Connections

ZeroC's Newsletter for the Ice Community

Issue Number 16, August 2006

Plans for the Future

Companies are often hesitant to reveal details about their plans for future products. As much as the lack of information might frustrate their customers at times, this caution is understandable: companies must strike a balance between preserving a competitive advantage and accommodating their customers' needs to plan their own futures. Silence can also be a publicity tool in its own right. Take Apple, for example, a company so adamant about restricting information that any rumor about an imminent gadget makes headlines worldwide.

Things aren't quite so glamorous for us yet at ZeroC. Our willingness to freely share information about our product roadmap is influenced by several factors, but concern about the competition is rarely one of them. However, we do feel it's important that we don't make promises we can't keep. For that reason, we don't make precise long-range forecasts about release dates; the odds are good that we'll pick the wrong date, and we'd rather release the software when we feel it's ready instead of having our hand forced by some arbitrary deadline.

A bigger factor is our style of development, which places more emphasis on immediate functionality and less importance on grandiose vision. As has often been stated, we don't add new features or introduce new products unless there is a compelling and practical use case. Some of these use cases we generate internally, and some are driven by customer demand. We also have a list of projects that would be interesting to do, but those don't receive any serious attention until that real-world requirement comes along to make the project worthwhile and give it a good chance of actually being useful.

Anyone who has frequented our user forums will know the standard ZeroC response to requests for new features: unless an idea will make a significant improvement in performance or ease of use, we generally suggest that the submitter contact us directly to discuss an arrangement in which the organization sponsors the development of the feature. ZeroC retains ownership of the new code and eventually includes it in a released product, while the sponsor gains influence over product direction and early access to new features, not to mention the appreciation and gratitude of the Ice community that benefits from these arrangements.

An example of this process is the new Ruby mapping for Ice, which is currently in development. We have been asked many times to support this popular scripting language, but until recently there was no commercial demand for it and therefore we couldn't justify the effort, no matter how much we liked the language. Now that a company has sponsored its development, we anticipate its inclusion in the next major Ice release. (You probably know better by now than to ask us when that will be.) Given Ruby's lack of support for native threads, the mapping will include only client-side functionality at first.

Another project that you can expect to bear fruit in the near future is support for replication in IceGrid. Not to be confused with replicated object adapters, replication in this case means eliminating the IceGrid registry as a single point of failure. We have always planned to rectify this situation, and its time has come. At least initially, it will most likely take the form of a master and slave configuration in which modifications to deployed applications are only possible while the master registry is available, with slave registries capable of responding to locator queries so that already-deployed applications can continue to function in case the master goes offline.

Although these aren't earth-shattering announcements that will keep bloggers up till the wee hours, we're excited about these developments and we think they'll add value to your applications. As always, we'll rely on our customers and the Ice community to tell us if we're achieving our goal of delivering high-quality middleware that doesn't require a degree in rocket science to use. Join us on our forum and tell us what you think.

Mark Spruiell
Senior Software Engineer

Issue Features

Ice And Character Encoding

In this article Bernard Normier discusses the new character converters that have been added to Ice in the 3.1 release.

Ice Sessions and Resource Allocation

Matthew Newhook writes the first article of a multi-part series on some of the new IceGrid features available in Ice 3.1.

Contents

Ice and Character Encoding 2
IceGrid Sessions and Resource Allocation 8
FAQ Corner 14

Ice and Character Encoding

Bernard Normier, Senior Software Engineer

Introduction

Every programming language provides one or more string types to handle sequences of characters: `java.lang.String` with Java, `std::string` and `std::wstring` with C++, and so on. Most applications don't need to know the in-memory representation of such strings: whether 'A' is encoded as 0x40, 0xC1, or 0x0040 does not matter because the application deals with letters and strings, not numbers. The representation matters only when a string is written to an external device (such as a terminal, a printer, or a file); often, any necessary conversion is performed automatically by the programming language run-time environment when reading from or writing to such a device.

Table 1: Strings in Different Programming Languages

Language	String types	In-memory character encoding format
Java	`java.lang.String`	UTF-16
C#	`System.String`	UTF-16
C++	`std::string` `std::wstring`	Implementation-defined

Naturally, Ice supports strings as well. Slice defines a single string type, `string`. The Ice data encoding, which defines how Slice types appear on the wire, maps a `string` to its length (as a one- or five-byte count), followed by the string's characters encoded in UTF-8.

Slice strings are mapped to each programming language's native string type. When several string types are available, such as C++ `std::string` and `std::wstring`, Slice metadata directives allow you to select the desired mapping.

UTF-8 Serialization with C# and Java

When you send a C# or Java string over the wire, the Ice run time needs to transform the string into a UTF-8 byte sequence. This transformation (known as marshaling) is straightforward since both C# and Java provide UTF-8 support in their standard libraries. For example, the Ice run time serializes a Java string to UTF-8 with:

```
// Java
byte[] buf = str.getBytes("UTF8");
```

Likewise, when an Ice program receives a C# or Java string, the Ice run time unmarshals the UTF-8 sequence into a native string object. This is also straightforward in C# and Java. For example, the Ice run creates a C# string from a UTF-8 byte sequence with:

```
// C#
System.Text.UTF8Encoding utf8 =
    new System.Text.UTF8Encoding();
string str = utf8.GetString(buf);
```

UTF-8 Serialization with C++

In C++, the standard-compliant way to perform character encoding conversion is to use a `codecvt` facet, as shown below:

```
// Convert a string to UTF-8
void toUTF8(const string& str, ostream& os)
{
    // Somehow retrieve the proper codecvt object
    codecvt<char, char, mbstate_t>*
        converter = ...;

    // Pass the converter to the output stream
    // through a locale
    os.imbue(locale(locale(), converter));

    // Converts from the internal char encoding
    //to UTF-8
    os << str;
}

// Convert a wstring to UTF-8
void toUTF8(const wstring& str, wostream& os)
{
    // Somehow retrieve the proper codecvt object
    codecvt<wchar_t, char, mbstate_t>*
        converter = ...;

    // Pass the converter to the output-stream
    // through a locale
    os.imbue(locale(locale(), converter));

    // Converts from the internal wchar_t encoding
    // to UTF-8
    os << str;
}
```

Unfortunately, there are several issues in this seemingly simple code. For one, where does the converter object come from? All the `codecvt` libraries that I am aware of (such as the Dinkumware Compleat Libraries) provide converters for narrow-to-wide conversion and vice-versa , but not for narrow-to-narrow conversion between different encodings, which is what we need. Therefore, in order to convert a narrow string to a sequence of UTF-8 bytes (or vice-versa), we have two choices:

- We can perform two conversions, each with its own library-provided `codecvt` object: we first convert the string into wide characters and then convert these wide characters into a sequence of UTF-8 bytes. (This approach is described in an article by Martin Sebor.) However, the double conversion is expensive and not an option for high-performance middleware such as Ice.
- We can provide our own `codecvt` object that performs the narrow-to-narrow conversion directly.

