OMG RFP Submission

IDL C++ Language Mapping Specification

Hewlett-Packard Company
IONA Technologies Ltd.
SunSoft, Inc.

OMG TC Document 93.4.4

April 23, 1993

Table of Contents

1 Introduction

This introduction:

- Lists the companies who are jointly submitting this proposal
- Explains the background to the proposal
- Provides an overview of the submission
- Summarizes the proposed IDL C++ language mapping
- Names the contacts for this OMG RFP submission

1.1 Cosubmitting Companies

The following companies are pleased to be able to jointly submit this specification in response to OMG's IDL C++ Language Mapping RFP:

- Hewlett-Packard Company
- IONA Technologies Ltd.
- SunSoft, Inc.

1.2 Proposal Background

This proposal is the merging of two separate and independent specifications, one developed by SunSoft/HP and one developed by IONA. Through technical discussions and the exchange of working papers, it was discovered that both Sunsoft/HP and IONA were proposing very similar IDL to C++ language mappings. The companies therefore judged it sensible to combine the separate specifications into a single cohesive proposal. The submitting companies have agreed to this merged specification and will be providing products that meet this specification.

In general the separate mappings were already 90% identical. This merged specification reflects the commonality of the two approaches, and includes additional worthwhile elements contained in one mapping but not in the other. The areas of commonality and additions are summarized as follows:

- The mapping of modules, interfaces and operations is common to IONA and SunSoft/HP.
- The mapping for basic data types, arrays, structs, unions and sequences is basically common to both IONA and SunSoft/HP.
- The mapping of sequences to templates originated from SunSoft/HP.
- The use of function name overloading for attribute access originated from IONA.
- The provision of destructors, copy constructors and assignment operators for structs, unions and sequences originated from IONA.
- The stress on the symmetry between client and server mapping originated from IONA.
- The support for context objects originated from IONA.

1.3 Submission Overview

The submission covers these subjects:

- Design rationale
- Mapping of IDL
- Mapping of CORBA, including the BOA

1.4 Technical Summary

The C++ language mapping follows the C language mapping in large part:

- Modules are mapped to C++ classes, used as a scoping mechanism.
- Datatypes are mapped the same as C.

- Object references behave as pointers to instances of C++ classes whose methods correspond to the IDL operations. The C++ classes have the same inheritance hierarchy as that expressed in IDL.
- Memory management rules for parameters are the same as for C.
- Exceptions are handled the same as in C for versions of C++ which do not support C++ exceptions. C++ exceptions are used when supported.
- CORBA objects (ORB, Object, etc.) are supported by mapping the IDL in the CORBA to C++ using the normal mapping rules. Some minor changes are made as required.
- An object implementation is an instance of a class derived from a class generated from the IDL.

1.5 Submission Contacts

Questions about this submission should be directed to:

- Mike Mathews
  Hewlett-Packard Company
  19447 Pruneridge Avenue
  Building 47, MS 47LP
  Cupertino, CA 95014
  U.S.A.
  Email: mjm@cup.hp.com
  Tel: 408 447-1959
  Fax: 408 447 4729

- Chris Horn
  IONA Technologies Ltd.
  O'Reilly Institute
  Westland Row
  Dublin 2, Ireland
  Email: horn@iona.ie
  Tel: +353 1 679-0677
  Fax: +353 1 679-8039

- Geoff Lewis
  SunSoft, Inc.
  2550 Garcia Avenue, MS MTV21-121
  Mountain View, CA 94043-1100
  U.S.A.
  Email: geoffrey.lewis@eng.sun.com
  Tel: 415 336-2839
  Fax: 415 336-6776

2 Design Rationale

2.1 Key Design Decisions

2.1.1 Interoperability with C

The mapping proposed here follows the approach of C++ with respect to C compatibility. The C datatypes of C++ (e.g. struct, etc.) are compatible and interoperable with C. C functions can be called from C++ and C++ functions can be declared in such a way so that they can be called from C. We believe that it is extremely important that the C and C++ mappings interoperate. C++ built its success on this natural upgrade path from C, and we should strive to maintain this feature in order to gain wide acceptance for the OMG specifications.

Interoperability between code written in C and C++ requires the ability to pass the mapped IDL data structures between the two languages. We cannot change the representations (for instance of strings or unions), without losing C interoperability.

While retaining interoperability with C, this mapping can nevertheless make good use of the features of C++. In particular, it improves on the C mapping by introducing constructors, destructors, assignment operators as well as utility functions[1] on most mapped types.

