specific user input.

In many cases, the function taking the callback is itself a template, or it accepts the callback through something akin to `std::tr1::function` in order to allow callable objects as well as plain function pointers to be passed. Under the current standard, this precludes the use of local classes, as they cannot be passed as template parameters.

In a GTK-based application, there might be a function `errorDialog` to display a dialog box to the user on error. As well as taking a message to display, it also takes a callback which is invoked when the user makes a selection. In this specific example, `errorDialog()` takes something like `std::tr1::function` as its last parameter and registers it as the handler for a GTK signal.

Because `std::tr1::function` is a template the function object can't be local, so the class has to be namespace-scope, and therefore more fully-fledged, with private data and an explicit constructor, to avoid accidental misuse elsewhere.

```
  // Functor that gives focus to a widget
  // (for use with DialogCallback, ignores user’s selection)
  class DialogFocuser:
    std::unary_function<GtkDialogSelection,void>
  {
  public:
    explicit DialogFocuser(GtkWidget* w):
      m_widget(w) { }

    void operator()(GtkDialogSelection) const
    { gtk_widget_grab_focus(m_widget); }

  private:
    GtkWidget* m_widget;
  };

  void Ticket::showError(const std::string& msg,
                         GtkWidget& focus_next)
  {
    errorDialog(GTK_WINDOW(m_screen), msg,
                DialogFocuser(&focus_next));
  }
```

Had it been possible to use a local class, which is thus not open to (mis)use in other functions, it would be reasonable to make the class much simpler:

```
  void Ticket::showError(const std::string& msg,
                         GtkWidget& focus_next)
  {
    struct {
      GtkWidget* m_widget;
      void operator()(GtkDialogSelection) const
      { gtk_widget_grab_focus(m_widget); }
    } focuser = { focus_next };

    errorDialog( GTK_WINDOW(m_screen), msg, focuser);
  }
```

If local classes were permitted, it would be possible to extract a factory function for retrieving the callback:

```
typedef std::tr1::function<void (GtkDialogSelection)> DialogCallback;
```

```
DialogCallback getCallback(GtkWidget* w)
{
  struct {
    // ...
  } focuser = { ... };
  return DialogCallback(focuser);
}
```

In some cases combining local classes with function wrappers like this could be simpler and more expressive than an equivalent `std::tr1::bind` expression. `getCallback` could be made a **`virtual`** function, for example.

4 Required Changes

The following changes are with respect to the current working draft, N2134.

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 a name for linkage purposes by the implementation, which shall not be the same as any other name in the program. This name, 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 the example in paragraph 8 to refer to a local type in a static function, since such a type will continue to have no linkage under this proposal:

[Remove:

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

] [Insert:

```
static void f() // internal linkage
{
  struct A { int x; }; // no linkage
```

```
    extern A a; // ill-formed
    typedef A B;
    extern B b; // ill-formed
  }
  void g() // external linkage
  {
    struct A { int x; }; // external linkage
    extern A a; // OK
    typedef A B;
    extern B b; // OK
  }
```

]

14.3.1 Template type arguments [temp.arg.type]

Modify the example in paragraph 2 to refer to a local class in a `static` function:

A type without linkage (3.5) shall not be used as a template-argument for a template type-parameter.

[Example:

[Remove:

```
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
}
```

]

[Insert:

```
template <class T> class X { /* ... */ };
static void f() // internal linkage
{
  struct S { /* ... */ }; // no linkage
  X<S> x3; // error: local type in static function
           // used as template-argument
  X<S*> x4; // error: pointer to local type in static
            // function used as template-argument
}
```

]

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

5 Acknowledgements

Thanks to those who have read and commented on drafts of this proposal, especially the members of the BSI C++ Panel. In particular, I would like to thank Roger Orr and Jonathan Wakely for providing concrete examples.