C++ Mapping for Exceptions

On this page:

- C++ Mapping for User Exceptions
- C++ Default Constructors for Exceptions
- C++ Mapping for Run-Time Exceptions

C++ Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

These exception definitions map as follows:

C++

```
class GenericError: public Ice::UserException {
public:
    std::string reason;

    GenericError() {}
    explicit GenericError(const string&);

    virtual const std::string&  ice_name() const;
    virtual Ice::Exception*     ice_clone() const;
    virtual void                ice_throw() const;
    // Other member functions here...
};

class BadTimeVal: public GenericError {
public:
    BadTimeVal() {}
    explicit BadTimeVal(const string&);

    virtual const std::string&  ice_name() const;
    virtual Ice::Exception*     ice_clone() const;
    virtual void                ice_throw() const;
    // Other member functions here...
};

class BadZoneName: public GenericError {
public:
    BadZoneName() {}
    explicit BadZoneName(const string&);

    virtual const std::string&  ice_name() const;
    virtual Ice::Exception*     ice_clone() const;
    virtual void                ice_throw() const;
};
```

Each Slice exception is mapped to a C++ class with the same name. For each exception member, the corresponding class contains a public data member. (Since `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.)

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception has three additional member functions:

- `ice_name`
  As the name suggests, this member function returns the name of the exception. For example, if you call the `ice_name` member function of a `BadZoneName` exception, it (not surprisingly) returns the string `"BadZoneName"`. The `ice_name` member function is useful if you catch exceptions generically and want to produce a more meaningful diagnostic, for example:

  C++

  ```
  try {
      // ...
  } catch (const GenericError& e) {
      cerr << "Caught an exception: " << e.ice_name() << endl;
  }
  ```

  If an exception is raised, this code prints the name of the actual exception (`BadTimeVal` or `BadZoneName`) because the exception is being caught by reference (to avoid slicing).

- `ice_clone`
  This member function allows you to polymorphically clone an exception. For example:

  C++

  ```
  try {
      // ...
  } catch (const Ice::UserException& e) {
     Ice::UserException* copy = e.clone();
  }
  ```

  `ice_clone` is useful if you need to make a copy of an exception without knowing its precise run-time type. This allows you to remember the exception and throw it later by calling `ice_throw`.

- `ice_throw`
  `ice_throw` allows you to throw an exception without knowing its precise run-time type. It is implemented as:

  C++

  ```
  void
  GenericError::ice_throw() const
  {
      throw *this;
  }
  ```

  You can call `ice_throw` to throw an exception that you previously cloned with `ice_clone`.

Each exception has a default constructor. This constructor performs memberwise initialization; for simple built?in types, such as integers, the constructor performs no initialization, whereas complex types such as strings, sequences, and dictionaries are initialized by their respective default constructors.

An exception also has a second constructor that accepts one argument for each exception member. This constructor allows you to instantiate and initialize an exception in a single statement, instead of having to first instantiate the exception and then assign to its members. For derived exceptions, the constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

Note that the generated exception classes contain other member functions that are not shown here. However, those member functions are internal to the C++ mapping and are not meant to be called by application code.

All user exceptions ultimately inherit from `Ice::UserException`. In turn, `Ice::UserException` inherits from `Ice::Exception` (which is an alias for `IceUtil::Exception`):

C++

```
namespace IceUtil {
    class Exception {
        virtual const std::string& ice_name() const;
        Exception*                 ice_clone() const;
        void                       ice_throw() const;
        virtual void               ice_print(std::ostream&) const;
        // ...
    };
    std::ostream& operator<<(std::ostream&, const Exception&);
    // ...
}

namespace Ice {
    typedef IceUtil::Exception Exception;

    class UserException: public Exception {
    public:
        virtual const std::string& ice_name() const = 0;
        // ...
    };
}
```

`Ice::Exception` forms the root of the exception inheritance tree. Apart from the usual `ice_name`, `ice_clone`, and `ice_throw` member functions, it contains the `ice_print` member functions. `ice_print` prints the name of the exception. For example, calling `ice_print` on a `BadTimeVal` exception prints:

```
BadTimeVal
```

To make printing more convenient, `operator<<` is overloaded for `Ice::Exception`, so you can also write:

C++

```
try {
    // ...
} catch (const Ice::Exception& e) {
    cerr << e << endl;
}
```

This produces the same output because `operator<<` calls `ice_print` internally.

For Ice run time exceptions, `ice_print` also shows the file name and line number at which the exception was thrown.

C++ Default Constructors for Exceptions

Exceptions have a default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

To ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value.

Exceptions also have a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement (instead of first having to construct the instance and then assign to its members). For derived exceptions, this constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

C++ Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice::LocalException` (which, in turn, derives from `Ice::Exception`). `Ice::LocalException` has the usual member functions: `ice_name`, `ice_clone`, `ice_throw`, and (inherited from `Ice::Exception`), `ice_print`, `ice_file`, and `ice_line`.

Recall the inheritance diagram for user and run-time exceptions. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice::Exception`
  This is the root of the complete inheritance tree. Catching `Ice::Exception` catches both user and run-time exceptions.

- `Ice::UserException`
  This is the root exception for all user exceptions. Catching `Ice::UserException` catches all user exceptions (but not run-time exceptions).

- `Ice::LocalException`
  This is the root exception for all run-time exceptions. Catching `Ice::LocalException` catches all run-time exceptions (but not user exceptions).

- `Ice::TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.

- `Ice::ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- C++ Mapping for Identifiers
- C++ Mapping for Modules
- C++ Mapping for Built-In Types
- C++ Mapping for Enumerations
- C++ Mapping for Structures
- C++ Mapping for Sequences
- C++ Mapping for Dictionaries
- C++ Mapping for Constants
- Facets and Versioning
- Object Life Cycle