To convert wide strings to UTF-8, libraries of `codecvt` facets are available, but getting the proper `codecvt` object is difficult: the C++ standard does not mandate UTF-8 support, so there is no standard wide-character-to-UTF-8 `codecvt` (or even a standard "UTF-8" locale). Even though the C++ standard leaves the character encoding form of `wchar_t` as implementation-defined, in practice, the encoding is usually UTF-16 or UTF-32 (depending on the size of `wchar_t`). In particular, `wchar_t` is always encoded using UTF-16 on Windows, and is always encoded using UTF-32 on Linux. However, on Solaris and AIX, `wchar_t` uses a non-UTF encoding for a few Asian locales.

On a system with UTF-8 locales, you would typically build a C++ `locale` with the desired `codecvt<wchar_t, char, mbstate_t>` facet—one that converts between UTF-16/32 and UTF-8—with the following code:

```
// C++
locale loc("en_US.UTF-8");
```

If this does not work, there are several other ways to obtain such a facet. With the Boost library, we could use:

```
// C++
codecvt<wchar_t, char, mbstate_t>* converter =
    new boost::utf8_codecvt_facet<wchar_t>;
```

With the RogueWave C++ Standard Library, we would use:

```
// C++
codecvt<wchar_t, char, mbstate_t>* converter =
    new std::codecvt_byname("UTF-8@UCS");
```

Finally, another issue with UTF-8 serialization is the stream object itself: the only standard C++ streams that perform character encoding conversion are file streams (`fstream`), but string streams (`stringstream`) do not use the `codecvt` facet at all. As a result, the previously presented `toUTF8` functions fail miserably if we pass a `stringstream` instead of an `fstream`! Therefore, in order to use the nice stream syntax to write a `string` or `wstring` into a UTF-8 byte buffer, we would need to write our own stream class. While possible, this is a lot of extra work.

Overall, these issues make `codecvt` facets a poor choice for UTF-8 serialization in Ice (or any high-performance, portable C++ program), so we decided to use the much simpler mechanism described below.

Default UTF-8 Serialization with Ice for C++

Prior to Ice 3.1, Ice for C++ did not perform any UTF-8 conversion during marshaling or unmarshaling. The Slice `string` type was always mapped to `std::string`. Strings handed to the application were encoded using UTF-8, and strings passed to the Ice runtime (for example, as operation parameters) were assumed to be encoded using UTF-8 as well.

This behavior (which is also the default in the current Ice release) is simple and efficient: Ice can simply copy strings to and from its marshaling buffer without any conversion. In English locales, the characters used in strings are just ASCII (a subset of UTF-8), so everything works fine. This also works well in UTF-8 locales, which are becoming more and more common. (UTF-8 is the default on recent Linux distributions.) The default behavior is also appropriate for programs that perform little string processing.

Ice 3.1 adds the ability to map Slice strings to `std::wstring` through metadata. For example:

```
// Slice
interface Printer
{
    // In C++, maps string to std::wstring
    void print(["cpp:type:wstring"] string msg);
};
```

It is important to keep in mind that the metadata directive (like all metadata directives) is not part of the Slice contract: metadata provides language-mapping flexibility, but does not alter the on-the-wire representation of data. For example, a C++ client could use the preceding definition to pass a `std::wstring` to a server. This server must use the same contract, but it can use the "same" Slice definitions with different metadata to receive the message as a C++ narrow string:

```
// Slice
interface Printer
{
    // In C++, maps string to std::string
    void print(string msg);
};
```

Naturally, the server can also be implemented in Java (and receive the message as a `java.lang.String`) or in any other programming language supported by Ice.

By default, Ice expects C++ `wstring`s to be encoded using UTF-16 or UTF-32 (depending on the size of `wchar_t`), and the runtime, during marshaling and unmarshaling, automatically converts UTF-8 byte sequences to and from UTF-16/32-encoded `wstring`s. As discussed previously, this is the proper encoding and desired behavior all the time on some platforms, and most of the time on other platforms. (Currently, Ice uses the UTF converter implementation from unicode.org for these conversions.)

Custom UTF-8 Serialization with Ice for C++

While the default UTF-8 serialization offered by Ice works well most of the time, there are some situations where it is not ideal. For example, if you are augmenting an existing application that uses `std::string` (or `char` buffers) encoded in ISO Latin-1, it is inconvenient to have to convert to and from UTF-8 with each Ice API call. Likewise, if your locale is one of the few Asian locales that do not use UTF-16 or UTF-32 for `wchar_t`, the default `wstring` handling is not appropriate.

If you are in such a situation, you can register your own `string` and/or `wstring` converter with Ice. For example, Ice uses the registered narrow string converter each time it marshals a `std::string` to UTF-8 or unmarshals a UTF-8 byte sequence into a `std::string`. Ice also uses the registered narrow string converter to read property files during initialization (because Ice property files are encoded using UTF-8), and Ice uses the registered narrow string converter in its implementation of `stringToIdentity`, `identityToString`, `stringToProxy`, and `proxyToString`: for these strings, non-ASCII characters are encoded using "stringified UTF-8 bytes". For example, the Greek letter Π is represented as `"\316\240"` in a stringified identity. (The octal values 316 and 240 are the two UTF-8 bytes for the letter Π.)

`StringConverter` and `WstringConverter` are abstract base classes declared in the `Ice/StringConverter.h` header:

```
// C++
class UTF8Buffer
{
public:
    virtual Byte* getMoreBytes(size_t howMany,
                               Byte* firstUnused) = 0;
    virtual ~UTF8Buffer() {}
};

template<typename charT>
class BasicStringConverter :
        public IceUtil::Shared
{
public:
    virtual Byte* toUTF8(const charT* sourceStart,
                         const charT* sourceEnd,
                         UTF8Buffer&) const = 0;

    virtual void fromUTF8(const Byte* sourceStart,
                          const Byte* sourceEnd,
      std::basic_string<charT>& target) const = 0;
};

typedef BasicStringConverter<char>
    StringConverter;
typedef IceUtil::Handle<StringConverter>
    StringConverterPtr;

typedef BasicStringConverter<wchar_t>
    WstringConverter;
typedef IceUtil::Handle<WstringConverter>
    WstringConverterPtr;
```

The `UTF8Buffer` object passed by Ice to your `toUTF8` implementation is a simple wrapper around the internal Ice marshaling buffer. This allows `toUTF` to write directly into the Ice marshaling buffer, and avoid a memory allocation during marshaling. The pointer returned by `toUTF8` must be the first unused byte in the last buffer returned by `UTF8Buffer::getMoreBytes`. (See the Ice Manual for further details.)

