Example of a File System Server in C++

This page presents the source code for a C++ server that implements our file system and communicates with the client we wrote earlier. The code is fully functional, apart from the required interlocking for threads.

The server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

The server code shown here is not quite correct as it stands: if two clients access the same file in parallel, each via a different thread, one thread may read the `_lines` data member while another thread updates it. Obviously, if that happens, we may write or return garbage or, worse, crash the server. However, it is trivial to make the `read` and `write` operations thread-safe: a single data member and two lines of source code are sufficient to achieve this. We discuss how to write thread-safe servant implementations in Threads and Concurrency with C++.

On this page:

- Implementing a File System Server in C++
- Server main Program in C++
- Servant Class Definitions in C++
- The Servant Implementation in C++
  - Implementing FileI
  - Implementing DirectoryI
  - Implementing NodeI

Implementing a File System Server in C++

We have now seen enough of the server-side C++ mapping to implement a server for our file system. (You may find it useful to review these Slice definitions before studying the source code.)

Our server is composed of two source files:

- `Server.cpp`
  This file contains the server main program.

- `FilesystemI.cpp`
  This file contains the implementation for the file system servants.

Server `main` Program in C++

Our server main program, in the file `Server.cpp`, uses the `Ice::Application` class. The `run` method installs a signal handler, creates an object adapter, instantiates a few servants for the directories and files in the file system, and then activates the adapter. This leads to a `main` program as follows:

C++

```
#include <Ice/Ice.h>
#include <FilesystemI.h>

using namespace std;
using namespace Filesystem;

class FilesystemApp : virtual public Ice::Application {
public:
    virtual int run(int, char*[]) {
        // Terminate cleanly on receipt of a signal
        //
        shutdownOnInterrupt();

        // Create an object adapter.
        //
        Ice::ObjectAdapterPtr adapter = communicator()->createObjectAdapterWithEndpoints(
            "SimpleFilesystem", "default -p 10000");
```

```
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root = new DirectoryI(communicator(), "/", 0);
        root->activate(adapter);

        // Create a file called "README" in the root directory
        //
        FileIPtr file = new FileI(communicator(), "README", root);
        Lines text;
        text.push_back("This file system contains a collection of poetry.");
        file->write(text);
        file->activate(adapter);

        // Create a directory called "Coleridge"
        // in the root directory
        //
        DirectoryIPtr coleridge = new DirectoryI(communicator(), "Coleridge", root);
        coleridge->activate(adapter);

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = new FileI(communicator(), "Kubla_Khan", coleridge);
        text.erase(text.begin(), text.end());
        text.push_back("In Xanadu did Kubla Khan");
        text.push_back("A stately pleasure-dome decree:");
        text.push_back("Where Alph, the sacred river, ran");
        text.push_back("Through caverns measureless to man");
        text.push_back("Down to a sunless sea.");
        file->write(text);
        file->activate(adapter);

        // All objects are created, allow client requests now
        //
        adapter->activate();

        // Wait until we are done
        //
        communicator()->waitForShutdown();
        if (interrupted()) {
            cerr << appName() << ": received signal, shutting down" << endl;
        }

        return 0;
    };
};

int
main(int argc, char* argv[])
{
    FilesystemApp app;
    return app.main(argc, argv);
}
```

There is quite a bit of code here, so let us examine each section in detail:

C++

```
#include <FilesystemI.h>
#include <Ice/Application.h>

using namespace std;
using namespace Filesystem;
```

The code includes the header file `FilesystemI.h`. That file includes `Ice/Ice.h` as well as the header file that is generated by the Slice compiler, `Filesystem.h`. Because we are using `Ice::Application`, we need to include `Ice/Application.h` as well.

Two `using` declarations, for the namespaces `std` and `Filesystem`, permit us to be a little less verbose in the source code.

