Server-Side C++ Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing virtual functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Classes in C++

On the client side, interfaces map to proxy classes. On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has a pure virtual member function for each operation on the corresponding interface. For example, consider our Slice definition for the `Node` interface:

Slice

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The Slice compiler generates the following definition for this interface:

C++

```
namespace Filesystem {

    class Node : virtual public Ice::Object {
    public:
        virtual std::string name(const Ice::Current& = Ice::Current()) = 0;
        // ...
    };
    // ...
}
```

For the moment, we will ignore a number of other member functions of this class. The important points to note are:

- As for the client side, Slice modules are mapped to C++ namespaces with the same name, so the skeleton class definition is nested in the namespace `Filesystem`.
- The name of the skeleton class is the same as the name of the Slice interface (`Node`).
- The skeleton class contains a pure virtual member function for each operation in the Slice interface.
- The skeleton class is an abstract base class because its member functions are pure virtual.
- The skeleton class inherits from `Ice::Object` (which forms the root of the Ice object hierarchy).

Servant Classes in C++

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

Copyright © 2017, ZeroC, Inc.

C++

```
#include <Filesystem.h> // Slice-generated header

class NodeI : public virtual Filesystem::Node {
public:
    NodeI(const std::string&);
    virtual std::string name(const Ice::Current&);
private:
    std::string _name;
};
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.)

Note that `NodeI` inherits from `Filesystem::Node`, that is, it derives from its skeleton class. It is a good idea to always use virtual inheritance when defining servant classes. Strictly speaking, virtual inheritance is necessary only for servants that implement interfaces that use multiple inheritance; however, the `virtual` keyword does no harm and, if you add multiple inheritance to an interface hierarchy half-way through development, you do not have to go back and add a `virtual` keyword to all your servant classes.

As far as Ice is concerned, the `NodeI` class must implement only a single member function: the pure virtual `name` function that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. Obviously, the constructor initializes the `_name` member and the `name` function returns its value:

C++

```
NodeI::NodeI(const std::string& name) : _name(name)
{
}

std::string
NodeI::name(const Ice::Current&) const
{
    return _name;
}
```

Normal and `idempotent` Operations in C++

The `name` member function of the `NodeI` skeleton is not a `const` member function. However, given that the operation does not modify the state of its object, it really should be a `const` member function. We can achieve this by adding the `["cpp:const"]` metadata directive. For example:

Slice

```
interface Example {
                void normalOp();

    idempotent  void idempotentOp();

    ["cpp:const"]
    idempotent  void readonlyOp();
};
```

The skeleton class for this interface looks like this:

Copyright © 2017, ZeroC, Inc.

C++

```
class Example : virtual public Ice::Object {
public:
    virtual void normalOp(const Ice::Current& = Ice::Current()) = 0;
    virtual void idempotentOp(const Ice::Current& = Ice::Current()) = 0;
    virtual void readonlyOp(const Ice::Current& = Ice::Current()) const = 0;
    // ...
};
```

Note that `readonlyOp` is mapped as a `const` member function due to the `["cpp:const"]` metadata directive; normal and `idempotent` operations (without the metadata directive) are mapped as ordinary, non-`const` member functions.

See Also

- Slice for a Simple File System
- C++ Mapping for Interfaces
- Parameter Passing in C++
- Raising Exceptions in C++

Copyright © 2017, ZeroC, Inc.