Slice Metadata Directives

Previous

Next

On this page:

General Metadata Directives

`amd`

This directive applies to interfaces, classes, and individual operations. It enables code generation for asynchronous method dispatch. (See the relevant language mapping chapter for details.)

`delegate`

This directive applies only to local interfaces with one operation. Interfaces with this metadata will be generated as a `std::function` in C++, a `delegate` in C#, and an interface with one operation (usable as a `FunctionalInterface` for Java `8)` in Java.

`deprecate`

This directive allows you to emit a deprecation warning for Slice constructs.

`ice-prefix`

This global directive allows the use of identifiers that start with the reserved prefix `Ice` (or `ICE`, `ice` etc.). Only Slice files provided by Ice should use this directive.

`format`

This directive defines the encoding format used for any classes or exceptions marshaled as the arguments or results of an operation. The tag can be applied to an interface, which affects all of its operations, or to individual operations. Legal values for the tag are `format:sliced`, `format:compact`, and `format:default`. A tag specified for an operation overrides any setting applied to its enclosing interface. The `Ice.Default.SlicedFormat` property defines the behavior when no tag is present.

`marshaled-result`

This directive changes the return type of servant methods so that a servant can force its results to be marshaled immediately in a thread-safe manner. Refer to the relevant server-side language mapping sections for more information on the rules for parameter passing.

`preserve-slice`

This directive applies to classes and exceptions, allowing an intermediary to forward an instance of the annotated type, or any of its subtypes, with all of its slices intact. Operations that transfer such types must be annotated with `format:sliced`. It is not necessary to repeat the `preserve-slice` tag on derived types, but you may wish to do so for documentation purposes.

`protected`

This directive applies to data members of classes and changes code generation to make these members protected. See class mapping of the relevant language mapping chapter for more information.

`suppress-warning`

This global directive allows to suppress Slice compiler warnings. It applies to all definitions in the Slice file that includes this directive. If one or more categories are specified (for example "`suppress-warning:invalid-metadata" or "suppress-warning:deprecated, invalid-metadata`") only warnings matching these categories will be suppressed, otherwise all warnings are suppressed. The categories are described in the following table:

Suppress Warning Category	Description
`all`	Suppress all Slice compiler warnings. Equivalent to `[["suppress-warning"]]`.
`deprecated`	Suppress warnings related to deprecated features.
`invalid-metadata`	Suppress warnings related to invalid metadata.

`underscore`

This global directive allows the use of identifiers with underscores. It applies to all definitions in the Slice file that includes this directive.

Back to Top ^

Metadata Directives for C++

The Slice to C++ compiler understands three C++ metadata prefixes: `cpp`, `cpp11` and `cpp98`. Most C++ metadata directives can be specified with either one of the prefixes.

`cpp11` and `cpp98` directives applies only to their respective mapping. A `cpp` metadata directive applies to both mappings, and can be overridden by the same directive with `cpp98` or `cpp11`. For example:

Slice

```
["cpp:type:MyType", "cpp11:type:MyNewType"] sequence<byte> ByteSeq;
```

maps `ByteSeq` to `MyType` with the C++98 mapping, and to `MyNewType` with the C++11 mapping.

`cpp:array`

This directive applies to sequence parameters in operations. It directs the code generator to map these parameters to pairs of pointers.

`cpp:class`

This directive applies to structures. It directs the code generator to create a C++ class (instead of a C++ structure) to represent a Slice structure. This is a C++98-only metadata: it has no effect with the C++11 mapping.

`cpp:comparable`

This directive applies to structures. It directs the code generator to generate comparison operators for a structure regardless of whether it qualifies as a legal dictionary key type. This is a C++98-only metadata: it has no effect with the C++11 mapping.

`cpp:const`

This directive applies to operations. It directs the code generator to create a `const` pure virtual member function for the skeleton class.

`cpp:dll-export:SYMBOL`

This global directive applies to all definitions in a Slice file.