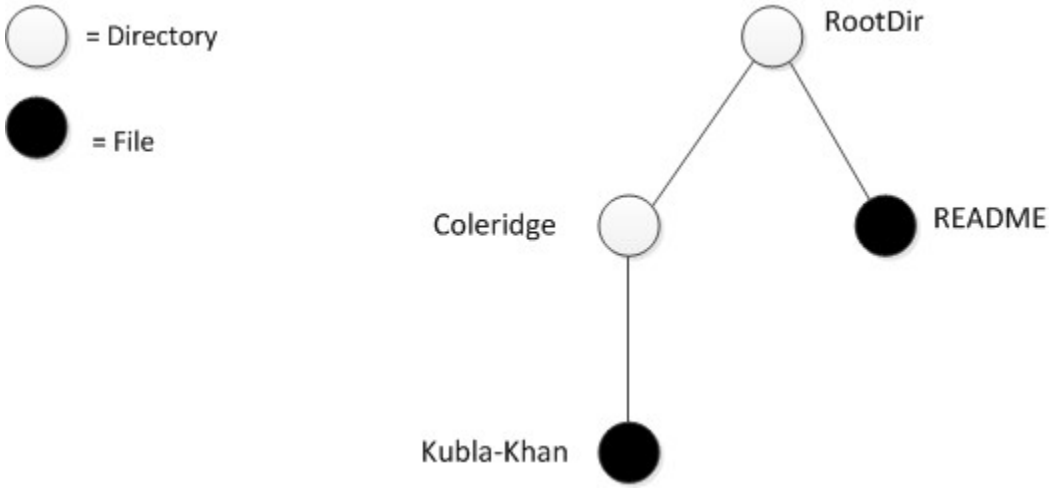
The next part of the source code is the definition of `FilesystemApp`, which derives from `Ice::Application` and contains the main application logic in its `run` method:

C++

```
class FilesystemApp : virtual public Ice::Application {
public:
    virtual int run(int, char*[]) {
        // Terminate cleanly on receipt of a signal
        //
        shutdownOnInterrupt();

        // Create an object adapter.
        //
        Ice::ObjectAdapterPtr adapter = communicator()->createObjectAdapterWithEndpoints(
            "SimpleFilesystem", "default -p 10000");

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root = new DirectoryI(communicator(), "/", 0);
        root->activate(adapter);

        // Create a file called "README" in the root directory
        //
        FileIPtr file = new FileI(communicator(), "README", root);
        Lines text;
        text.push_back("This file system contains a collection of poetry.");
        file->write(text);
        file->activate(adapter);

        // Create a directory called "Coleridge"
        // in the root directory
        //
        DirectoryIPtr coleridge = new DirectoryI(communicator(), "Coleridge", root);
        coleridge->activate(adapter);

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = new FileI(communicator(), "Kubla_Khan", coleridge);
        text.erase(text.begin(), text.end());
        text.push_back("In Xanadu did Kubla Khan");
        text.push_back("A stately pleasure-dome decree:");
        text.push_back("Where Alph, the sacred river, ran");
        text.push_back("Through caverns measureless to man");
        text.push_back("Down to a sunless sea.");
        file->write(text);
        file->activate(adapter);

        // All objects are created, allow client requests now
        //
        adapter->activate();

        // Wait until we are done
        //
        communicator()->waitForShutdown();
        if (interrupted()) {
            cerr << appName() << ": received signal, shutting down" << endl;
        }

        return 0;
    };
};
```

Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown below:



A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts three parameters: the communicator, the name of the directory or file to be created, and a handle to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent handle.) Thus, the statement

C++

```
DirectoryIPtr root = new DirectoryI(communicator(), "/", 0);
```

creates the root directory, with the name `"/"` and no parent directory. Note that we use the smart pointer class to hold the return value from `new`; that way, we avoid any memory management issues. The types `DirectoryIPtr` and `FileIPtr` are defined as follows in a header file `FilesystemI.h`:

C++

```
typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;
typedef IceUtil::Handle<FileI> FileIPtr;
```

Here is the code that establishes the structure in the illustration above.

