C-Sharp Mapping for Structures

Ice for .NET supports two different mappings for Slice structures. By default, Slice structures map to C# structures if they (recursively) contain only value types. If a Slice structure (recursively) contains a string, proxy, class, sequence, or dictionary member, it maps to a C# class. A metadata directive allows you to force the mapping to a C# class for Slice structures that contain only value types.

In addition, for either mapping, you can control whether Slice data members are mapped to fields or to properties.

On this page:

- Structure Mapping for Structures in C#
- Class Mapping for Structures in C#
- Property Mapping for Structures in C#

Structure Mapping for Structures in C#

Consider the following structure:

Slice

```
struct Point {
    double x;
    double y;
};
```

This structure consists of only value types and so, by default, maps to a C# partial structure:

C#

```
public partial struct Point
{
    public double x;
    public double y;

    public Point(double x, double y);

    public override int GetHashCode();
    public override bool Equals(object other);

    public static bool operator==(Point lhs, Point rhs);
    public static bool operator!=(Point lhs, Point rhs);
}
```

For each data member in the Slice definition, the C# structure contains a corresponding public data member of the same name.

The generated constructor accepts one argument for each structure member, in the order in which they are defined in the Slice definition. This allows you to construct and initialize a structure in a single statement:

C#

```
Point p = new Point(5.1, 7.8);
```

Note that C# does not allow a value type to declare a default constructor or to assign default values to data members.

The structure overrides the `GetHashCode` and `Equals` methods to allow you to use it as the key type of a dictionary. (Note that the static two-argument version of `Equals` is inherited from `System.Object`.) Two structures are equal if (recursively) all their data members are equal. Otherwise, they are not equal. For structures that contain reference types, `Equals` performs a deep comparison; that is, reference types are compared for value equality, not reference equality.

Copyright © 2017, ZeroC, Inc.

Class Mapping for Structures in C#

The mapping for Slice structures to C# structures provides value semantics. Usually, this is appropriate, but there are situations where you may want to change this:

- If you use structures as members of a collection, each access to an element of the collection incurs the cost of boxing or unboxing. Depending on your situation, the performance penalty may be noticeable.
- On occasion, it is useful to be able to assign null to a structure, for example, to support "not there" semantics (such as when implementing parameters that are conceptually optional).

To allow you to choose the correct performance and functionality trade-off, the Slice-to-C# compiler provides an alternative mapping of structures to classes, for example:

Slice

```
["clr:class"] struct Point {
    double x;
    double y;
};
```

The `"clr:class"` metadata directive instructs the Slice-to-C# compiler to generate a mapping to a C# partial class for this structure. The generated code is almost identical, except that the keyword `struct` is replaced by the keyword `class` and that the class has a default constructor and inherits from `ICloneable`:

C#

```
public partial class Point : _System.ICloneable
{
    public double x;
    public double y;

    public Point();
    public Point(double x, double y);

    public object Clone();

    public override int GetHashCode();
    public override bool Equals(object other);

    public static bool operator==(Point lhs, Point rhs);
    public static bool operator!=(Point lhs, Point rhs);
}
```

Some of the generated marshaling code differs for the class mapping of structures, but this is irrelevant to application code.

The class has a default constructor that default-constructs each data member. This means members of primitive type are initialized to the equivalent of zero, and members of reference type are initialized to null. Note that applications must always explicitly initialize a member whose type is a class-mapped structure because the Ice run time does not accept null as a legal value for these types.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value.

The class also provides a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement:

C#

```
Point p = new Point(5.1, 7.8);
```

The `Clone` method performs a shallow memberwise copy, and the comparison methods have the usual semantics (they perform value comparison).

Copyright © 2017, ZeroC, Inc.

Note that you can influence the mapping for structures only at the point of definition of a structure, that is, for a particular structure type, you must decide whether you want to use the structure or the class mapping. (You cannot override the structure mapping elsewhere, for example, for individual structure members or operation parameters.)

As we mentioned previously, if a Slice structure (recursively) contains a member of reference type, it is automatically mapped to a C# class. (The compiler behaves as if you had explicitly specified the `"clr:class"` metadata directive for the structure.)

Here is our Employee structure once more:

Slice

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The structure contains two strings, which are reference types, so the Slice-to-C# compiler generates a C# class for this structure:

C#

```
public partial class Employee : _System.ICloneable
{
    public long number;
    public string firstName;
    public string lastName;

    public Employee();
    public Employee(long number, string firstName, string lastName);

    public object Clone();

    public override int GetHashCode();
    public override bool Equals(object other);

    public static bool operator==(Employee lhs, Employee rhs);
    public static bool operator!=(Employee lhs, Employee rhs);
}
```

Property Mapping for Structures in C#

You can instruct the compiler to emit property definitions instead of public data members. For example:

Slice

```
["clr:property"] struct Point {
    double x;
    double y;
};
```

The `"clr:property"` metadata directive causes the compiler to generate a property for each Slice data member:

Copyright © 2017, ZeroC, Inc.

C#

```
public partial struct Point
{
    private double x_prop;
    public double x {
        get {
            return x_prop;
        }
        set {
            x_prop = value;
        }
    }

    private double y_prop;
    public double y {
        get {
            return y_prop;
        }
        set {
            y_prop = value;
        }
    }

    // Other methods here...
}
```

Note that the properties are non-virtual because C# structures cannot have virtual properties. However, if you apply the `"clr:property"` directive to a structure that contains a member of reference type, or if you combine the `"clr:property"` and `"clr:class"` directives, the generated properties are virtual. For example:

Slice

```
["clr:property", "clr:class"]
struct Point {
    double x;
    double y;
};
```

This generates the following code:

Copyright © 2017, ZeroC, Inc.

C#

```
public partial class Point : System.ICloneable
{
    private double x_prop;
    public virtual double x {
        get {
            return x_prop;
        }
        set {
            x_prop = value;
        }
    }

    private double y_prop;
    public virtual double y {
        get {
            return y_prop;
        }
        set {
            y_prop = value;
        }
    }

    // Other methods here...
}
```

See Also

- Metadata
- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Sequences
- C-Sharp Mapping for Dictionaries
- C-Sharp Collection Comparison
- C-Sharp Mapping for Constants
- C-Sharp Mapping for Exceptions

Copyright © 2017, ZeroC, Inc.