Use *`SYMBOL`* to control the export and import of symbols from DLLs on Windows and dynamic shared libraries on other platforms. This option allows you to export symbols from the generated code, and place such generated code in a DLL (on Windows) or shared library (on other platforms). As an example, compiling a Slice file `Widget.ice` with:

Slice

```
[["cpp:dll-export:WIDGET_API"]]
```

results in the following additional code being generated into `Widget.h`:

C++

```
#ifndef WIDGET_API
#   if defined(ICE_STATIC_LIBS)
#       define WIDGET_API /**/
#   ifdef WIDGET_API_EXPORTS
#       define WIDGET_API ICE_DECLSPEC_EXPORT
#   else
#       define WIDGET_API ICE_DECLSPEC_IMPORT
#   endif
#endif
```

The generated code also includes the provided `SYMBOL` name (`WIDGET_API` in our example) in the declaration of classes and functions that need to be exported (when building a DLL or dynamic library) or imported (when using such library).

`ICE_DECLSPEC_EXPORT` and `ICE_DECLSPEC_IMPORT` are macros that expand to compiler-specific attributes. For example, for Visual Studio, they are defined as:

C++

```
#if defined(_MSC_VER)
#   define ICE_DECLSPEC_EXPORT __declspec(dllexport)
#   define ICE_DECLSPEC_IMPORT __declspec(dllimport)
```

With GCC and clang, they are defined as:

C++

```
#elif defined(__GNUC__) || defined(__clang__)
#   define ICE_DECLSPEC_EXPORT __attribute__((visibility ("default")))
#   define ICE_DECLSPEC_IMPORT __attribute__((visibility ("default")))
```

The generated .cpp file (`Widget.cpp` in our example) defines `SYMBOL_EXPORTS`; this way, you don't need to do anything special when compiling generated files.

`cpp:header-ext`

This global directive allows you to use a file extension for C++ header files other than the default `.h` extension.

`cpp:ice_print`

This directive applies to exceptions. It directs the code generator to declare (but not implement) an `ice_print` member function that overrides the `ice_print` virtual function inherited from an Ice base class. The application must provide the implementation of this `ice_print` function.

`cpp:include`

This global directive allows you inject additional `#include` directives into the generated code. This is useful for custom types.

`cpp:noexcept`

This directive applies only to operations on local interfaces. When specified, the generated C++ pure virtual function declaration will carry the `noexcept` specifier (C++11) or the `ICE_NOEXCEPT macro` (C++98). ICE_NOEXCEPT expands to `noexcept` or `throw()` depending the C++ compiler and C++ compilation flags.

`cpp:range`

This directive applies to sequence parameters in operations. It directs the code generator to map these parameters to pairs of iterators. This is a C++98-only metadata direcetive: it has no effect with the C++11 mapping.

`cpp:scoped`

This directive applies to enumerations. It directs the code generator to use the enumeration's name as prefix for all generated C++ enumerators. This is a C++98-only metadata directive: it has no effect on the C++11 mapping. See also `cpp:unscoped` below.

`cpp:source-ext`

This global directive allows you to use a file extension for C++ source files other than the default `.cpp` extension.

`cpp:type:`*`c++-type`*

This directive applies to sequences and dictionaries. It directs the code generator to map the Slice type or parameter to the provided C++ type.

`cpp:type:string` and `cpp:type:wstring`

These directives apply to data members of type string as well as to containers, such as structures, classes, exceptions, and modules. String members map by default to `std::string`. You can use the `cpp:type:wstring` metadata to cause a string data member (or all string data members in a structure, class or exception) to map to `std::wstring` instead. Use the `cpp:type:string` metadata to force string members to use the default mapping regardless of any enclosing metadata.

Slice

```
["cpp:type:wstring"]
module A // All string members in this module map by default to std::wstring
{
    struct Struct1
    {
        string s; // Maps to std::wstring
    }
    struct Struct2
    {
        ["cpp:type:string"] string s; // Maps to std::string
    }

    ["cpp:type:string"] // All string members in this module map by default to std::string
    module Inner
    {
        struct Struct4
        {
            string s; // Maps to std::string
        }

        ["cpp:type:wstring"] // All string members of Struct4 map by default to std::wstring
        struct Struct3
        {
            string s; // Maps to std::wstring
        }
    }
}
```

`cpp:unscoped`

This directive applies to enumerations. It directs the code generator to create an old-type unscoped C++ enumeration instead of a scoped enumeration (`enum class`). This is a C++11-only directive: it has no effect on the C++98 mapping. See also `cpp:scoped` above.

`cpp:view-type:`***`c++-view-type`***

This directive applies to string, sequence and dictionary parameters. It directs the code generator to map this parameter to the provided C++ type when this parameter does not need to hold any memory, for example when mapping an in-parameter to a proxy function.

