# Server-Side Slice-to-C++ Mapping

# Server-Side Slice-to-C++ Mapping

- The mapping for Slice data types to C++ is identical on the client side and server side.
- For the server side, there are a few additional things you need to know, specifically:
  - How to initialize and finalize the server-side run time
  - How to implement servants
  - How to pass parameters and throw exceptions
  - How to create servants and register them with the Ice run time.

# The Server-Side `main` Function

- The main entry point to the Ice run time is represented by the local interface Ice::Communicator.
- As for the client side, you must initialize the Ice run time by calling **`Ice::initialize`** before you can do anything else in your server.
- Ice::initialize returns a smart pointer to an instance of an Ice::Communicator:

```
int main(int argc, char* argv[])
{
Ice::CommunicatorPtr ic
= Ice::initialize(argc, argv);
// ...
}
```

- **`Ice::initialize`** accepts a C++ reference to argc and argv.
- The function scans the argument vector for any command-line options that are relevant to the Ice run time.
  - Any such options are removed from the argument vector so, when **`Ice::initialize`** returns, the only options and arguments remaining are those that concern your application.
- If anything goes wrong during initialization, initialize throws an exception.

# The Server-Side `main` Function

- Before leaving your main function, you must call **`Communicator::destroy`**.
- The destroy operation is responsible for finalizing the Ice run time.
  - In particular, destroy waits for any operation implementations that are still executing in the server to complete. In addition, destroy ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory.
  - Never allow your main function to terminate without calling destroy first; doing so has undefined behavior.

# The Server-Side `main` Function

- The general shape of our server-side main function is therefore as follows:

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

int
main(int argc, char* argv[])
{
  int status = 0;
  Ice::CommunicatorPtr ic;
  try {
    ic = Ice::initialize(argc, argv);
    // Server code here...
  } catch (const Ice::Exception& e) {
    cerr << e << endl;
    status = 1;
  } catch (const std::string& msg) {
    cerr << msg << endl;
    status = 1;
  } catch (const char* msg) {
    cerr << msg << endl;
    status = 1;
  }
  if (ic) {
    try {
      ic->destroy();
    } catch (const std::string& msg) {
      cerr << msg << endl;
      status = 1;
    }
  }
  return status;
}
```

# The `Ice::Application` Class

- The preceding structure for the **`main`** function is so common that Ice offers a class, **`Ice::Application`**, that encapsulates all the correct initialization and finalization activities.

```
namespace Ice {
  enum SignalPolicy { HandleSignals, NoSignalHandling };
  class Application /* ... */ {
  public:
    Application(SignalPolicy = HandleSignals);
    virtual ~Application();
    int main(int argc, char*[] argv);
    int main(int, char*[], const char* config);
    int main(int argc, char*[] argv,
             const Ice::InitializationData& id);
    int main(const Ice::StringSeq&);
    int main(const Ice::StringSeq&, const char* config);
    int main(const Ice::StringSeq&,
             const Ice::InitializationData& id);
    virtual int run(int, char*[]) = 0;
    static const char* appName();
    static CommunicatorPtr communicator();
    static bool interrupted();
    // ...
  };
}
```

# The `Ice::Application` Class

- The intent of this class is that you specialize **`Ice::Application`** and implement the pure virtual run method in your derived class. Whatever code you would normally place in main goes into the run method instead.
- Using **`Ice::Application`**, our program looks as follows:

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

class MyApplication : virtual public Ice::Application {
  public:
  virtual int run(int, char*[]) {
  // Server code here...
  if (interrupted())
    cerr << appName() << ": terminating" << endl;
  return 0;
  }
};

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

- The class also handles the OS signals and, by default, shuts down the server cleanly.
- The **`interrupted`** function returns true if a signal caused the communicator to shut down, false otherwise.
  - This allows us to distinguish intentional shutdown from a forced shutdown that was caused by a signal.

# The `Ice::Application` Class

- The **`Application::main`** function does the following:
  - It installs an exception handler for **`Ice::Exception`**. If your code fails to handle an Ice exception, **`Application::main`** prints the exception details on stderr before returning with a non-zero return value.
  - It installs exception handlers for **`const std::string &`** and **`const char *`**. This allows you to terminate your server in response to a fatal error condition by throwing a **`std::string`** or a string literal. **`Application::main`** prints the string on **`stderr`** before returning a nonzero return value.
  - It initializes (by calling **`Ice::initialize`**) and finalizes (by calling **`Communicator::destroy`**) a communicator. You can get access to the communicator for your server by calling the static **`communicator()`** member.
  - It scans the argument vector for options that are relevant to the Ice run time and removes any such options. The argument vector that is passed to your run method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.

# The `Ice::Application` Class

- The **`Application::main`** function does also the following:
  - It provides the name of your application via the static **`appName`** member function. The return value from this call is **`argv[0]`**, so you can get at **`argv[0]`** from anywhere in your code by calling **`Ice::Application::appName`** (which is usually required for error messages).
  - It creates an **`IceUtil::CtrlCHandler`** that properly destroys the communicator.
  - It installs a per-process logger, if the application has not already configured one. The per-process logger uses the value of the **`Ice.ProgramName`** property as a prefix for its messages and sends its output to the standard error channel. An application can specify an alternate logger by including it in the **`InitializationData`** structure.
- Using **`Ice::Application`** ensures that your program properly finalizes the Ice run time, whether your server terminates normally or in response to an exception or signal.

# 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.

# Skeleton Classes

- 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.
- Consider the Slice definition for the Node interface:

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

- The Slice compiler generates the following definition for this interface:

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

# Skeleton Classes

- 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 order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class.

