Names and Scoping

Slice has a number of rules regarding identifiers. You will typically not have to concern yourself with these. However, occasionally, it is good to know how Slice uses naming scopes and resolves identifiers.

On this page:

Naming Scope

The following Slice constructs establish a naming scope:

- the global (file) scope
- modules
- interfaces
- classes
- structures
- exceptions
- parameter lists

Within a naming scope, identifiers must be unique, that is, you cannot use the same identifier for different purposes. For example:

Slice

```
interface Bad {
    void op(int p, string p);   // Error!
};
```

Because a parameter list forms a naming scope, it is illegal to use the same identifier p for different parameters. Similarly, data members, operation names, interface and class names, etc. must be unique within their enclosing scope.

Case Sensitivity

Identifiers that differ only in case are considered identical, so you must use identifiers that differ not only in capitalization within a naming scope. For example:

Slice

```
struct Bad {
    int    m;
    string M;   // Error!
};
```

The Slice compiler also enforces consistent capitalization for identifiers. Once you have defined an identifier, you must use the same capitalization for that identifier thereafter. For example, the following is in error:

Slice

```
sequence<string> StringSeq;

interface Bad {
    stringSeq op();     // Error!
};
```

Note that identifiers must not differ from a Slice keyword in case only. For example, the following is in error:

Slice

```
interface Module {      // Error, "module" is a keyword
    // ...
};
```

Qualified Names

The scope-qualification operator `::` allows you to refer to a type in a non-local scope. For example:

Slice

```
module Types {
    sequence<long> LongSeq;
};

module MyApp {
    sequence<Types::LongSeq> NumberTree;
};
```

Here, the qualified name `Types::LongSeq` refers to `LongSeq` defined in module `Types`. The global scope is denoted by a leading `::`, so we could also refer to `LongSeq` as `::Types::LongSeq`.

The scope-qualification operator also allows you to create mutually dependent interfaces that are defined in different modules. The obvious attempt to do this fails:

Slice

```
module Parents {
    interface Children::Child;  // Syntax error!
    interface Mother {
        Children::Child* getChild();
    };
    interface Father {
        Children::Child* getChild();
    };
};

module Children {
    interface Child {
        Parents::Mother* getMother();
        Parents::Father* getFather();
    };
};
```

This fails because it is syntactically illegal to forward-declare an interface in a different module. To make it work, we must use a reopened module:

Slice

```
module Children {
    interface Child;                    // Forward declaration
};

module Parents {
    interface Mother {
        Children::Child* getChild();    // OK
    };
    interface Father {
        Children::Child* getChild();    // OK
    };
};

module Children {                       // Reopen module
    interface Child {                   // Define Child
        Parents::Mother* getMother();
        Parents::Father* getFather();
    };
};
```

While this technique works, it is probably of dubious value: mutually dependent interfaces are, by definition, tightly coupled. On the other hand, modules are meant to be used to place related definitions into the same module, and unrelated definitions into different modules. Of course, this begs the question: if the interfaces are so closely related that they depend on each other, why are they defined in different modules? In the interest of clarity, you probably should avoid this construct, even though it is legal.

Names in Nested Scopes

Names defined in an enclosing scope can be redefined in an inner scope. For example, the following is legal:

Slice

```
module Outer {
    sequence<string> Seq;

    module Inner {
        sequence<short> Seq;
    };
};
```

Within module `Inner`, the name `Seq` refers to a sequence of `short` values and hides the definition of `Outer::Seq`. You can still refer to the other definition by using explicit scope qualification, for example:

Slice

```
module Outer {
    sequence<string> Seq;

    module Inner {
        sequence<short> Seq;

        struct Confusing {
            Seq          a;     // Sequence of short
            ::Outer::Seq b;     // Sequence of string
        };
    };
};
```

Needless to say, you should try to avoid such redefinitions — they make it harder for the reader to follow the meaning of a specification.

Same-named constructs cannot be nested inside each other. For example, a module named `M` cannot (recursively) contain any construct also named `M`. The same is true for interfaces, classes, structures, exceptions, and operations. For example, the following examples are all in error:

Slice

```
module M {
    interface M { /* ... */ };  // Error!

    interface I {
        void I();               // Error!
        void op(string op);     // Error!
    };

    struct S {
        long s;                 // Error, even if case differs!
    };

};

module Outer {
    module Inner {
        interface Outer {       // Error!
            // ...
        };
    };
};
```

The reason for this restriction is that nested types that have the same name are difficult to map into some languages. For example, C++ and Java reserve the name of a class as the name of the constructor, so an interface `I` could not contain an operation named `I` without artificial rules to avoid the name clash.

Similarly, some languages (such as C# prior to version 2.0) do not permit a qualified name to be anchored at the global scope. If a nested module or type is permitted to have the same name as the name of an enclosing module, it can become impossible to generate legal code in some cases.

In the interest of simplicity, Slice prohibits the name of a nested module or type to be the same as the name of one of its enclosing modules.

Introduced Identifiers

Within a naming scope, an identifier is introduced at the point of first use; thereafter, within that naming scope, the identifier cannot change meaning.

For example:

Slice

```
module M {
    sequence<string> Seq;

    interface Bad {
        Seq op1();      // Seq and op1 introduced here
        int Seq();      // Error, Seq has changed meaning
    };
};
```

The declaration of `op1` uses `Seq` as its return type, thereby introducing `Seq` into the scope of interface `Bad`. Thereafter, `Seq` can only be used as a type name that denotes a sequence of strings, so the compiler flags the declaration of the second operation as an error.

Note that fully-qualified identifiers are not introduced into the current scope:

Slice

```
module M {
    sequence<string> Seq;

    interface Bad {
        ::M::Seq op1(); // Only op1 introduced here
        int Seq();      // OK
    };
};
```

In general, a fully-qualified name (one that is anchored at the global scope and, therefore, begins with a `::` scope resolution operator) does not introduce any name into the current scope. On the other hand, a qualified name that is not anchored at the global scope introduces only the first component of the name:

Slice

```
module M {
    sequence<string> Seq;

    interface Bad {
        M::Seq op1();   // M and op1 introduced here, but not Seq
        int Seq();      // OK
    };
};
```

Name Lookup Rules

When searching for the definition of a name that is not anchored at the global scope, the compiler first searches backward in the current scope of a definition of the name. If it can find the name in the current scope, it uses that definition. Otherwise, the compiler successively searches enclosing scopes for the name until it reaches the global scope. Here is an example to illustrate this:

Slice

```
module M1 {
    sequence<double> Seq;

    module M2 {
        sequence<string> Seq;   // OK, hides ::M1::Seq

        interface Base {
            Seq op1();          // Returns sequence of string
        };
    };

    module M3 {
        interface Derived extends M2::Base {
            Seq op2();          // Returns sequence of double
        };

        sequence<bool> Seq;     // OK, hides ::M1::Seq

        interface I {
            Seq op();           // Returns sequence of bool
        };
    };

    interface I {
        Seq op();               // Returns sequence of double
    };
};
```

Note that `M2::Derived::op2` returns a sequence of `double`, even though `M1::Base::op1` returns a sequence of `string`. That is, the meaning of a type in a base interface is irrelevant to determining its meaning in a derived interface — the compiler always searches for a definition only in the current scope and enclosing scopes, and never takes the meaning of a name from a base interface or class.

See Also

- Lexical Rules