`cpp:virtual`

This directive applies to classes with the C++98 mapping only. It has no effect with the C++11 mapping. If the directive is present and a class has base classes, the generated C++ class derives virtually from its bases; without this directive, slice2cpp generates the class so it derives non-virtually from its bases.

This directive is useful if you use Slice classes as servants and want to inherit the implementation of operations in the base class in the derived class. For example:

Slice

```
class Base
{
    int baseOp();
}

["cpp:virtual"]
class Derived extends Base
{
    string derivedOp();
}
```

The metadata directive causes slice2cpp to generate the class definition for `Derived` using virtual inheritance:

C++98

```
class Base : public virtual Ice::Object
{
    // ...
};

class Derived : public virtual Base
{
    // ...
};
```

This allows you to reuse the implementation of `baseOp` in the servant for `Derived` using ladder inheritance:

C++98

```
class BaseI : public virtual Base
{
    Ice::Int baseOp(const Ice::Current&);
    // ...
};

class DerivedI : public virtual Derived, public virtual BaseI
{
    // Re-use inherited baseOp()
};
```

Note that, if you have data member in classes and use virtual inheritance, you need to take care to correctly call base class constructors if you implement your own one-shot constructor. For example:

Slice

```
class Base
{
    int baseInt;
}

class Derived extends Base
{
    int derivedInt;
}
```

The generated one-shot constructor for `Derived` initializes both `baseInt` and `derivedInt`:

C++98

```
Derived::Derived(Ice::Int iceP_baseInt, Ice::Int iceP_derivedInt) :
    M::Base(iceP_baseInt),
    derivedInt(iceP_derivedInt)
{
}
```

If you derive your own class from `Derived` and add a one-shot constructor to your class, you must explicitly call the constructor of all the base classes, including `Base`. Failure to call the `Base` constructor will result in `Base` being default-constructed instead of getting a defined value. For example:

C++98

```
class DerivedI : public virtual Derived
{
public:
    DerivedI(int baseInt, int derivedInt, const string& s) :
        Base(baseInt), Derived(baseInt, derivedInt), _s(s)
    {
    }

private:
    string _s;
};
```

This code correctly initializes the `baseInt` member of the `Base` part of the class. Note that the following does not work as intended and leaves the `Base` part default-constructed (meaning that `baseInt` is not initialized):

C++98

```
class DerivedI : public virtual Derived
{
public:
    DerivedI(int baseInt, int derivedInt, const string& s) :
        Derived(baseInt, derivedInt), _s(s)
    {
        // WRONG: Base::baseInt is not initialized.
    }

private:
    string _s;
};
```

Back to Top ^

Metadata Directives for C#

The metadata for C# (or .NET) directives uses the `cs` prefix or the `clr` prefix. These two prefixes are interchangeable: `cs:attribute` and `clr:attribute` have exactly the same meaning. We present these directives with the `cs` prefix below.

In Ice releases prior to 3.7, the `cs` prefix was used exclusively for the metadata directives `cs:attribute` and `cs:tie` while the `clr` prefix was used for all other directives.

`cs:attribute`

This directive can be used both as a global directive and as directive for specific Slice constructs. It injects C# attribute definitions into the generated code. (See C-Sharp Attribute Metadata Directive.)

`cs:class`

This directive applies to Slice structures. It directs the code generator to emit a C# class instead of a structure.

`cs:generic:List`, `cs:generic:LinkedList`, `cs:generic:Queue` and `cs:generic:Stack`

These directives apply to sequences and map them to the specified sequence type.

`cs:generic:SortedDictionary`

This directive applies to dictionaries and maps them to `SortedDictionary`.

`cs:generic`

This directive applies to sequences and allows you map them to custom types.

`cs:implements:`*`type`*

This directive adds the specified base type to the generated code for a Slice structure, class or interface. For example, Ice defines the `Communicator` interface as shown below:

Slice

```
["cs:implements:_System.IDisposable"]
local interface Communicator { ... }
```

Consequently, the generated C# interface `Ice.Communicator` implements `IDisposable`.

Every Slice-generated C# source file defines two namespace aliases:

```
using _System = global::System;
using _Microsoft = global::Microsoft;
```

We recommend using these aliases if your metadata refers to the `System` or `Microsoft` namespaces.

When used with structs, this metadata can only refer to interfaces without operations. With classes, the code is responsible for registering a value factory if the Slice class is transferred over-the-wire and uses this metadata to implement native C# interfaces.