To install a custom `string` or `wstring` converter object, you set the corresponding data member in the `InitializationData` parameter of `Ice::initialize`. For example:

```
// C++
InitializationData initData;
initData.stringConverter =
    new Latin1StringConverter;
CommunicatorPtr communicator =
    Ice::initialize(argc, argc, initData);
```

Using UTF-8 Encoded Strings with C++

Before you write your own narrow string converter, I recommend that you evaluate how practical (or impractical) the UTF-8 encoding is for your application. UTF-8 is a multi-byte encoding, but a number of `std::string` functions work on `char` elements, not complete Unicode characters. For example, the following code will print 2 because the UTF-8 encoding of Π uses two bytes:

```
// C++
// A string that contains just 'Π' encoded
// using UTF-8
std::string pi = ...;
cout << pi.size() << endl;
```

This could mean trouble if you parse strings naively. For example, do not do the following because you could split a single UTF-8 encoded character in two:

```
// C++: don't do this!
std::string str = ...; // Some random UTF-8
                       // encoded string
std::string firstPart =
    str.substring(0, str.size() / 2);
```

However, a major feature of UTF-8 is its ASCII-compatibility: all bytes in a UTF-8 encoded string that have values in the ASCII range actually are ASCII characters. For example, if you search for 'a' in a random UTF-8 string, you don't have to worry about accidentally matching the second or third byte of a multi-byte character because, in a multi-byte UTF-8 character, the most-significant bit is set in all bytes.

Splitting a string using ASCII-delimiters works for UTF-8 strings just as it does for plain ASCII strings. (I will address interaction with terminals through `cout`, `cerr`, and `cin` shortly.)

```
// C++
// Works well with UTF-8 strings!
// Some random UTF-8 encoded string
std::string str = ...;
size_t pos = str.find(';');
std::string firstPart =
    (pos != std::string::npos)
        ? str.substr(0, pos)
        : str;
```

You should also be careful when passing UTF-8 encoded strings to system calls. On Windows, there are often two variants for each system call: one that expects a C-string encoded using the current ANSI code page, and one that takes a UTF-16 encoded wide string.

For example, to create a directory on Windows, you have several choices: `_mkdir`, `_wmkdir`, and `CreateDirectory`. If you have a UTF-8 encoded string that may contain non-ASCII characters, you should use the Unicode version of these system calls:

```
// C++
// UTF-8 encoded string with possibly non-ASCII
// characters
std::string dirName = ...;
_wmkdir(IceUtil::stringToWstring(
    dirName).c_str());
// Or define _UNICODE and call:
CreateDirectory(IceUtil::stringToWstring(
    dirName).c_str(), 0);
```

Please note that the `IceUtil::stringToWstring` function always converts from UTF-8 to UTF-16/32. Likewise, `IceUtil::wstringToString` always converts from UTF-16/32 to UTF-8. These functions ignore the locale setting and do not use the Ice string converters.

`iconv` String Converter Example

`iconv` is the standard character encoding conversion facility on Linux and UNIX, typically provided as part of the C library. On a Linux or UNIX system, you can use `iconv -l` to list the character encoding schemes supported on your system.

The `IconvStringConverter` template class uses these standard `iconv` functions to implement the `Ice::StringConverter` and `Ice::WSstringConverter` classes. You just need to give the constructor the name of your internal character or wide character encoding, for example:

```
// C++
// The internal narrow character encoding is
// ISO-Latin-1
initData.stringConverter =
    new IconvStringConverter<char>("ISO-8859-1");

// The internal wide character encoding is
// WCHAR_T, i.e. the fixed  wchar_t encoding on
// the local system (this works on Linux but not
// Solaris or AIX)
initData.stringConverter =
    new IconvStringConverter<wchar_t>("WCHAR_T");
```

To illustrate how all this works, let's use the Glacier2 chat demo to send messages between different GNOME terminals on a Linux Fedora Core 5 system. The GNOME terminal (`gnome-terminal`) is ideal for experimenting with character encodings because it lets us select any encoding we like (using the *Terminal→Set Character Encoding* menu).

We modify the C++ client of this demo to register an instance of the `IconvStringConverter`; the `main` function becomes:

```
// C++
#include <IconvStringConverter.h>
// ...

int
main(int argc, char* argv[])
{
    Ice::InitializationData initData;
    if(argc > 1)
    {
        cout << "Installing iconv string "
            << "converter for" << argv[1] << endl;
        try
        {
            initData.stringConverter =
                new IconvStringConverter<char>(
                    argv[1]);
        }
        catch(const IceUtil::Exception& ex)
        {
            cerr << "IconvStringConverter "
                << "creation" failed:\n"
                << ex << endl;
            return 1;
        }
    }

    //
    // We create the properties with the narrow
    // string converter (or null, when not set)
    // in case config.client contains any
    // non-ASCII character. The config.client file
    // that ships with this demo is pure ASCII.
    //
    initData.properties = Ice::createProperties(
        initData.stringConverter);
    initData.properties->load("config.client");

    ChatClient app;
    return app.main(argc, argv, initData);
}
```

With this change, we can pass the desired string encoding as a command-line argument. For example, to start a client in ISO Latin-1 mode, we run:

```
$ ./client ISO-8859-1
```

This installs a narrow string `IconvStringConverter` for ISO 8859-1, and Ice will treat all strings in memory as encoded in ISO 8859-1. (ISO 8859-1 is the official name for ISO Latin-1.) When the client is started without a command-line argument, no string converter is registered with Ice.

Note that going through Glacier2 is not important here—we just want a chat demo, and the only chat demo among the Ice demos happens to use Glacier2.

First, let's see what happens if we do not register a string encoder. We'll use the famous line from Marcel Pagnol's movie "Marius": *"tu me fends le cœur"* (you're breaking my heart).

Figure 1 shows César sending his message using an ISO 8859-15 terminal. ISO 8859-15 (or ISO Latin-9) is an encoding with support for some French characters, in particular 'œ'. The message goes to the server (through Glacier2), comes back to the client and is displayed correctly because no conversion takes place. The message travels encoded in ISO 8859-15; technically, this is a violation of the Ice protocol, but so far without any consequence.

