Namespace Pervasiveness & Modules

Nathan Sidwell

This paper discusses how namespace names might implicitly pervade the program. While the modules-its claims not to make changes to how namespaces may interact, modules, at the very least can lead, to some unexpected behaviours. Those might or might not be desirable.

1 Background

P714r0 was presented at Toronto'17, and subsequent discussion showed that some important cases had been ignored. This paper reformulates the examples anew.

2 Non-Module Existing Behaviour

The current standard defines consistency across translation units as follows:

[6.5,basic.link]/9 Two names that are the same (Clause 6) and that are declared in different scopes shall denote the same variable, function, type, template or namespace if …

[6.5,basic.link]/10 After all adjustments of types (during which typedefs (10.1.3) are replaced by their definitions), the types specified by all declarations referring to a given variable or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (11.3.4). A violation of this rule on type identity does not require a diagnostic.

Note that there is no requirement for the same name to refer to the same kind of entity – only that if they do, the entities must be consistent.

Programs are often able to define different kinds of entities with the same name¹ in different translation units. That is due to name mangling being a common implementation to represent nested names and function overload sets at the object file level. Further, in the case of namespaces and named types, there is no object-file symbol for the namespace or type itself – such entities only appear as part of the name of some other entity. Thus traditional linker technology never encounters a duplicate symbol.

¹ Unless otherwise specified, this paper uses the shorthand ‘name’ to refer to the more general concept of a name in a specific scope, or qualified-id.
The following two translation units behave satisfactorily, because of the above implementation scheme, in a single program even though formal language semantics make it ill-formed:

// Translation unit #A part of library #1
namespace MyInternal {
    // stuff
}
and

// Translation unit #B part of library #2
void MyInternal (void);

While it might be undesirable for the above example to work, it is undeniable that the the following must be compatible with either of the above two:

// Translation unit #C, part of library #3
static void MyInternal (void);

As the function declaration in TU #C has internal linkage, it is distinct from the declarations of the same name in TUs #A and #B.2

3 Exporting Namespaces

The modules-ts specifies that namespaces with external linkage are always exported:

[10.3,basic.namespace]/1 A namespace with external linkage is always exported regardless of whether any of its namespace-definition is introduced by export.

It also specifies that non-exported imports are not transitive:3

[10.7.3,dcl.module.export]/1 An exported module-import-declaration nominating a module $M'$ in the purview of a module $M$ makes all exported names of $M'$ visible to any translation unit importing $M$. [ Note: A module interface unit (for a module $M$) containing an import-declaration does not make the imported names transitively visible to translation units importing the module $M$. — end note ]

Note that the first quoted paragraph refers to a ‘namespace’ not a namespace-definition. These two requirements have not immediately obvious outcomes. Consider:

// Translation unit #D
export module D;
namespace N {
    export class X {...}; // #D1

2 Implementations will still typically give it the same mangled name as that for translation unit #B’s declaration, but the object-file symbol will have local, rather than global, visibility.
3 Paper p0731 suggests clarifying edits to this paragraph. The interpretation used in this paper is that documented in p0731.
export int Baz (X const &); // #D2
}
// Translation unit #E
export module E;
import D;
export N::X Frob (); // #E1

Note that the declaration of Frob at #E1 uses a type that is not itself reexported from module E. This does not violate the requirements of:

[10.7.1,module.dcl.interface]/2 If that declaration introduces an entity with a non-dependent type, then that type shall have external linkage or shall involve only types with external linkage

I shall use Frob in a later example.

Is namespace N (but not its contents) implicitly reexported by module E or not? (Answer, it is not).

3.1  No Implicit Reexport

Not implicitly reexporting namespaces encountered solely from imported modules is consistent with [dcl.module.export]/1 (but contradicts [basic.namespace]/1). It will permit the following import of module E:

// Translation unit #F
import E;
static int N (); // #F1

As namespace N is not exported by E, the declaration of N at #F1 is well formed. As it has internal linkage, there is no conflict combining translation units #D, #E & #F in a single program.

However, the namespace N must still exist within the compilation of translation unit #F. Depending on module E’s other exports it might become visible via types used in ADL or type inference. For instance:

// Translation unit #G
import E;
dcltype (Frob()) x; // #G1 type N::X@D
int y = Baz (x); // #G2 N::Baz@D found by ADL

We have created a variable, x, using dcltype to get hold of N::X. Then used ADL to search the partitions of N visible to N::X defined in module D. While perhaps surprising, this is not completely new behaviour. An analagous situation occurs with private access of class members:
class Outer {
    class Inner { }
    public:
        static Inner Frob ();
};
decltype (Outer::Frob ()) x2;

Although Outer::Inner is a private member, it is exposed in the return type of public member Outer::Frob. The same decltype trick can be used again access to it.

Due to the typical linker and mangling implementations described above, there should be no confusion at the binary file level between namespace N and other entities named N.

This non-reexport behaviour is most similar to non-modules source bases.

3.2 Implicit Reexport

Implicitly reexporting namespaces is an implication of the literal wording of [basic.namespace]/1, but of course contradicts [dcl.module.export]/1. The above example would become ill-formed, with the declaration of N at #F1 conflicting with the namespace implicitly brought in by importing module E.

Such reexport would still require the decltype trick of translation unit #G to get at the contents of module D’s namespace partition of N. It would make the translation unit #E above ill-formed, which is undesirable.

4 Proposal

The modules-ts should be clear that:

- There is no implicit re-export of a namespace.

This is the original design intent, and editorial changes have altered the design in an unintended manner.

4.1 Changes to basic.namespace [10.3]

Alter the new wording appended to [basic.namespace]/1 as follows:

... A namespace with external linkage introduced by a namespace-definition is always exported regardless of whether any of its namespace-definitions is introduced by export.  
[ Note: There is no way to define a namespace with module linkage. — end note ] ...

This change makes it clear that it is namespace-definitions that create exported namespaces, and that the export is just as any other exported declaration.
5 Acknowledgements

I thank Gaby dos Reis <gdr@microsoft.com> and Gor Nishanov <gorn@microsoft.com> for drafting review.