C++

```
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root = new DirectoryI(communicator(), "/", 0);
        root->activate(adapter);

        // Create a file called "README" in the root directory
        //
        FileIPtr file = new FileI(communicator(), "README", root);
        Lines text;
        text.push_back("This file system contains a collection of poetry.");
        file->write(text);
        file->activate(adapter);

        // Create a directory called "Coleridge"
        // in the root directory
        //
        DirectoryIPtr coleridge = new DirectoryI(communicator(), "Coleridge", root);
        coleridge->activate(adapter);

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = new FileI(communicator(), "Kubla_Khan", coleridge);
        text.erase(text.begin(), text.end());
        text.push_back("In Xanadu did Kubla Khan");
        text.push_back("A stately pleasure-dome decree:");
        text.push_back("Where Alph, the sacred river, ran");
        text.push_back("Through caverns measureless to man");
        text.push_back("Down to a sunless sea.");
        file->write(text);
        file->activate(adapter);
```

We first create the root directory and a file `README` within the root directory. (Note that we pass the handle to the root directory as the parent pointer when we create the new node of type `FileI`.)

After creating each servant, the code calls `activate` on the servant. (We will see the definition of this member function shortly.) The `activate` member function adds the servant to the ASM.

The next step is to fill the file with text:

C++

```
        FileIPtr file = new FileI(communicator(), "README", root);
        Lines text;
        text.push_back("This file system contains a collection of poetry.");
        file->write(text);
        file->activate(adapter);
```

Recall that Slice sequences map to STL vectors. The Slice type `Lines` is a sequence of strings, so the C++ type `Lines` is a vector of strings; we add a line of text to our `README` file by calling `push_back` on that vector.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

C++

```
        file->write(text);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a smart class pointer (of type `FilePtr`) and not via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place — such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary C++ function call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in the above illustration.

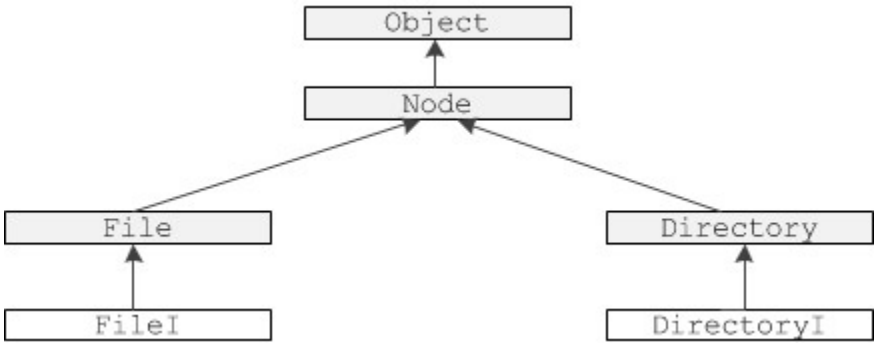
Servant Class Definitions in C++

We must provide servants for the concrete interfaces in our Slice specification, that is, we must provide servants for the `File` and `Directory` interfaces in the C++ classes `FileI` and `DirectoryI`. This means that our servant classes might look as follows:

C++

```
namespace Filesystem {
  class FileI : virtual public File {
    // ...
  };