```
#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;
};
```

- **`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.
  - Virtual inheritance is necessary only for servants that implement interfaces that use multiple inheritance.

# 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.
  - In the preceding definition, we added a **`_name`** member and a constructor

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

# Parameter Passing

- For each parameter of a Slice operation, the C++ mapping generates a corresponding parameter for the virtual member function in the skeleton.
- In addition, every operation has an additional, trailing parameter of type Ice::Current, which provides provides access to additional information about the currently executing request.
- Parameter passing on the server side follows the rules for the client side:
  - in-parameters are passed by value or const reference.
  - out-parameters are passed by reference.
  - return values are passed by value

# Parameter Passing

```
module M {
  interface Example {
    string op(string sin, out string sout);
  };
};
```

```
namespace M {
  class Example : virtual public ::Ice::Object {
  public:
    virtual std::string op(const std::string&, std::string&,
                           const Ice::Current& = Ice::Current()) = 0;
  // ...
  };
}
```

- We could implement **op** as follows:

```
std::string
ExampleI::op(const std::string& sin,
             std::string& sout,
             const Ice::Current&)
{
  cout << sin << endl; // In parameters are initialized
  sout = "Hello World!"; // Assign out parameter
  return "Done"; // Return a string
}
```

# Raising Exceptions

- To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it.

```
void Filesystem::FileI::write(const Filesystem::Lines& text,
                              const Ice::Current&)
{
  // Try to write the file contents here...
  // Assume we are out of space...
  if (error) {
    Filesystem::GenericError e;
    e.reason = "file too large";
    throw e;
  }
};
```

- No memory management issues arise in the presence of exceptions.

# Raising Exceptions

- The Slice compiler never generates exception specifications for operations, regardless of whether the corresponding Slice operation definition has an exception specification or not.
  - C++ exception specifications do not add any value and are therefore not used by the Ice C++ mapping.
- If you throw an arbitrary C++ exception (such as an int or other unexpected type), the Ice run time catches the exception and then returns an **`UnknownException`** to the client.
- Similarly, if you throw an "impossible" user exception (a user exception that is not listed in the exception specification of the operation), the client receives an **`UnknownUserException`**.
- If you throw a run-time exception, such as **`MemoryLimitException`**, the client receives an **`UnknownLocalException`**.
  - For that reason, you should never throw system exceptions from operation implementations.
  - If you do, all the client will see is an **`UnknownLocalException`**, which does not tell the client anything useful.

# Object Incarnation

- Having created a servant class such as the rudimentary **`NodeI`** class, you can instantiate the class to create a concrete servant that can receive invocations from a client.
- Merely instantiating a servant class is insufficient to incarnate an object.
- Specifically, to provide an implementation of an Ice object, you must follow the following steps:
  - Instantiate a servant class.
  - Create an identity for the Ice object incarnated by the servant.
  - Inform the Ice run time of the existence of the servant.
  - Pass a proxy for the object to a client so the client can reach it.

# Instantiating a Servant

- Instantiating a servant means to allocate an instance on the heap:
  - `NodePtr servant = new NodeI("Fred");`
- This code creates a new **`NodeI`** instance on the heap and assigns its address to a smart pointer of type **`NodePtr`**.
  - This works because **`NodeI`** is derived from Node, so a smart pointer of type **`NodePtr`** can also look after an instance of type **`NodeI`**.
- However, if we want to invoke a member function of the derived **`NodeI`** class at this point, we have a problem: we cannot access member functions of the derived **`NodeI`** class through a **`NodePtr`** smart pointer, only member functions of **`Node`** base class.
- To get around this, we can modify the code as follows:

```
typedef IceUtil::Handle<NodeI> NodeIPtr;
NodeIPtr servant = new NodeI("Fred");
```

# Creating an Identity

- Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.
- An Ice object identity is a structure with the following Slice definition:

```
module Ice {
   struct Identity {
      string name;
      string category;
   };
   // ...
};
```

- The full identity of an object is the combination of both the name and category fields of the Identity structure.
  - For now, we will leave the category field as the empty string and simply use the name field.
- To create an identity, we simply assign a key that identifies the servant to the name field of the Identity structure:

```
Ice::Identity id;
id.name = "Fred"; // Not unique, but good enough for now
```

# Activating a Servant

- Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant.
- To activate a servant, you invoke the add operation on the object adapter.
- Assuming that we have access to the object adapter in the **`_adapter`** variable, we can write:
  - `_adapter->add(servant, id);`
- Note the two arguments to add: the smart pointer to the servant and the object identity.
- Calling add on the object adapter adds the servant pointer and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:
  - The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
  - The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
  - If a servant with that identity is active, the object adapter retrieves the servant pointer from the servant map and dispatches the incoming request into the correct member function on the servant.
- Assuming that the object adapter is in the active state, client requests are dispatched to the servant as soon as you call add.

# Servant Life Time and Reference Counts

- Putting the preceding points together, we can write a simple function that instantiates and activates one of our **`NodeI`** servants.
- For this example, we use a simple helper function called activateServant that creates and activates a servant with a given identity:

```
void activateServant(const string& name)
{
  NodePtr servant = new NodeI(name);
     // Refcount == 1
  Ice::Identity id;
  id.name = name;
  _adapter->add(servant, id);
     // Refcount == 2
}
```

# Servant Life Time and Reference Counts

- We create the servant on the heap and that, once **`activateServant`** returns, we lose the last remaining handle to the servant.
- What happens to the heap-allocated servant instance?
  - When the new servant is instantiated, its reference count is initialized to 0.
  - Assigning the servant's address to the servant smart pointer increments the servant's reference count to 1.
  - Calling add passes the servant smart pointer to the object adapter which keeps a copy of the handle internally. This increments the reference count of the servant to 2.
  - When **`activateServant`** returns, the destructor of the servant variable decrements the reference count of the servant to 1.
- The net effect is that the servant is retained on the heap with a reference count of 1 for as long as the servant is in the servant map of its object adapter. (If we deactivate the servant, that is, remove it from the servant map, the reference count drops to zero and the memory occupied by the servant is reclaimed.

# UUIDs as Identities

- The Ice object model assumes that object identities are globally unique.
- One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities.
- The IceUtil namespace contains a helper function to create such identities:

```
#include <IceUtil/UUID.h>
#include <iostream>
using namespace std;
int main()
{
  cout << IceUtil::generateUUID() << endl;
}
```

- When executed, this program prints a unique string such as 5029a22c-e333-4f87-86b1-cd5e0fcce509.
- Each call to **`generateUUID`** creates a string that differs from all previous ones.

# UUIDs as Identities

- You can use a UUID such as this to create object identities.
- For convenience, the object adapter has an operation **`addWithUUID`** that generates a UUID and adds a servant to the servant map in a single step.
- Using this operation, we can rewrite the code like this:

```
void activateServant(const string& name)
{
  NodePtr servant = new NodeI(name);
  _adapter->addWithUUID(servant);
}
```

# Creating Proxies

- Once we have activated a servant for an Ice object, the server can process incoming client requests for that object.
- Clients can only access the object once they hold a proxy for the object.
  - If a client knows the server's address details and the object identity, it can create a proxy from a string.
  - Creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping.
  - Once the client has an initial proxy, it typically obtains further proxies by invoking operations.
- The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity.
- The Ice run time offers a number of ways to create proxies.
- Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

# Proxies and Servant Activation

- The **`add`** and **`addWithUUID`** servant activation operations on the object adapter return a proxy for the corresponding Ice object.
- This means we can write:

```
typedef IceUtil::Handle<NodeI> NodeIPtr;
NodeIPtr servant = new NodeI(name);
NodePrx proxy = NodePrx::uncheckedCast(
                        _adapter->addWithUUID(servant));
// Pass proxy to client...
```

- Here, **`addWithUUID`** both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.
- We need to use an **`uncheckedCast`** here because **`addWithUUID`** returns a proxy of type **`Ice::ObjectPrx`**.

# Direct Proxy Creation

- The object adapter offers an operation to create a proxy for a given identity:

```
module Ice {
  local interface ObjectAdapter {
    Object* createProxy(Identity id);
    // ...
  };
};
```

- Note that **`createProxy`** creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

```
Ice::Identity id;
id.name = IceUtil::generateUUID();
ObjectPrx o = _adapter->createProxy(id);
```

- This creates a proxy for an Ice object with the identity returned by **`generateUUID`**. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an **`ObjectNotExistException`**.