Doc No: SC22/WG21/N1427=03-0009
This is a revision of SC22/WG21/N1353
Project: JTC1.22.32
Date: 10 February 2003
Reply To: Anthony Williams
Beran Instruments Ltd
Email: anthony.williams@anthonyw.cjb.net

Making Local Classes more Useful

1 The Problem

Currently, classes can be declared and defined at block scope, but these classes and their member functions have no linkage, which prohibits their use as template arguments.

The majority of the Standard library classes and functions are templates, so this precludes their use with local types. In particular, it would be useful to be able to define a functor class at local scope, for use as a sort key with `std::map`, or as a parameter to an algorithm such as `std::for_each` or `std::transform`; or to be able to have a `std::vector<>` holding objects of a local type.

Also, this is an inconsistency in the standard which makes it harder to teach and learn — the properties of a class, and the language features available when writing a class depend on where it is defined (i.e. whether it is defined at local scope, or at namespace or class scope).

Furthermore, it is desirable to allow friend functions to be declared and defined inside a local class, as for a normal class, to allow the definition of non-member operators. It is also desirable to allow templates to be friends. This proposal would allow both these cases.

Currently, the work-around is to move the definition of the type out to namespace scope, or to class scope, where the enclosing class has linkage, separating the definition from the point of use, and polluting the namespace or class scope unnecessarily. For local classes declared inside functions defined in precisely one translation unit, this can be alleviated by placing the class that would be local in the anonymous namespace, so the pollution is at least restricted to that translation unit. If the function definition is visible across multiple translation units, (because it is inline, or a function template, or a member function of a template class, for example), then this is not possible, since referring to a class in the anonymous namespace under such conditions would lead to a violation of the ODR.

1.1 Examples

What follows is a set of examples showing code that one might like to write, which would be permitted under the current proposal, and the current workaround. The desired code is shown on the left, and the workaround on the right.

1.1.1 Containers of local classes

If you want to have a container of a local class, you currently need to write a local container class, or move the local class to namespace or class scope.

```
void func()
{
  struct Name
  {
    std::string firstName;
    std::string lastName;
  };

  std::vector<Name> names;
}
```

```
namespace
{
  // annotate struct name
  // to avoid clashes
  struct func_Name
  {
    std::string firstName;
    std::string lastName;
  };
}

void func()
{
  std::vector<func_Name> names;
}
```

1.1.2 Using local classes as functors

If you want to use a local class as a functor, you currently have to move it to class or namespace scope.

```
void func()
{
  struct Name
  {
    std::string firstName;
    std::string lastName;
  };

  struct SortByFirstName
  {
    bool
    operator()(const Name& lhs,
               const Name& rhs)
      const
    {
      return lhs.firstName
        <rhs.firstName;
    }
  };

  struct SortByLastNameReversed
  {
    bool
    operator()(const Name& lhs,
               const Name& rhs)
      const
    {
      return lhs.lastName
        >rhs.lastName;
    }
  };

  std::set<Name,
    SortByFirstName> names;
  std::set<Name,
    SortByLastNameReversed>
    reversedNames(names.begin(),
                  names.end());
}
```

```
namespace
{
  struct func_Name
  {
    std::string firstName;
    std::string lastName;
  };

  struct func_SortByFirstName
  {
    bool
    operator()(const func_Name& lhs,
               const func_Name& rhs)
      const
    {
      return lhs.firstName
        <rhs.firstName;
    }
  };

  struct func_SortByLastNameReversed
  {
    bool
    operator()(const func_Name& lhs,
               const func_Name& rhs)
      const
    {
      return lhs.lastName
        >rhs.lastName;
    }
  };
}
void func()
{
  std::set<func_Name,
    func_SortByFirstName> names;
  std::set<func_Name,
    func_SortByLastNameReversed>
    reversedNames(names.begin(),
                  names.end());
}
```

1.1.3 Defining Non-member friend functions in local classes

If you want to enforce invariants in your local class by using private data members, then all functions which need access to those data members must currently also be members. In particular, this prohibits binary operators that allow conversions on the LHS (whether or not members are private).

