C++ Mapping for Structures

A Slice structure maps to a C++ structure by default. In addition, you can use a metadata directive to map structures to C++ classes.

On this page:

Default Mapping for Structures in C++

Slice structures map to C++ structures with the same name. For each Slice data member, the C++ structure contains a public data member. For example, here is our Employee structure once more:

Slice

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

The Slice-to-C++ compiler generates the following definition for this structure:

C++

```
struct Employee {
    Ice::Long   number;
    std::string firstName;
    std::string lastName;
    bool operator==(const Employee&) const;
    bool operator!=(const Employee&) const;
    bool operator<(const Employee&) const;
    bool operator<=(const Employee&) const;
    bool operator>(const Employee&) const;
    bool operator>=(const Employee&) const;
};
```

For each data member in the Slice definition, the C++ structure contains a corresponding public data member of the same name. Constructors are intentionally omitted so that the C++ structure qualifies as a *plain old datatype* (POD).

Note that the structure also contains comparison operators. These operators have the following behavior:

- `operator==`
  Two structures are equal if (recursively), all its members are equal.
- `operator!=`
  Two structures are not equal if (recursively), one or more of its members are not equal.
- `operator<`
  `operator<=`
  `operator>`
  `operator>=`
  The comparison operators treat the members of a structure as sort order criteria: the first member is considered the first criterion, the second member the second criterion, and so on. Assuming that we have two `Employee` structures, `s1` and `s2`, this means that the generated code uses the following algorithm to compare `s1` and `s2`:

C++

```
bool Employee::operator<(const Employee& rhs) const
{
    if (this == &rhs)   // Short?cut self?comparison
        return false;

    // Compare first members
    //
    if (number < rhs.number)
        return true;
    else if (rhs.number < number)
        return false;

    // First members are equal, compare second members
    //
    if (firstName < rhs.firstName)
        return true;
    else if (rhs.firstName < firstName)
        return false;

    // Second members are equal, compare third members
    //
    if (lastName < rhs.lastName)
        return true;
    else if (rhs.lastName < lastName)
        return false;

    // All members are equal, so return false
    return false;
}
```

The comparison operators are provided to allow the use of structures as the key type of Slice dictionaries, which are mapped to `std::map` in C++.

Note that copy construction and assignment always have deep-copy semantics. You can freely assign structures or structure members to each other without having to worry about memory management. The following code fragment illustrates both comparison and deep-copy semantics:

C++

```
Employee e1, e2;
e1.firstName = "Bjarne";
e1.lastName = "Stroustrup";
e2 = e1;                        // Deep copy
assert(e1 == e2);
e2.firstName = "Andrew";        // Deep copy
e2.lastName = "Koenig";         // Deep copy
assert(e2 < e1);
```

Because strings are mapped to `std::string`, there are no memory management issues in this code and structure assignment and copying work as expected. (The default member-wise copy constructor and assignment operator generated by the C++ compiler do the right thing.)

Class Mapping for Structures in C++

Occasionally, the mapping of Slice structures to C++ structures can be inefficient. For example, you may need to pass structures around in your application, but want to avoid having to make expensive copies of the structures. (This overhead becomes noticeable for structures with many complex data members, such as sequences or strings.) Of course, you could pass the structures by const reference, but that can create its own share of problems, such as tracking the life time of the structures to avoid ending up with dangling references.

For this reason, you can enable an alternate mapping that maps Slice structures to C++ classes. Classes (as opposed to structures) are reference-counted. Because the Ice C++ mapping provides smart pointers for classes, you can keep references to a class instance in many places in the code without having to worry about either expensive copying or life time issues.

The alternate mapping is enabled by a metadata directive, `["cpp:class"]`. Here is our Employee structure once again, but this time with the additional metadata directive:

Slice

```
["cpp:class"] struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

Here is the generated class:

C++

```
class Employee : public IceUtil::Shared {
public:
    Employee() {}
    Employee(::Ice::Long,
             const ::std::string&,
             const ::std::string&);
    ::Ice::Long number;
    ::std::string firstName;
    ::std::string lastName;

    bool operator==(const Employee&) const;
    bool operator!=(const Employee&) const;
    bool operator<(const Employee&) const;
    bool operator<=(const Employee&) const;
    bool operator>(const Employee&) const;
    bool operator>=(const Employee&) const;
};
```

Note that the generated class, apart from a default constructor, has a constructor that accepts one argument for each member of the structure. This allows you to instantiate and initialize the class in a single statement (instead of having to first instantiate the class and then assign to its members).

As for the default structure mapping, the class contains one public data member for each data member of the corresponding Slice structure.

The comparison operators behave as for the default structure mapping.

You can learn how to instantiate classes, and how to access them via smart pointers, in the sections describing the mapping for Slice classes — the API described there applies equally to Slice structures that are mapped to classes.

Default Constructors for Structures in C++

Structures have an implicit default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built?in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

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

If you declare a default value for at least one member of a structure, or use the class mapping for the structure, the Slice compiler also generates a second constructor. This *one-shot* constructor has one parameter for each data member, allowing you to construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members).

See Also

- Structures
- C++ Mapping for Enumerations
- C++ Mapping for Sequences
- C++ Mapping for Dictionaries