  class DirectoryI : virtual public Directory {
     // ...
  };
}
```

This leads to the C++ class structure as shown:



File system servants using interface inheritance.

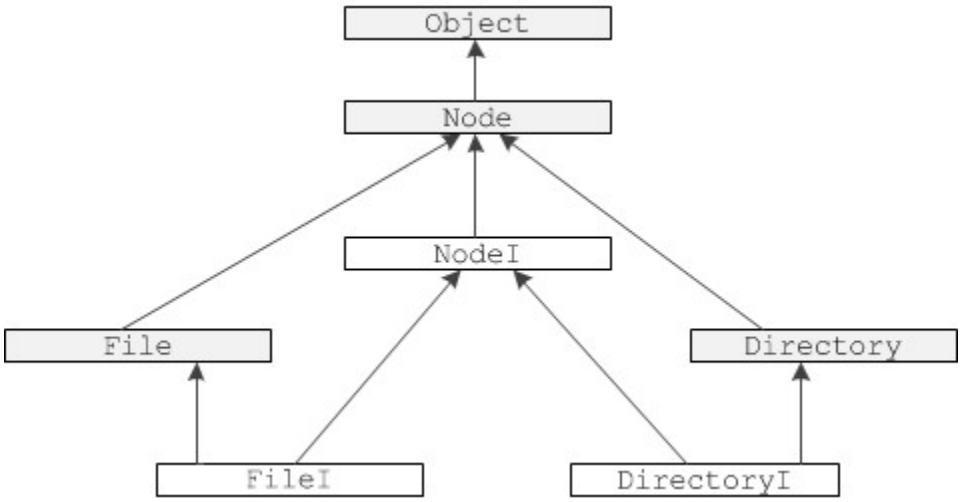
The shaded classes in the illustration above are skeleton classes and the unshaded classes are our servant implementations. If we implement our servants like this, `FileI` must implement the pure virtual operations it inherits from the `File` skeleton (`read` and `write`), as well as the operation it inherits from the `Node` skeleton (`name`). Similarly, `DirectoryI` must implement the pure virtual function it inherits from the `Directory` skeleton (`list`), as well as the operation it inherits from the `Node` skeleton (`name`). Implementing the servants in this way uses interface inheritance from `Node` because no implementation code is inherited from that class.

Alternatively, we can implement our servants using the following definitions:

C++

```
namespace Filesystem {
  class NodeI : virtual public Node {
    // ...
  };

  class FileI : virtual public File, virtual public NodeI {
    // ...
  };

  class DirectoryI : virtual public Directory, virtual public NodeI {
    // ...
  };
}
```

This leads to the C++ class structure shown:



File system servants using implementation inheritance.

In this implementation, `NodeI` is a concrete base class that implements the `name` operation it inherits from the `Node` skeleton. `FileI` and `DirectoryI` use multiple inheritance from `NodeI` and their respective skeletons, that is, `FileI` and `DirectoryI` use implementation inheritance from their `NodeI` base class.

Either implementation approach is equally valid. Which one to choose simply depends on whether we want to re-use common code provided by `NodeI`. For the implementation that follows, we have chosen the second approach, using implementation inheritance.

Given the structure in the above illustration and the operations we have defined in the Slice definition for our file system, we can add these operations to the class definition for our servants:

C++

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

  class FileI : virtual public File, virtual public NodeI {
  public:
    virtual Lines read(const Ice::Current&);
    virtual void write(const Lines&, const Ice::Current&);
  };

  class DirectoryI : virtual public Directory, virtual public NodeI {
  public:
    virtual NodeSeq list(const Ice::Current&);
  };
}
```

This simply adds signatures for the operation implementations to each class. Note that the signatures must exactly match the operation signatures in the generated skeleton classes — if they do not match exactly, you end up overloading the pure virtual function in the base class instead of overriding it, meaning that the servant class cannot be instantiated because it will still be abstract. To avoid signature mismatches, you can copy the signatures from the generated header file (`Filesystem.h`), or you can use the `--impl` option with `slice2cpp` to generate header and implementation files that you can add your application code to.

Now that we have the basic structure in place, we need to think about other methods and data members we need to support our servant implementation. Typically, each servant class hides the copy constructor and assignment operator, and has a constructor to provide initial state for its data members. Given that all nodes in our file system have both a name and a parent directory, this suggests that the `NodeI` class should implement the functionality relating to tracking the name of each node, as well as the parent-child relationships:

C++

```
namespace Filesystem {
  class DirectoryI;
  typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

  class NodeI : virtual public Node {
  public:
    virtual std::string name(const Ice::Current&);
    NodeI(const Ice::CommunicatorPtr&, const std::string&, const DirectoryIPtr&);
    void activate(const Ice::ObjectAdapterPtr&);
  private:
    std::string _name;
    Ice::Identity _id;
    DirectoryIPtr _parent;
    NodeI(const NodeI&);               // Copy forbidden
    void operator=(const NodeI&);      // Assignment forbidden
  };
}
```