The workaround is to move the class to namespace or class scope, as above.

```
void func()
{
  class Name
  {
  private:
    std::string firstName;
    std::string lastName;
  public:
    // constructors etc.

    friend bool operator<
      (const Name& lhs,
       const Name& rhs)
    {
      return lhs.firstName
        <rhs.firstName;
    }
  };
  std::set<Name> names;
}
```

```
namespace
{
  class func_Name
  {
  private:
    std::string firstName;
    std::string lastName;
  public:
    // constructors etc.

    friend bool operator<
      (const func_Name& lhs,
       const func_Name& rhs)
    {
      return lhs.firstName
        <rhs.firstName;
    }
  };
}
void func()
{
  std::set<func_Name> names;
}
```

1.1.4 Declaring templates as friends

If your local class has private data, then there is currently no way to allow a function template, or member functions of a class template, to access that private data, even if you know it is desirable. The workaround is to make the data public, and thus allow for the possiblity of uncontrolled modifications that would violate the class invariants, or to move the class to namespace or class scope. This is only an issue if local classes can be used as template arguments, since otherwise one has to move the class to a wider scope anyway.

```
template<typename T,typename U>
void foo(const T& t,const U& u);

void func()
{
  class Local
  {
  private:
    // private members
    // needed by foo
    template<typename T,
             typename U>
    friend void foo(const T&,
                    const U&);
  };

  Local obj;
  foo(obj,3);
  foo(obj,2.718);
  foo(obj,"hello");
}
```

```
template<typename T,typename U>
void foo(const T& t,const U& u);

void func()
{
  class Local
  {
  private:
    // private members
    // not needed by foo
  public:
    // private members
    // needed by foo
    // made public
  };

  Local obj;
  foo(obj,3);
  foo(obj,2.718);
  foo(obj,"hello");
}
```

2 The Proposal

In brief, this proposal is to make local classes more useful by allowing them to be used as template type arguments, and increasing the range of acceptable friend declarations in local classes to include declarations that are definitions, and template declarations.

3 Interactions and Implementability

3.1 Interactions

3.1.1 Naming

Local classes declared in different scopes with the same name are distinct entities. This proposal does not affect this. Consequently, implementations which require unique names for distinct entities must have some mechanism for ensuring that the linkage names of two distinct local classes declared with the same identifier are distinct. See 3.2.1.

3.1.2 Access to local variables from the enclosing function

This proposal does not involve any change to name lookup rules.

Local classes should have no special access to automatic variables from the scope in which they are declared, as this would require passing a reference to the stack frame into each instance of a local object. This would have a large impact, and is not thought worth the change.

However, local classes should have full access to variables of static storage duration declared within the enclosing function that are in scope at the point of declaration of the local class, as is currently permitted:

```
void f()
{
  static int i1;

  class x
  {
    x()
    {
      i1=1; // OK, i1 is in scope here
      i2=2; // Error, i2 not yet in scope
    }
  };

  static int i2;
}
```

3.1.3 Static Instances of Local Class Objects

Variables declared at block scope may have static storage duration, in which case all references to that variable within the given scope refer to the same instance of the variable, even if the function is declared **`inline`** and used in multiple translation units. In order to maintain this functionality, even when the static object has local-class-type, the algorithm for generating the unique name must be such that the same names are generated for the same function definition in different translation units.

Consequently, the algorithm for generating the unique names must depend solely on the scope in which the current function definition lies, the name and signature of the enclosing function, any relevant template parameters, and the information available within the function definition. The precise details of the algorithm remain implementation defined.

3.1.4 Local Class within Local Classes

Local classes can have member functions, which introduce block scopes, that may therefore have their own local classes, and so forth. All such classes should also be distinct types (as currently), and should also be useable as template argumetns.

3.1.5 Forward Declarations and Out-of-line Member Definitions

No change is proposed to the language rules regarding forward declarations and out-of-line member definitions — both remain illegal for local classes.

3.1.6 Templates

The interaction of local classes with templates is a key part of this proposal. Firstly, if a local class is defined within a function template, or a member function of a class template, it is essential that every unique instantiation of the template yields a unique type for the local class, just as every unique instantiation of the template yields a unique address for the function in which the local class is defined. If a local class is defined within function which is itself a template, or is a member of a template, then instantiations of the template with the same set of template parameters must yield the same instance of the local class, even if the instantiations are performed in different translation units.

Secondly, it is now possible to instantiate a template with a local class as a template parameter. If the same template is instantiated for different local classes of the same name, each instantiation should be unique.

For consistency with non-local classes, it may be desirable to allow local classes to be templates, or to have member templates, or even to allow specializations of templates for local classes. However, any of these would require a syntax change, and it has been drawn to my attention that there may be implementation problems. Therefore I do not propose such extensions.

3.1.7 Friends

Currently, local classes are only permitted to have friends that have already been declared in another scope. This can be inconvenient, as described in 1. Therefore I propose that friends declared for the first time within a normal class are members of the enclosing lexical scope, as in sections 11.4, 14.5.3 and 14.6.5 of the Standard, and I propose extending this to cover local classes. In particular, this would allow the definition of binary operators that operate on the local class, and accept conversions on the left hand side. I also propose to allow the declaration of friends that are templates, provided the template has already been declared.

