

Slice Metadata Directives

On this page:

- [General Metadata Directives](#)
- [Metadata Directives for C++](#)
- [Metadata Directives for Java](#)
- [Metadata Directives for C#](#)
- [Metadata Directives for .NET and Mono](#)
- [Metadata Directives for Objective-C](#)
- [Metadata Directives for Python](#)
- [Metadata Directives for Freeze](#)

General Metadata Directives

ami

This directive applies to interfaces, classes, and individual operations. It enable code generation for asynchronous method invocation.



This directive applies to the [deprecated AMI mapping](#). For the new AMI mapping there is no need for this directive.

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.)

deprecated

This directive allows you to emit a [deprecation warning for Slice constructs](#).

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.

UserException

This directive applies only to operations on local interfaces. The metadata directive indicates that the operation can throw any user exception, regardless of its specific definition. (This directive is used for the `locate` and `finished` operations on servant locators, which can throw any user exception.)

Metadata Directives for C++

cpp:array and cpp:range:array

These directives apply to sequences. They direct the code generator to create zerocopy APIs for [passing sequences as parameters](#).

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.

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:type:wstring

This directive applies to data members of type `string` as well as to containers, such as structures, classes, and exceptions. It changes the [default mapping for strings](#) from `std::string` to `std::wstring`.

cpp:header-ext

This global directive allows you to use a [file extension for C++ header files](#) other than the default .h extension.

cpp:include

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

cpp:virtual

This directive applies to classes. 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++

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

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

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

C++

```
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++

```
Derived::Derived(Ice::Int __ice_baseInt, Ice::Int __ice_derivedInt)
    : M::Base(__ice_baseInt),
      derivedInt(__ice_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++

```
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++

```
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;
};
```

Metadata Directives for Java

java:package

This global directive instructs the code generator to place the generated classes into a [specific package](#).

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: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:type

This directive allows to use [custom types](#) for sequences and dictionaries.

Metadata Directives for C#

Note that C# (and other Common Language Runtime languages) are also affected by metadata with a `clr:` prefix. (See [#Metadata Directives for .NET and Mono](#).)

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 Specific Metadata Directives](#).)

Metadata Directives for .NET and Mono

clr:class

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

clr:collection

This directive applies to [sequences](#) and [dictionaries](#) and maps them to types that are derived from `CollectionBase` and `DictionaryBase`, respectively.

clr:generic:List, clr:generic:LinkedList, clr:generic:Queue and clr:generic:Stack

These directives apply to [sequences](#) and map them to the specified sequence type.

clr:generic:SortedDictionary

This directive applies to [dictionaries](#) and maps them to `SortedDictionary`.

clr:generic

This directive applies to [sequences](#) and allows you map them to custom types.

clr:property

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

clr: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.

Metadata Directives for Objective-C

objc:prefix

This directive applies to modules and changes the [default mapping for modules](#) to use a specified prefix.

Metadata Directives for Python

python:package

This global directive instructs the code generator to place the generated code into a [specified Python package](#).

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

These directives allow you to change the [mapping for Slice sequences](#).

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.

See Also

- [Metadata](#)