Figure 1: Client Using an ISO 8859-15 Terminal without a String Converter

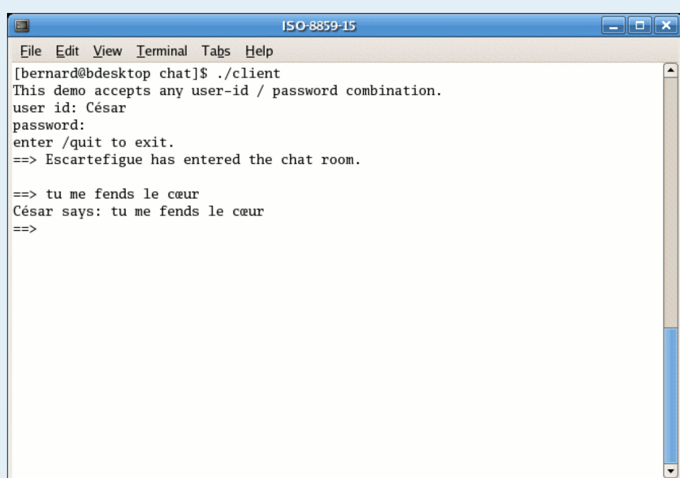
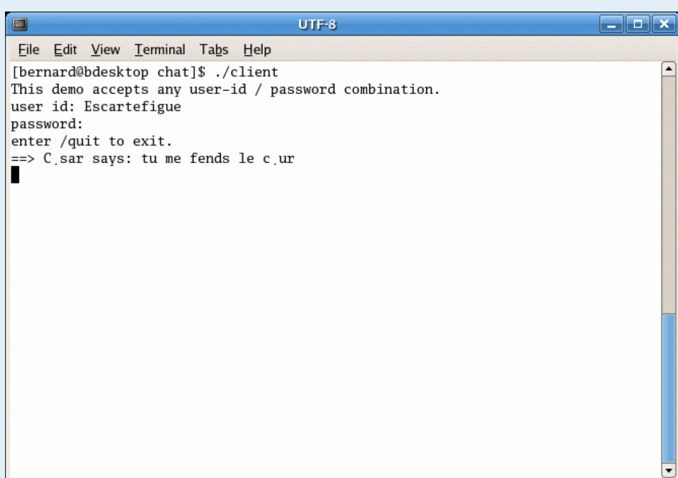


Figure 2 shows our control client, Escartefigue, running in a UTF-8 terminal. No string converter is needed for Escartefigue because UTF-8 is the default narrow string encoding for Ice. The message

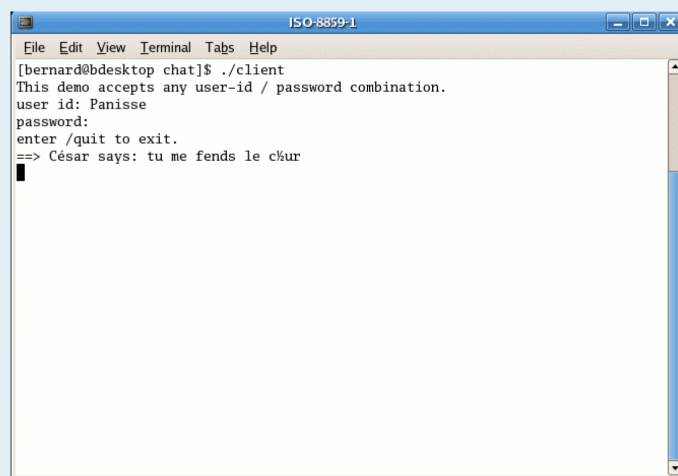
Figure 2: Client Using a UTF-8 Terminal



from César arrives with the two non-ASCII characters garbled because it incorrectly uses ISO 8859-15 to encode the strings on the wire.

Figure 3 shows one more client, Panisse, using an ISO Latin-1 (ISO 8859-1) terminal. The 'é' of César is printed correctly because 'é' is coded as 0xE9 in both ISO 8859-1 and ISO 8859-15. However, the 'œ' is displayed as '½' because both are encoded as 0xBD in their respective ISO Latin encodings.

Figure 3: Client Using an ISO 8859-1 Terminal without a String Converter



Now, let's see what happens when we install a narrow string converter. Figure 4 shows that everything is still working well for César using the ISO 8859-15 terminal.

Figure 4: Client Using an ISO 8859-15 Terminal with a Correct String Converter

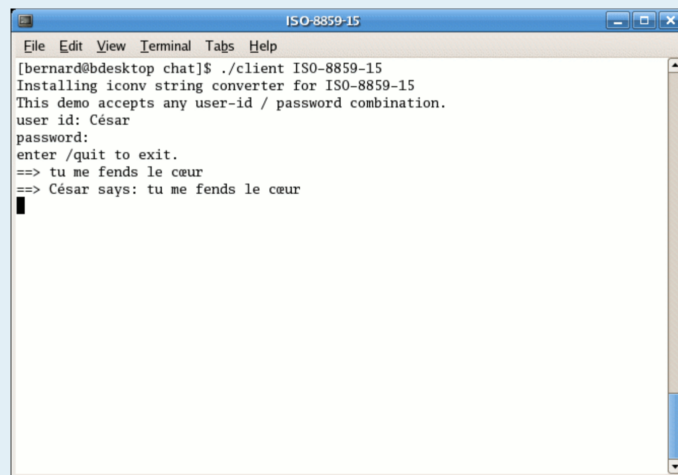


Figure 5 shows Escartefigue, using the UTF-8 terminal, receiving the message correctly this time.

Figure 5: Client Using a UTF-8 Terminal

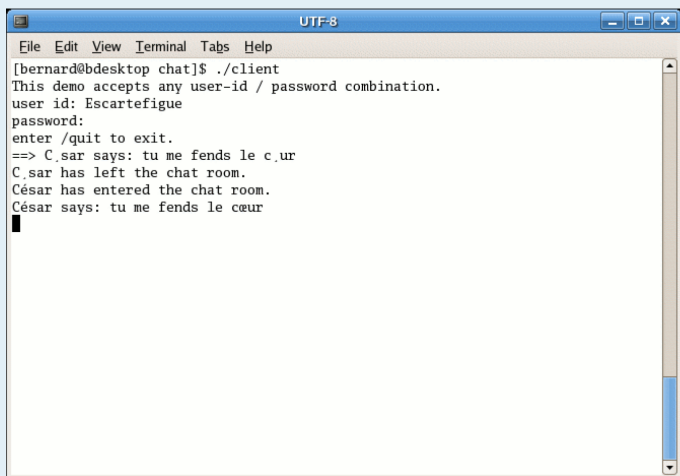
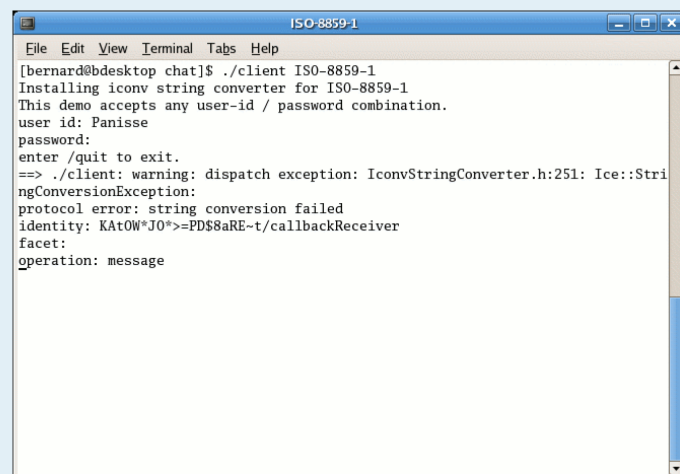


Figure 6 shows how `iconv` behaves when a character is not available in the target encoding (there is no 'œ' in ISO Latin-1): the string unmarshaling fails, and a warning is sent to the default logger (`stderr`).