`cs:property`

This directive applies to Slice structures and classes. It directs the code generator to create C# property definitions for data members.

`cs:serializable`

This directive allows you to use Ice to transmit serializable CLR classes as native objects, without having to define corresponding Slice definitions for these classes.

`cs:tie`

This directive applies to an interface or a class with operations, and triggers the generation of a tie class.

Back to Top ^

Metadata Directives for Java

`java:buffer`

This directive applies to sequences of certain primitive types. It directs the translator to map the sequence to a subclass of `java.nio.Buffer`.

`java:getset`

This directive applies to data members and structures, classes, and exceptions. It adds accessor and modifier methods (JavaBean methods) for data members.

`java:implements:`*`type`*

This directive adds the specified base interface to the generated code for a Slice structure, class or interface. For example, Ice defines the `Communicator` interface as shown below:

Slice

```
["java:implements:java.lang.AutoCloseable"]
local interface Communicator { ... }
```

Consequently, the generated Java interface `Communicator` implements `java.lang.AutoCloseable`.

When used with structs, this metadata can only refer to interfaces without operations. With classes or interfaces, the generated Java interface will be marked as `abstract`. The code is responsible for registering a value factory if the Slice class is transferred over-the-wire and uses this metadata to implement native Java interfaces.

`java:optional`

> This directive is only used by the Java Compat mapping and has no effect in the Java mapping.

This directive forces optional output parameters to use the optional mapping instead of the default required mapping in servants.

`java:package`

This global directive instructs the code generator to place the generated classes into a specific package.

`java:serializable`

This directive allows you to use Ice to transmit serializable Java classes as native objects, without having to define corresponding Slice definitions for these classes.

`java:serialVersionUID`

This directive overrides the default (generated) value of `serialVersionUID` for a Slice type.

`java:tie`

> This directive is only used by the Java Compat mapping and has no effect in the Java mapping.

This directive applies to an interface or a class with operations, and triggers the generation of a tie class.

`java:type`

This directive allows you to use custom types for sequences and dictionaries.

`java:UserException`

This directive applies to operations, and indicates that the generated Java methods on the mapped servant interfaces and class can throw any user exception, regardless of its specific definition. The exception specification for these methods is simply `throws com.zeroc.Ice.UserException` (Java) or `throws Ice.UserException` (Java Compat). This metadata has no effect on the methods of generated proxies. The directive `UserExceptio` (without the `java:` prefix) is a deprecated alias for `java:UserException`.

Back to Top ^

Metadata Directives for Objective-C

`objc:dll-export:SYMBOL`

This global directive applies to all definitions in a Slice file.

Use *`SYMBOL`* to control the export of symbols from dynamic shared libraries. This option allows you to export symbols from the generated code and place such generated code in a shared library. Compiling a Slice definition with:

Slice

```
[["objc:dll-export:WIDGET_API"]]
```

adds the provided `SYMBOL` name (`WIDGET_API` in our example) to the declaration of interfaces and protocols that need to be exported. The generated code also defines the provided SYMBOL as `__attribute__((visibility ("default")))`.

> This option is useful when you create a shared library and compile your Objective-C code with `-fvisibility=hidden` to reduce the number of symbols exported.

`objc:header-dir`

This global directive allows you to specify a prefix for the header path in generated Objective-C files. For example, all the Slice files from Ice set this metadata to `objc` to allow importing Objective-C headers from the `objc` directory (e.g.: `#import <objc/Ice/Ice.h>`).

`objc:prefix`

This directive applies to modules and changes the default mapping for modules to use a specified prefix.

`objc:scoped`

This directive applies to enumerations. It directs the code generator to use the enumeration's name as prefix for all generated Objective-C enumerators.

Back to Top ^

Metadata Directives for Python

`python:package`

This global directive instructs the code generator to enclose the generated code in a specified Python package.

`python:pkgdir`

This global directive instructs the code generator to place the generated code into a specified directory.

`python:seq:default`, `python:seq:list` and `python:seq:tuple`

These directives allow you to change the mapping for Slice sequences.

Back to Top ^

Metadata Directives for Freeze

`freeze:read` and `freeze:write`

These directives inform a Freeze evictor whether an operation updates the state of an object, so the evictor knows whether it must save an object before evicting it.

Back to Top ^

See Also

- Metadata

Previous

Next