3.2 Implementability

3.2.1 Naming

For local classes to be usable as template arguments, we need a way in which they can be uniquely identified, so that classes or objects of the same name from a different scope do not cause name clashes. Note that many implementations do this already, to ensure that if typeid is applied to a local class then the correct behaviour ensues.

To resolve this issue, I propose that each block scope that defines local classes be given a unique identifier, similar to the way that anonymous namespaces are given unique identifiers in different translation units. The local classes defined within each block scope are then made members of a namespace with this uniquely-identifying name. Since this name is compiler-generated and unique, there is thus no way of identifying a local class from outside its block. e.g.

```
void f()
{
  class X
  {};
}
```

is (almost) equivalent to

```
namespace __unique_name_1
{
  class X
  {};
}

void f()
{
  using __unique_name_1::X;
}
```

and

```
void f()
{
  class X
  {};

  {
    class X
    {};
  }

  {
    class X
    {};

    {
      class X
      {};
    }
  }
}
```

is (almost) equivalent to

```
namespace __unique_name_1
{
  class X
  {};
  namespace __unique_name_2
  {
    class X
    {};
  }
  namespace __unique_name_3
  {
    class X
    {};
    namespace __unique_name_4
    {
      class X
      {};
    }
  }
}

void f()
{
  using __unique_name_1::X;

  {
    using __unique_name_2::X;
  }

  {
    using __unique_name_3::X;

    {
      using __unique_name_4::X;
    }
  }
}
```

Note that these examples with namespaces are purely for exposition and are not intended to reflect the actual way in which names are looked up, which should remain unchanged.

It is essential that different overloads of the same function name yield different unique names for their enclosed scopes, and that different instantations or specializations of the same function template also yield different unique names, so that name lookup rules remain unchanged, and the One Definition Rule is not violated. It is also essential that a local class defined within a function definition visible in more than one translation unit (such as a local class defined in an inline function or function template definition) be given the same unique name in all translation units, to ensure the ODR is not violated.

4 Required Changes

3.5 Program and Linkage [basic.link]

Change paragraph 8 to the last sentence:

> 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; // illformed
>   typedef A B;
>   extern B b; // illformed
> }
> ```
>
> —*end example*] ***[Remove: This implies that names with no linkage cannot be used as template arguments (14.3).]***

11.4 Friends [class.friend]

Remove the restriction on defining friend functions in local classes from paragraphs 5:

> A function can be defined in a friend declaration of a class if and only if ***[Remove: the class is a nonlocal class (9.8),]*** the function name is unqualified, and the function has namespace scope. [*Example:*
>
> ```
> class M {
>   friend void f() { } // definition of global f, a friend of M,
>                       // not the definition of a member function
> };
> ```
>
> —*end example*] Such a function is implicitly inline. A friend function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not (3.4.1).

Modify paragraph 9 to allow friend functions to be first declared in a local class only if they are also defined:

> If a friend declaration appears in a local class (9.8) and the name specified is an unqualified name, a prior declaration is looked up without considering scopes that are outside the innermost enclosing nonclass scope. For a friend function declaration, if there is no prior declaration, **and the declaration is not a definition**, the program is illformed.

For a friend class declaration, if there is no prior declaration, the class that is specified belongs to the innermost enclosing nonclass scope, but if it is subsequently referenced, its name is not found by name lookup until a matching declaration is provided in the innermost enclosing nonclass scope. [*Example:*

```
class X;
void a();
void f() {
  class Y;
  extern void b();
  class A {
    friend class X; // OK, but X is a local class, not ::X
    friend class Y; // OK
    friend class Z; // OK, introduces local class Z
    friend void a(); // error, ::a is not considered
    friend void b(); // OK
    friend void c(); // error
    friend void d(const A&) {} // OK, introduces a new function
                               // that is found by ADL
  };
  X *px; // OK, but ::X is found
  Z *pz; // error, no Z is found
  A a;
  d(a); // finds friend function by ADL
}
```

—*end example*]

14.3.1 Template type arguments [temp.arg.type]

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

[Remove: A local type,] a type with no linkage **(other than local types)**, 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).]

5 Acknowledgements

Thanks to those who have read and commented on drafts of this proposal, especially John Spicer, Andy Sawyer, Roger Orr, Lois Goldthwaite, Christopher Hill, Garry Lancaster, John Washington, Alan Lenton and Alan Griffiths.