Writing an Ice Application with C++

This page shows how to create an Ice application with C++.

On this page:

Compiling a Slice Definition for C++

The first step in creating our C++ application is to compile our Slice definition to generate C++ proxies and skeletons. You can compile the definition as follows:

```
$ slice2cpp Printer.ice
```

The `slice2cpp` compiler produces two C++ source files from this definition, `Printer.h` and `Printer.cpp`.

- `Printer.h`
  The `Printer.h` header file contains C++ type definitions that correspond to the Slice definitions for our `Printer` interface. This header file must be included in both the client and the server source code.

- `Printer.cpp`
  The `Printer.cpp` file contains the source code for our `Printer` interface. The generated source contains type-specific run-time support for both clients and servers. For example, it contains code that marshals parameter data (the string passed to the `printString` operation) on the client side and unmarshals that data on the server side.
  The `Printer.cpp` file must be compiled and linked into both client and server.

Writing and Compiling a Server in C++

The source code for the server takes only a few lines and is shown in full here:

C++

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

using namespace std;
using namespace Demo;

class PrinterI : public Printer {
public:
    virtual void printString(const string& s, const Ice::Current&);
};

void
PrinterI::
printString(const string& s, const Ice::Current&)
{
    cout << s << endl;
}

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter =
            ic->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object, ic->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ic->waitForShutdown();
    } catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const Ice::Exception& e) {
            cerr << e << endl;
            status = 1;
        }
    }
    return status;
}
```

There appears to be a lot of code here for something as simple as a server that just prints a string. Do not be concerned by this: most of the preceding code is boiler plate that never changes. For this very simple server, the code is dominated by this boiler plate.

Every Ice source file starts with an include directive for `Ice.h`, which contains the definitions for the Ice run time. We also include `Printer.h`, which was generated by the Slice compiler and contains the C++ definitions for our printer interface, and we import the contents of the `std` and `Demo` namespaces for brevity in the code that follows:

C++

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

using namespace std;
using namespace Demo;
```

Our server implements a single printer servant, of type `PrinterI`. Looking at the generated code in `Printer.h`, we find the following (tidied up a little to get rid of irrelevant detail):

C++

```
namespace Demo {
    class Printer : virtual public Ice::Object {
    public:
        virtual void printString(const std::string&, const Ice::Current& = Ice::Current()) = 0;
    };
};
```

The `Printer` skeleton class definition is generated by the Slice compiler. (Note that the `printString` method is pure virtual so the skeleton class cannot be instantiated.) Our servant class inherits from the skeleton class to provide an implementation of the pure virtual `printString` method. (By convention, we use an `I`-suffix to indicate that the class implements an interface.)

C++

```
class PrinterI : public Printer {
public:
    virtual void printString(const string& s, const Ice::Current&);
};
```

The implementation of the `printString` method is trivial: it simply writes its string argument to `stdout`:

C++

```
void
PrinterI::
printString(const string& s, const Ice::Current&)
{
    cout << s << endl;
}
```

Note that `printString` has a second parameter of type `Ice::Current`. As you can see from the definition of `Printer::printString`, the Slice compiler generates a default argument for this parameter, so we can leave it unused in our implementation. (We will examine the purpose of the `Ice::Current` parameter later.)

What follows is the server main program. Note the general structure of the code:

C++

```
int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {

        // Server implementation here...

    } catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const Ice::Exception& e) {
            cerr << e << endl;
            status = 1;
        }
    }
    return status;
}
```

The body of `main` contains the declaration of two variables, `status` and `ic`. The `status` variable contains the exit status of the program and the `ic` variable, of type `Ice::CommunicatorPtr`, contains the main handle to the Ice run time.

Following these declarations is a `try` block in which we place all the server code, followed by two `catch` handlers. The first handler catches all exceptions that may be thrown by the Ice run time; the intent is that, if the code encounters an unexpected Ice run-time exception anywhere, the stack is unwound all the way back to `main`, which prints the exception and then returns failure to the operating system. The second handler catches string constants; the intent is that, if we encounter a fatal error condition somewhere in our code, we can simply throw a string literal with an error message. Again, this unwinds the stack all the way back to `main`, which prints the error message and then returns failure to the operating system.

Following the `try` block, we see a bit of cleanup code that calls the `destroy` method on the communicator (provided that the communicator was initialized). The cleanup call is outside the first `try` block for a reason: we must ensure that the Ice run time is finalized whether the code terminates normally or terminates due to an exception.

Failure to call `destroy` on the communicator before the program exits results in undefined behavior.

The body of the first `try` block contains the actual server code:

C++

```
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter =
            ic->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object, ic->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ic->waitForShutdown();
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice::initialize`. (We pass `argc` and `argv` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns a smart pointer to an `Ice::Communicator` object, which is the main object in the Ice run time.

2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.) The server starts to process incoming requests from clients as soon as the adapter is activated.
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice provides such a helper class, called `Ice::Application`.) As far as actual application code is concerned, the server contains only a few lines: six lines for the definition of the `PrinterI` class, plus three lines to instantiate a `PrinterI` object and register it with the object adapter.

Assuming that we have the server code in a file called `Server.cpp`, we can compile it as follows:

```
$ c++ -I. -c Printer.cpp Server.cpp
```

This compiles both our application code and the code that was generated by the Slice compiler. Depending on your platform, you may have to add additional include directives or other options to the compiler; please see the demo programs that ship with Ice for the details.

Finally, we need to link the server into an executable:

```
$ c++ -o server Printer.o Server.o -lIce -lIceUtil
```

Again, depending on the platform, the actual list of libraries you need to link against may be longer. The demo programs that ship with Ice contain all the detail. The important point to note here is that the Ice run time is shipped in two libraries, `libIce` and `libIceUtil`.

Writing and Compiling a Client in C++

The client code looks very similar to the server. Here it is in full:

C++

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

using namespace std;
using namespace Demo;

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectPrx base = ic->stringToProxy("SimplePrinter:default -p 10000");
        PrinterPrx printer = PrinterPrx::checkedCast(base);
        if (!printer)
            throw "Invalid proxy";

        printer->printString("Hello World!");
    } catch (const Ice::Exception& ex) {
        cerr << ex << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}
```

Note that the overall code layout is the same as for the server: we include the headers for the Ice run time and the header generated by the Slice compiler, and we use the same `try` block and `catch` handlers to deal with errors.

The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice::initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrx::checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy to a `Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns a null proxy.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling and linking the client looks much the same as for the server:

```
$ c++ -I. -I$ICE_HOME/include -c Printer.cpp Client.cpp
$ c++ -o client Printer.o Client.o -lIce -lIceUtil
```

Running Client and Server in C++

To run client and server, we first start the server in a separate window:

```
$ ./server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ ./client
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in our discussion of `Ice::Application`.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get:

```
Network.cpp:471: Ice::ConnectFailedException:
connect failed: Connection refused
```

See Also

- Client-Side Slice-to-C++ Mapping
- Server-Side Slice-to-C++ Mapping
- The `Ice::Application` Class
- The Current Object
- IceGrid