Figure 6: Client Using an ISO 8859-1 Terminal with a Correct String Converter



Conclusion

Most of the time, you can pass strings in and out of Ice and everything just works—you don't need to worry about character encoding. In C++, Ice provides a default behavior that is efficient and correct for most applications.

With previous Ice for C++ releases, if you used non-Unicode encodings, you were forced to make explicit conversion calls around each operation that transmitted a string in order to convert between the native encoding and the on-the-wire UTF-8 encoding. As of Ice 3.1, this is no longer necessary: `string` and `wstring` can use any encoding you choose—as long as you register the corresponding UTF-8 converters with the Ice runtime, the necessary conversion happens transparently and without explicit involvement of the application code.

IceGrid Sessions and Resource Allocation

Matthew Newhook, Senior Software Engineer

Introduction

This article demonstrates how to use one of the important new features introduced in Ice 3.1: IceGrid sessions and the associated resource allocation framework. The IceGrid chapter in the Ice Manual uses an MP3 encoding application to illustrate the use of various IceGrid features. This article shows how to implement an MP3 encoder, and how to coordinate access to the available encoders using IceGrid sessions.

Here is the interface we will implement for this article:

```
//Slice
module Ripper
{

    exception EncodingFailedException
    {
        string reason;
    };

    sequence<short> Samples;

    interface Mp3Encoder
    {
        Ice::ByteSeq encode(Samples leftSamples,
                Samples rightSamples)
            throws EncodingFailedException;
        Ice::ByteSeq flush()
            throws EncodingFailedException;
    };

    interface Mp3EncoderFactory
    {
        Mp3Encoder* createEncoder(int channels,
            int sampleRate);
    };
};
```

This is the same interface as presented in the Ice manual with the exception of the `createEncoder` method, which takes two additional parameters: the number of channels to encode, and the sample rate in Hertz.

Consider a grid that is dedicated to encoding MP3 files. Given that the process of MP3 encoding is primarily CPU-bound, to make best use of resources, each machine should concurrently encode only as many files as it has CPUs. (To account for bandwidth limitations, we might also use CPUs*1.5, or a similar factor.) Each encoding client should use only one MP3 encoder factory, from which it creates an encoder for each WAV file to be encoded. We do not want more files to be encoded concurrently than we have CPU resources because this would be inefficient. Also, to keep this demo simple, we do not want a single client to use more than one encoding factory. While a factory is in use, no other client must be able to use the same factory.

In order to implement this requirement, there must be some service that keeps track of the used and unused encoder factories. Clients must request a factory from this service and inform the service when they have completed their work with the factory. Fortunately, IceGrid now provides such a service out of the box—the IceGrid allocation mechanism.

IceGrid Sessions

Let us briefly review this new feature. Starting with Ice 3.1, IceGrid provides a resource allocation facility that coordinates access to hosted servers and objects. To gain access to this feature, the client must first authenticate itself with IceGrid. If authentication is successful, IceGrid allocates a session object that it returns to the client. While the session is alive, the client can allocate objects or servers for its exclusive use. Alternatively, authentication and session establishment can occur through Glacier2 if additional security is desired. I will explain how to integrate with Glacier2 in a future article; for now, we will establish the session directly with IceGrid.

```
// Slice
module IceGrid
{
    exception PermissionDeniedException
    {
        string reason;
    };

    interface Registry
    {
        Session* createSession(string userId,
                string password)
            throws PermissionDeniedException;

        Session*
        createSessionFromSecureConnection()
            throws PermissionDeniedException;

        nonmutating int getSessionTimeout();
    };
};
```

There are two ways to create a session: by providing a user name and password, and by using the credentials of an SSL connection. To create a session using a user name and password, we call `createSession`; to create a session using an SSL connection, we call `createSessionFromSecureConnection`. If IceGrid cannot create a session, it raises `PermissionDeniedException`. In order to enable session creation, you must configure IceGrid with the proxy of a permissions verifier. For the `createSession` call, you must provide an implementation of a `Glacier2::PermissionsVerifier` and configure IceGrid with the property

IceGrid.Registry.PermissionsVerifier; for the createSessionFromSecureConnection call, you must provide an implementation of a Glacier2::SSLPermissionsVerifier and configure IceGrid with the property IceGrid.Registry.SSLPermissionsVerifier. (For development purposes, IceGrid provides a permissions verifier for the createSession call that accepts any user name and password. I will show how to use and configure this verifier shortly.)

Here is IceGrid's Session interface:

```
// Slice
module IceGrid
{
    exception ObjectNotRegisteredException
    {
        Ice::Identity id;
    };

    exception AllocationException
    {
        string reason;
    };

    exception AllocationTimeoutException
        extends AllocationException
    {
    };

    interface Session extends Glacier2::Session
    {
        idempotent void keepAlive();

        Object* allocateObjectById(
                Ice::Identity id)
            throws ObjectNotRegisteredException,
                   AllocationException;

        Object* allocateObjectByType (string type)
            throws AllocationException;

        void releaseObject(Ice::Identity id)
            throws ObjectNotRegisteredException,
                   AllocationException;

        idempotent void setAllocationTimeout(
            int timeout);
    };
};
```

A client must periodically call keepAlive while it uses its session; without these calls, IceGrid assumes that the client has terminated unexpectedly or is otherwise misbehaved, and automatically destroys the session. You can call getSessionTimeout on the IceGrid::Registry interface to obtain the value of the session timeout in seconds.