The `NodeI` class has a private data member to store its name (of type `std::string`) and its parent directory (of type `DirectoryIPtr`). The constructor accepts parameters that set the value of these data members. For the root directory, by convention, we pass a null handle to the constructor to indicate that the root directory has no parent. The constructor also requires the communicator to be passed to it. This is necessary because the constructor creates the identity for the servant, which requires access to the communicator. The `activate` member function adds the servant to the ASM (which requires access to the object adapter) and connects the child to its parent.

The `FileI` servant class must store the contents of its file, so it requires a data member for this. We can conveniently use the generated `Lines` type (which is a `std::vector<std::string>`) to hold the file contents, one string for each line. Because `FileI` inherits from `NodeI`, it also requires a constructor that accepts the communicator, file name, and parent directory, leading to the following class definition:

C++

```
namespace Filesystem {
  class FileI : virtual public File, virtual public NodeI {
  public:
    virtual Lines read(const Ice::Current&);
    virtual void write(const Lines&, const Ice::Current&);
    FileI(const Ice::CommunicatorPtr&, const std::string&, const DirectoryIPtr&);
  private:
    Lines _lines;
  };
}
```

For directories, each directory must store its list of child notes. We can conveniently use the generated `NodeSeq` type (which is a `vector<NodePrx>`) to do this. Because `DirectoryI` inherits from `NodeI`, we need to add a constructor to initialize the directory name and its parent directory. As we will see shortly, we also need a private helper function, `addChild`, to make it easier to connect a newly created directory to its parent. This leads to the following class definition:

C++

```
namespace Filesystem {
  class DirectoryI : virtual public Directory, virtual public NodeI {
  public:
    virtual NodeSeq list(const Ice::Current&) const;
    DirectoryI(const Ice::CommunicatorPtr&, const std::string&, const DirectoryIPtr&);
    void addChild(NodePrx child);
  private:
    NodeSeq _contents;
  };
}
```

Servant Header File Example

Putting all this together, we end up with a servant header file, `FilesystemI.h`, as follows:

C++

```
#include <Ice/Ice.h>
#include <Filesystem.h>

namespace Filesystem {
  class DirectoryI;
  typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

  class NodeI : virtual public Node {
  public:
    virtual std::string name(const Ice::Current&);
    NodeI(const Ice::CommunicatorPtr&, const std::string&, const DirectoryIPtr&);
    void activate(const Ice::ObjectAdapterPtr&);
  private:
    std::string _name;
    Ice::Identity _id;
    DirectoryIPtr _parent;
    NodeI(const NodeI&);            // Copy forbidden
    void operator=(const NodeI&);   // Assignment forbidden
  };

  typedef IceUtil::Handle<NodeI> NodeIPtr;

  class FileI : virtual public File, virtual public NodeI {
  public:
    virtual Lines read(const Ice::Current&);
    virtual void write(const Lines&, const Ice::Current& = Ice::Current());
    FileI(const Ice::CommunicatorPtr&, const std::string&, const DirectoryIPtr&);
  private:
    Lines _lines;
  };

  typedef IceUtil::Handle<FileI> FileIPtr;

  class DirectoryI : virtual public Directory, virtual public NodeI {
  public:
    virtual NodeSeq list(const Ice::Current&);
    DirectoryI(const Ice::CommunicatorPtr&, const std::string&, const DirectoryIPtr&);
    void addChild(const Filesystem::NodePrx&);
  private:
    Filesystem::NodeSeq _contents;
  };
}
```

The Servant Implementation in C++

The implementation of our servants is mostly trivial, following from the class definitions in our `FilesystemI.h` header file.

