Doc No: SC22/WG21/N1605
J16/04-0045
Date: 13-Feb-2004
Project: JTC1.22.32
Reply to: Daniel Gutson
danielgutson@hotmail.com

EXTENDING TEMPLATE TYPE PARAMETERS I

Namespace and scope

1. The problem

There is no way of passing a scope –as a type definitions container- as a template parameter. Namespaces cannot be specified as template parameters.

Templates cannot accept *general* 'scopes' as parameters; only classes and structures are accepted for this purpose. There is no way of specifying a general scope-type as template parameter type, for applying the '::' operator –regardless it is a structure, class or namespace-.

This paper proposes both the ability to pass namespaces as template parameters, and the addition of the 'scope' notion to the 'type, non-type, and template' set (**temp.arg**).

- Why is the problem important?

a) Specifying different type-definitions sources: namespaces are usually a source of type definitions. However, namespaces are not allowed as template parameters.
b) Differentiation between types, non-types, and scopes: a scope is qualitatively different to a type and a non-type concept. Scopes cannot be instantiated, but used as identifiers-definitions repository (i.e., they can be the left operand of the '::' operator). The importance of this difference involves two facts:
    a. Forces the template implementation to use the scope template type parameter as a scope, preventing instantiation or being used as a type or non-type template parameter.
    b. Acts as a self-documentation feature, providing the information to the template client about the template parameter nature.

- How are people addressing, or working around the problem today?

Structures are commonly used as definition sources. The concept of a structure/class is misused in that case [*Example*

```
struct InfoRep1
{
      enum { value = 1 };
      typedef char Type;
};
```

```
struct InfoRep2
{
      enum { value = 2 };
      typedef int Type;
};

template <class Rep> class C
{
      Rep::Type t;
      int f()    {    return Rep::value;    }
};

C<InfoRep1> c1; C<InfoRep2> c2;
```

-*end example*] since empty structures (in terms of allocable members) must be defined, while namespaces are more suitable candidates for containing definitions.

- This feature fits in the following subset of categories mentioned in the proposal template:

- improve support for system programming: allows the usage of namespaces for the purpose they are intended to be. Additionally, allows no to be forced to use structures for containing definitions.
- <u>improve support for library building</u>:
    i. <u>Abstraction</u>: by using the scope concept, allows library builders to be abstracted whether the scope is a structure, namespace, or class.
    ii. <u>Misuse prevention</u>: prevents a scope template parameter to be used as an instantiable type.

2. <u>The proposal</u>

Add the concept of 'scope' to the template parameter possibilities. Use the 'namespace' keyword for declaring a scope parameter, as extension to the template type-parameter clause.

2.1. <u>Basic Cases</u>

```
//rewriting the previous example:
namespace InfoRep1
{
     enum { value = 1 };
     typedef char Type;
}
```

```
namespace InfoRep2
{
      enum { value = 2 };
      typedef int Type;
}

template <namespace Rep> class C
{
      Rep::Type t;
      int f(){    return Rep::value;    }
      Rep r; //error: Rep is a scope
};

C<InfoRep1> c1; C<InfoRep2> c2;
```

2.2. Advanced Cases

Indistinctive usage of classes, structures and namespaces as scopes:

```
//rewriting the previous example again:
namespace InfoRep1
{
      enum { value = 1 };
      typedef char Type;
}

struct InfoRep2
{
      enum { value = 2 };
      typedef int Type;
      int i;
      char c[20];
}

template <namespace Rep> class C
{
      Rep::Type t;
      int f(){    return Rep::value;    }
      Rep r; //error: Rep is a scope
      size_t g(){  return sizeof(Rep);   }     //error
};

template <namespace S1, namespace S1::S2, class
S1::S2::T> S1::S2::T function(S1::S2::T t);

C<InfoRep1> c1; C<InfoRep2> c2;
```

Despite InfoRep2 is a structure, it cannot be instantiated while it is used as a scope. Scopes can only be used as a left operand of the '::' operator.

3. Interactions and implementability

3.1. Interactions:

a) Both namespaces, classes and structures may be passed as scope template parameters.

b) Scope template parameters shall not be treated as types within the template definition.

c) Scope template parameters will not be able to be re-opened within the template definition; scope template parameters are treated as closed-entities

[*Example*

```
template <namespace Rep> C
{
      namespace Rep { typedef char yy; } //error
}
```

–*end example*]

d) Scopes and types shall be able to be specified as template parameters belonging to a previous scope parameter (as shown in the 'function' example above)

e) Scope template parameters can also have default scopes – classes, namespaces, structures or another scope parameter of an outer template definition

[*Example*

```
template <namespace S1> struct Outer
{
        template <namespace S2 = S1> struct Inner
        {
              S2::Type t;
        };
};
```

–*end example*]

3.2 Implementability

- Considering that a namespace is an open entity, only contained definitions present in the current compilation unit namespace will be able to be specified.

[Worst case] The following syntactical situations are behaviorally equivalent:

Situation A: using a scope template parameter

compilation unit **U1** contains:

1) a namespace N:

Namespace N contains a symbol definition S (i.e. a structure), with definition D1.

2) a template definition T accepting a *scope* template parmeter P:

T acceses a member of P named S.

3) a global instance 'I' of T instantiated with N::S.

compilation unit **U2** contains:

1) a namespace N: (same name as U1)

Namespace N contains a symbol definition S, with definition D2 (D1 ≠ D2);

2) the template definition T (same U1's definition).

3) an import of the U1's 'I' global instance;

4) a function F that uses 'I'

Situation B: using a type template parameter

compilation unit **U1** contains:

1) a symbol definition S (i.e. a structure), with definition D1.

2) a template definition T accepting a *type* template parmeter P:

T acceses P.

3) a global instance 'I' of T instantiated with S.

compilation unit **U2** contains:

1) a symbol definition S, with definition D2 (D1 ≠ D2);

2) the template definition T (same U1's definition).

3) an import of the U1's 'I' global instance;

4) a function F that uses 'I'

Observe that Situation B does not contain scope template parameters and may be generated with current C++ syntax. (situation A is same as situation B plus the grayed texts, specific to this paper definitions).

Also observe that F uses S with definition D2, while 'I' was instantiated with S defined as D1.

- This paper leaves an open syntax issue regarding how to specify the global namespace both for template parameter specification and default template parameter value. The `::''` syntax is suggested (scope operator followed by two consecutive single quotes):

```
template <namespace NS = ::''> class X;
template <namespace NS> class Y{};  Y<::''> y;
```

4. Future work

This paper provides the basis for 'template namespaces', which needs a rigorous analysis and the experience that may emerge from this proposal.