The methods allocateObjectById and allocateObjectByType allocate objects for the exclusive use of a client, until the client releases the object (by calling releaseObject) or destroys the session. These two methods block until an object is available for allocation, or until the value established by setAllocationTimeout is reached. The default timeout value is −1, meaning no timeout. If IceGrid cannot allocate an object within the given timeout period, the operations raise AllocationTimeoutException. If IceGrid does not have an object with the given identity available, the operations raise ObjectNotRegisteredException. If an allocation fails for some other reason, the operations raise AllocationException. (See the Ice Manual for more details on this exception.) If a client calls allocateObjectByType, IceGrid may have several of that type; in that case, IceGrid chooses an object at random.

In addition to allocating objects, a client can also allocate an entire server for its exclusive use. This is useful for security purposes; I will cover this feature in a future article.

Client

After parsing the command line arguments, the client must contact the IceGrid registry to create a session as follows:

```
// C++
IceGrid::RegistryPrx registry =
    IceGrid::RegistryPrx::checkedCast(
        communicator()->stringToProxy(
            "EncoderIceGrid/Registry"));
IceGrid::SessionPrx session;
try
{
    session = registry->createSession(
        "foo", "bar");
}
catch(const IceGrid::PermissionDeniedException&ex)
{
    cout << "permission denied:\n" << ex.reason
        << endl;
    return 1;
}
```

Note that the session's user name and password can be anything (other than the empty string). Once the client has created the session, it must periodically refresh the session. We will create a thread to do this:

```
// C++
SessionRefreshThreadPtr refresh =
    new SessionRefreshThread(
IceUtil::Time::seconds(
    registry->getSessionTimeout()/2), session);
refresh->start();
```

Note that we call getSessionTimeout to obtain the interval at which the session must be kept alive. We use half the timeout in between calls to refresh to ensure that at least one keepAlive message is sent before the timeout kicks in, and to allow for some network latency. For brevity, I do not show the implementation of the refresh thread here; you can find the implementation in the source code for this article.

The next step is to allocate an encoder for exclusive use by our client by calling `allocateObjectByType`. Note that there could be many encoder factories available, in which case the call will return one of the factories selected at random. In the event that all factories are in use by other clients, the call will block (potentially indefinitely because we have not set an allocation timeout) until a factory becomes available.

```
// C++
Mp3EncoderFactoryPrx obj =
    session->allocateObjectByType(
        Mp3EncoderFactory::ice_staticId());
try
{
    Mp3EncoderFactoryPrx factory =
        Mp3EncoderFactoryPrx::checkedCast(obj);
    // Remainder of implementation here...
}
catch(const Exception& e)
{
    cerr << "Exception: " << e << endl;
}

session->releaseObject(obj->ice_getIdentity());

refresh->destroy();
refresh->getThreadControl().join();

session->destroy();
```

First we allocate the object and cast it to the correct type. Note that the code is careful to do this within a try-catch block. This ensures that, if the object is of the wrong type or is inaccessible, we correctly release the object and clean up the session. In addition, the code destroys the refresh thread and joins with it.

The client encodes each input file using a newly created encoder object. The client uses libsndfile 1.0.15 to read the contents of the WAV files. For brevity, I have omitted the code to actually read the file as well as a few other details; please consult the code for this article for the full listing.

```
// C++
Mp3EncoderPrx encoder =
    factory->createEncoder(channels, samplerate);
ByteSeq bytes;
while(true)
{
    Samples lbuf(nsamples), rbuf(nsamples);
    // Read the samples from the source media.
    if(/*no data*/)
    {
        break;
    }
    bytes = encoder->encode(lbuf, rbuf);
    // Write bytes to output media.
}

bytes = encoder->flush();
// Write bytes to output media.
```

The client first creates an encoder object, and then reads and encodes samples from a sound file and writes the corresponding encoded output to an MP3 file. Once all available data has been read and encoded, the client calls `flush`, which gets one more buffer of data and destroys the encoder object. The client writes this final buffer to the MP3 file before closing the file.

Server

The server implementation consists of two components, the encoder factory and the MP3 encoder. First we'll look at the MP3 encoder factory.

```
// Slice
interface Mp3EncoderFactory
{
    Mp3Encoder* createEncoder(
        int channels, int sampleRate);
};
```

We do not want more encoders to run in parallel than we have CPUs because MP3 encoding is primarily CPU bound: if more encoders run than we have CPUs (or some factor thereof), the system will not make the best use of computing resources. This means that we will provide a limited number of factories according to the number of CPUs, with each factory allowing only one file to be encoded at a time; each factory is for the exclusive use of a single client until that client no longer uses that factory.

We will use the IceGrid session allocation mechanism to parcel out factories to clients. Once a client has obtained an encoder factory, the client creates an encoder to encode the MP3 data. On each encoding machine, we will deploy a limited number of encoder factory services, according to the number of CPUs on that machine. Because each client can encode only one file at a time using a particular factory, this effectively limits the number of concurrent encodings in the server.

Each encoder factory service will be deployed into a separate server. Since we do not permit concurrent encoding using a single factory, each server will be single threaded, meaning that we do not need any concurrency protection. At this point, I am sure you are wondering why we are not deploying multiple factories in a single server. The answer is two-fold. Firstly, the IceGrid security model does not support this—I will discuss the exact details in a future article. The second reason relates to the deployment descriptor, and I will return to this topic shortly, when we examine how to deploy the application.

```
// C++
class Mp3EncoderFactoryI :
    public Mp3EncoderFactory
{
public:

    virtual Mp3EncoderPrx
    createEncoder(int channels, int rate,
                  const Current&)
```

```
    {
        if(_proxy)
        {
            try
            {
                _proxy->flush();
            }
            catch(const ObjectNotExistException&)
            {
                // Ignore
            }
            _proxy = 0;
        }

        ObjectPtr encoder =
            new Mp3EncoderI(channels, rate);
        _proxy = Mp3EncoderPrx::uncheckedCast(
            c.adapter->addWithUUID(encoder));
        return _proxy;
    }

private:

    Mp3EncoderPrx _proxy;
};
```

First, the factory calls `flush` on the encoder to destroy it. This is necessary because the same (or another) client may previously have used this factory instance but neglected to call `flush`, in which case the previous encoder still exists. (If the previous client *did* call `flush`, no harm is done: in that case, the call to `flush` raises an `ObjectNotExistException` that the code ignores.) Next, the factory allocates a new encoder, adds the encoder to the object adapter, and returns the encoder's proxy to the client.

Note that this design allows a malicious client to hold onto the proxy to a factory and, once another client has acquired its proxy to the same factory, call `createEncoder` a second time, thereby destroying the other client's encoder while the encoder is still in use. For now, we will ignore this issue—in a future article, I will discuss how to use Glacier2 to prevent this scenario.