Implementing `FileI`

The implementation of the `read` and `write` operations for files is trivial: we simply store the passed file contents in the `_lines` data member. The constructor is equally trivial, simply passing its arguments through to the `NodeI` base class constructor:

C++

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&)
{
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text, const Ice::Current&)
{
    _lines = text;
}

Filesystem::FileI::FileI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent)
    : NodeI(communicator, name, parent)
{
}
```

Implementing `DirectoryI`

The implementation of `DirectoryI` is equally trivial: the `list` operation simply returns the `_contents` data member and the constructor passes its arguments through to the `NodeI` base class constructor:

C++

```
Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current&)
{
    return _contents;
}

Filesystem::DirectoryI::DirectoryI(const Ice::CommunicatorPtr& communicator,
                                   const string& name,
                                   const DirectoryIPtr& parent)
    : NodeI(name, parent)
{
}

void
Filesystem::DirectoryI::addChild(const NodePrx child)
{
    _contents.push_back(child);
}
```

The only noteworthy thing is the implementation of `addChild`: when a new directory or file is created, the constructor of the `NodeI` base class calls a `ddChild` on its own parent, passing it the proxy to the newly-created child. The implementation of `addChild` appends the passed reference to the contents list of the directory it is invoked on (which is the parent directory).

Implementing `NodeI`

The name operation of our `NodeI` class is again trivial: it simply returns the `_name` data member:

C++

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

The `NodeI` constructor creates an identity for the servant:

C++

```
Filesystem::NodeI::NodeI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent)
    : _name(name), _parent(parent)
{
    _id.name = parent ? IceUtil::generateUUID() : "RootDir";
}
```

For the root directory, we use the fixed identity `"RootDir"`. This allows the client to create a proxy for the root directory. For directories other than the root directory, we use a UUID as the identity.

Finally, `NodeI` provides the `activate` member function that adds the servant to the ASM and connects the child node to its parent directory:

C++

```
void
Filesystem::NodeI::activate(const Ice::ObjectAdapterPtr& a)
{
    NodePrx thisNode = NodePrx::uncheckedCast(a->add(this, _id));
    if(_parent)
    {
        _parent->addChild(thisNode);
    }
}
```

This completes our servant implementation. The complete source code is shown here once more:

C++

```
#include <IceUtil/IceUtil.h>
#include <FilesystemI.h>

using namespace std;

// Slice Node::name() operation

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

// NodeI constructor

Filesystem::NodeI::NodeI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent)
    : _name(name), _parent(parent)
```

```
{
    // Create an identity. The root directory has the fixed identity "RootDir"
    //
    _id.name = parent ? IceUtil::generateUUID() : "RootDir";
}

// NodeI activate() member function

void
Filesystem::NodeI::activate(const Ice::ObjectAdapterPtr& a)
{
    NodePrx thisNode = NodePrx::uncheckedCast(a->add(this, _id));
    if(_parent)
    {
        _parent->addChild(thisNode);
    }
}

// Slice File::read() operation

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&)
{
    return _lines;
}

// Slice File::write() operation

void
Filesystem::FileI::write(const Filesystem::Lines& text, const Ice::Current&)
{
    _lines = text;
}

// FileI constructor

Filesystem::FileI::FileI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent)
    : NodeI(communicator, name, parent)
{
}

// Slice Directory::list() operation

Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current& c)
{
    return _contents;
}

// DirectoryI constructor

Filesystem::DirectoryI::DirectoryI(const Ice::CommunicatorPtr& communicator,
                                   const string& name,
                                   const DirectoryIPtr& parent)
    : NodeI(communicator, name, parent)
{
}

// addChild is called by the child in order to add
// itself to the _contents member of the parent

void
Filesystem::DirectoryI::addChild(const NodePrx& child)
{
```

```
    _contents.push_back(child);
}
```

See Also

- Slice for a Simple File System
- Example of a File System Client in C++
- The `Ice::Application` Class
- C++ Mapping for Sequences
- slice2cpp Command-Line Options
- UUIDs as Identities in C++
- Threads and Concurrency with C++