Next we look at the implementation of the encoder object. This is mostly straightforward. I have used LAME version 3.97b2 to implement the MP3 encoding; however, for brevity, I have omitted many of the details of how to use LAME. (Please consult the code for this article for the details.) The class definition for the encoder is as follows:

```
// C++
class Mp3EncoderI : public Mp3Encoder
{
public:

    Mp3EncoderI(int, int);
    ~Mp3EncoderI();

    virtual ByteSeq encode(const Samples&,
        const Samples&, const Current&);
    virtual ByteSeq flush(const Current&);

private:

    ByteSeq _data;
    // LAME-specific details here...
};
```

The class keeps a `ByteSeq` buffer in memory to avoid continuously allocating and deallocating the encoding buffer.

The constructor and destructor set up and tear down the LAME encoder. The implementation is as follows:

```
 // C++
Mp3EncoderI::Mp3EncoderI(int channels, int rate)
{
    // Set up LAME here...
}

Mp3EncoderI::~Mp3EncoderI()
{
    // Clean up LAME here...
}
```

Note that it is the destructor that cleans up the resources allocated by the LAME library, instead of `flush`. (For a summary of the life cycle issues that can arise during resource clean-up, have a look at Michi's article "The Samsara of Objects" in Issue 11 of *Connections*.)

Next we examine the implementation of the `encode` method.

```
// C++
ByteSeq
Mp3EncoderI::encode(const Samples& leftSamples,
                    const Samples& rightSamples,
                    const Current& current)
{
    long nsamples = leftSamples.size();
    unsigned long sz =
        static_cast<unsigned long>(
        1.25f*nsamples + 7200);
    if(_data.capacity() < sz)
    {
        _data.reserve(sz);
    }
    _data.resize(sz);
    int n = lame_encode_buffer(_gfp,
        &leftSamples[0], &rightSamples[0],
        nsamples, &_data[0], _data.size());
    if(n < 0)
    {
        throw EncodingFailedException();
    }
    _data.resize(n);
    return _data;
}
```

Next, the code calculates a preliminary size of the amount of data to be encoded. This estimate comes from the LAME documentation—the maximum number of bytes occupied from encoding `n`

samples is `1.25*n + 7200`. The code keeps the `_data` buffer around to avoid excessive memory allocation and deallocation. The call to `lame_encode_buffer` returns the actual number of bytes encoded thus, before returning `_data` to the caller, the code resizes the buffer to the correct number of bytes.

The implementation of `flush` is as follows:

```
// C++
ByteSeq
Mp3EncoderI::flush(const Current& current)
{
    current.adapter->remove(current.id);

    // Flush MP3 data here...
    return _data;
}
```

The code simply removes the servant from the object adapter. Note that, to do this, we need not store the hosting object adapter or the object identity inside the servant because these values are available from the `Current` object. Finally, the code flushes the encoder and returns the result.

Configuration and Deployment

In order to deploy our encoder application as a service, we require an IceGrid configuration and deployment descriptor. For the purposes of this article, we'll be running everything on a single node, so we deploy the IceGrid node and registry as a single process:

```
# IceGrid Configuration
IceGrid.Node.CollocateRegistry=1
```

Next, we need to settle on a style of session authentication (user name and password or SSL authentication). Since I want to leave securing the encoder for a later article, we use name and password authentication with `createSession`. In addition, we will use a null permissions verifier that is provided with IceGrid. This permissions verifier accepts any user name and password. For this example, the IceGrid instance name of the application is `EncoderIceGrid`, leading to the following configuration. (You can see the complete configuration in the accompanying source code.)

```
# IceGrid Configuration
IceGrid.InstanceName=EncoderIceGrid
IceGrid.Registry.PermissionsVerifier=EncoderIceGri
d/NullPermissionsVerifier
Ice.Default.Locator=EncoderIceGrid/Locator:default
-p 12000
```

We also need a deployment descriptor for the encoder. We'll first create a template for the encoder factory. Most of this should look familiar if you have used IceGrid before. (If not, have a look at the deployment descriptor section of the IceGrid chapter in the Ice Manual, or read the articles by Benoit Foucher and me in Issue 9 of *Connections*.)

We will host the implementation of the encoder factory in an IceBox service. What follows is a template for the MP3 encoder service.

```
<server-template id="Mp3Encoder">
  <parameter name="instance-name"/>
  <parameter name="host"/>
  <icebox id="${instance-name}" exe="icebox"
    activation="on-demand">
    <service name="${instance-name}"
     entry="Mp3EncoderService:create">
      <adapter
        name="${instance-name}-EncoderFactory"
        endpoints="tcp -h ${host}">
        <allocatable
          identity="${instance-name}-Mp3Encoder"
          type="::Ripper::Mp3EncoderFactory"/>
      </adapter>
    </service>
  </icebox>
</server-template>
```

The magic in this descriptor that makes the whole process work is the `allocatable` element. This defines an object with a given identity and type as an allocatable object. We have to ensure that the encoder implementation hosts an Ice object with this identity.

The final piece is the deployment of two instances of the encoder on the `localhost` node.

```
<icegrid>
  <application name="Mp3Ripper">
    <node name="localhost">
      <server-instance template="Mp3Encoder"
        instance-name="localhost-encoder"
        host="localhost"/>
      <server-instance template="Mp3Encoder"
        instance-name="localhost-encoder2"
        host="localhost"/>
    </node>
  </application>
</icegrid>
```

The service template also illustrates why we cannot deploy multiple factories in a single server: note that, for each encoder factory, we need an `allocatable` element. However, there is no way to provide a variable number of `allocatable` elements in a template. If there is no `allocatable` element for each of the additional factories then the object may as well not exist since it can never be allocated by an IceGrid client. The limitation is that we have to use a constant number of factories in each template definition. What number of factories should we pick? This could vary based on your deployment environment. Therefore, our template definition defines one factory, and you deploy as many instances of this template on each node as you have CPUs.

Service Implementation

Now that we have a deployment descriptor, we can write the service:

```
// C++
class Mp3EncoderServiceI : public Service
{
public:

    virtual void
    start(const string& name,
        const CommunicatorPtr& communicator,
        const StringSeq& args)
    {
        _adapter =
            communicator->createObjectAdapter(
                name + "-EncoderFactory");
        _adapter->add(
            new Mp3EncoderFactoryI(name),
            communicator->stringToIdentity(
                name + "-Mp3Encoder"));
        _adapter->activate();

    }

    virtual void
    stop()
    {
        _adapter->deactivate();
    }

private:

    ObjectAdapterPtr _adapter;
};
```

The `start` method creates the object adapter as defined in the IceGrid template.

```
<adapter name="${instance-name}-EncoderFactory"
  endpoints="tcp -h ${host}">
```

The encoder factory is registered with the same identity as the identity in the IceGrid template:

```
<allocatable
  identity="${instance-name}-Mp3Encoder"
  type="::Ripper::Mp3EncoderFactory"/>
```

A common source of errors is to define one thing in the deployment descriptor, and then do something different in your application. The names and object identities used in the application must match those in the deployment descriptor; otherwise you will get hard to diagnose errors.

The `stop` method deactivates the object adapter, which causes the destructor of the encoder factory to run. In turn, the destructor cleans up all allocated resources.

Finally, we need to create a service entry point that allocates a new encoder service:

```
extern "C"
{

MP3ENCODERSERVICE_SERVICE_API IceBox::Service*
create(CommunicatorPtr communicator)
{
    return new Mp3EncoderServiceI;
}

}
```

Conclusion

There you have it—with only a little code, we have developed a full MP3 encoding solution. Furthermore, by taking advantage of the IceGrid session mechanism, we have a solution that should allow us to maximize the use of the available computing resources without having to write complex and error-prone scheduling and queuing code. In the next article, I will discuss how to secure access to the encoder factory.

FAQ Corner

In each issue of our newsletter, we present a few frequently-asked questions about Ice. The questions and answers are taken from our support forum at http://www.zeroc.com/vbulletin/ and deal with specific problems that developers tend to encounter, and for which the answer may not be readily apparent from reading the documentation. We hope that you will find the hints and explanations in this section useful.

Q: When should I use `__setNoDelete` with a C++ smart pointer?

Ice for C++ provides reference-counting smart pointers that ensure that your code does not leak memory. Once you assign the return value from `operator new` to a smart pointer, you no longer have to worry about memory leaks. For example:

```
// Slice
class Node
{
    Node next;
};
```

This defines a simple self-referential class that can be used to create a linked list:

```
// C++
{   // Open scope
    NodePtr p = new Node;
    p->next = new Node;
    // ...
} // Close scope, no leak here.
```

The reference counting that is performed by smart pointers takes care of deleting both `Node` instances once the variable `p` goes out of scope, even if the code leaves the scope in which `p` is defined due to an exception.

Smart pointers are instances of the `IceUtil::Handle` template. For example, the Slice compiler generates a type definition for `NodePtr`:

```
// C++
typedef IceUtil::Handle<Node> NodePtr;
```

The `Handle` template contains member functions that take care of maintaining the reference count. The template provides overloaded constructors, a copy constructor, a destructor, and overloaded assignment operators that ensure that the pointed-at object's reference count is maintained appropriately.

To maintain the reference count, these member functions of the `Handle` template expect the pointed-at object to provide two member functions, `__incRef` and `__decRef`, that increment and decrement the reference count, respectively. The pointed-at class provides these member functions by deriving from the `IceUtil::Shared` base class, which also contains `__getRef` and `__setNoDelete` member functions:

```
class Shared
{
public:
    Shared() : _ref(0), _noDelete(false) {}
    virtual ~Shared() {}
    void __incRef();
    void __decRef();
    void __getRef();
    void __setNoDelete(bool b)
    {
        _noDelete = b;
    }

private:
    int _ref; // Exact type varies with platform
    bool _noDelete;
};
```

The `__getRef` member function returns the value of the `_ref` member and is useful mainly for debugging. The implementation of `__incRef` increments the value of the `_ref` member, and `__decRef` decrements the value. Ignoring threading issues, `__decRef` is implemented something like this:

```
// C++
void
Shared::__decRef()
{
    if(--_ref == 0 && !_noDelete)
    {
        delete this;
    }
}
```

In other words, when the smart pointer calls `__decRef` on the pointed-at instance and the reference count reaches zero (which happens when the last smart pointer for a class instance goes out of scope), the instance self-destructs by calling `delete this`.

However, as you can see, the instance self-destructs only if `_noDelete` is false (which it is by default, because the constructor initializes it to false). You can call `__setNoDelete(true)` to prevent this self-destruction and, later, call `__setNoDelete(false)` to enable it again. Why would you want to do this? As it turns out, doing this is necessary if, for example, a class in its constructor needs to pass `this` to another function:

```
// C++
void
someFunction(const Handle<SomeClass>& h)
{
    // ...
}

// SomeClass derives from IceUtil::Shared
SomeClass::SomeClass()
{
    someFunction(this); // Trouble looming here!
}
```

At first glance, this code looks innocuous enough. While `SomeClass` is being constructed, it passes `this` to `someFunction`, which expects a smart pointer. No problem with that: the compiler constructs a temporary smart pointer at the point of call (because `Handle` has a single-argument constructor that accepts a pointer to a heap-allocated instance, so the constructor acts a conversion function). However, this code fails badly. The `SomeClass` instance is constructed with a statement such as:

```
// C++
SomeClassPtr p = new SomeClass;
```

The constructor of `SomeClass` is called by `operator new` and, when the constructor starts executing, the reference count of the instance is zero (because that is what the reference count is initialized to by the `Shared` base class of `SomeClass`). When the constructor calls `someFunction`, the compiler creates a temporary smart pointer, which increments the reference count of the instance and, once `someFunction` completes, the compiler dutifully destroys that temporary smart pointer again. But, of course, that drops the reference count back to zero and causes the `SomeClass` instance to self-destruct by calling `delete this`. The net effect is that the call to `new SomeClass` returns a pointer to an already deleted object! Of course, this is likely to cause havoc—a core dump is the most likely outcome, but the program might exhibit various difficult-to-diagnose forms of insanity before that happens.

To get around the problem, you can call `__setNoDelete`:

```
// C++
SomeClass::SomeClass()
{
    __setNoDelete(true);
    someFunction(this);    // OK
    __setNoDelete(false);
}
```

The code disables self-destruction while `someFunction` uses its temporary smart pointer by calling `__setNoDelete(true)`. By doing this, the reference count of the instance is incremented before `someFunction` is called and decremented back to zero when `someFunction` completes *without* causing the object to self-destruct. The constructor then re-enables self-destruction by calling `__setNoDelete(false)` before returning, so the statement

```
// C++
SomeClassPtr p = new SomeClass;
```

does the usual thing, namely to increment the reference count of the object to 1, despite the fact that a temporary smart pointer existed while the constructor ran.

Note that you may also come across a similar scenario if a derived class constructor throws an exception. This can happen, for example, if you use the listener pattern, for which a base class constructor passes `this` to the listener class, and an exception in the derived constructor can lead to double deallocation. You can use the same `__setNoDelete` mechanism to temporarily disable self-destruction in this case. In general, whenever a class constructor passes `this` to a function or another class that accepts a smart pointer, you must temporarily disable self-destruction. (Please check the client-side C++ mapping chapter in the Ice Manual for an example of how to do this for the listener pattern.)