2.1.2 Mapping of CORBA Operations

The CORBA defines operations on pseudo objects such as the ORB. The language mapping must define how these operations are performed in the specified language. It is the belief of the submitters that a C++ programmer looking at the CORBA operations specified in IDL should immediately know how those operations would be specified in C++ since IDL was modeled on C++. We accomplish this by mapping the CORBA operations from IDL into C++ using the general mapping rules specified here.

2.1.3 C++ Language Specification

The version of C++ for this mapping is that defined in *The Annotated C++ Reference Manual* by Ellis and Stroustrup.

Because not all C++ compilers currently support exceptions, this mapping describes how IDL exceptions are handled without using C++ exceptions.

2.1.4 Future C++ Features

A feature called *Namespaces* has been proposed for C++. This construct would be much better than embedded classes for the mapping of IDL modules. This C++ mapping uses classes for modules in a way that will be totally upwards compatible with the eventual use of namespaces.

2.2 RFP-mandated Requirements

2.2.1 Maps Entire IDL Language

This proposal maps the entire IDL language as defined in CORBA 1.1.

2.2.2 Consistent Style

The style of this mapping is consistent with the C mapping in the CORBA 1.1 as well as consistent with common usage of C++ including such things as default constructors, copy constructors, etc.

---

1. We have equipped our mapped datatypes with a minimum set of common methods; more can be added later in a totally upwardly compatible way when experience is gained with the mapping.

2.2.3 Justification of IDL, CORBA, Object Model Extensions

No extensions to IDL, CORBA, or the object model are proposed here.

2.2.4 No Implementation Descriptions

Care has been taken to avoid details in the mapping that would constrain implementations while still allowing clients to be fully source compatible with any compliant implementation.

2.3 Evaluation Criteria

2.3.1 Reliability

The proposed mapping is natural for C++ and has no element that would impact the reliability either of CORBA implementations or of clients and servers built on the ORB.

2.3.2 Performance

There are many concerns about the performance of C++. For example, some C++ compilers produce very large object instances when multiple inheritance is used. Virtual method invocation can be considered expensive compared to normal function calls.

Care has been taken to retain the flexibility of various implementation strategies. When such flexibility had to be limited, justification is provided.

2.3.3 Portability

This mapping uses standard C++ constructs. The only relatively new constructs used are templates and (optionally) exceptions. This mapping (except for C++ exceptions) has been tested on cfront version 3.0.

3 The Mapping of IDL to C++

This chapter describes the mapping of OMG IDL constructs to C++ constructs. In general, the mapping is straightforward, due to the similarity of the IDL and C++ languages. This chapter follows the order and style of the CORBA chapter *C Language Stub Mapping* (chapter 5).

Chapter 4 describes the server side mapping and the mapping of CORBA objects.

3.1 Scoped Names

Scoped names in IDL are specified by C++ scopes:

- IDL modules are mapped to C++ nested classes.
- IDL interfaces are mapped to C++ classes (as described in “Mapping for Interfaces” on page 15).
- All IDL constructs scoped to an interface are accessed via C++ scoped names. For example, if a type *mode* is defined in interface *printer* then the type would be referred to as *printer::mode*.

These mappings allow the corresponding mechanisms in IDL and C++ to be used to build scoped names. For instance:

```
// IDL
module M
{
    interface A
    {
        struct E {
            long L;
        };
    };
};
```

is mapped into:

```
// C++
class M {
 public:
    class A : ... {
     public:
        struct E {
            long L;
        };
    };
};
```

and `E` can be referred outside of `M` as `M::A::E`.

NOTE *The ellipses in this and other examples in this chapter stand for declarations that are not germane to the current subject.*

To avoid C++ compilation problems, every use in IDL of a C++ keyword is mapped into the same name preceded by an underscore.

3.2 The CORBA Module

The objects and datatypes defined in the CORBA are realized in C++ with the same names as in the CORBA. Since many of these names (such as **Object**) are very generic, there is a risk that these names are already in use and collisions are inevitable.

To minimize the chances of such a collision, the proposed C++ mapping scopes all of the CORBA symbols in a module called CORBA. That means that the CORBA **Object** is referred to in C++ as **`CORBA::Object`**.

3.3 Mapping for Interfaces

Each OMG IDL interface is mapped to a corresponding C++ class. This C++ class is used to define a C++ type (called the pointer type) which is the object reference for objects that satisfy the interface. Note that the pointer type will likely not be a simple pointer to a class object. The pointer type bears the name of the IDL interface plus the suffix **Ref**.

The C++ class can be specified in a variety of ways depending on implementation decisions. The C++ class can be pure virtual which provides the maximum flexibility, or it can define non virtual operations. The submitters believe that a pure virtual class provides the best implementation. The examples given here are not pure virtual and so are not exactly what our implementation would produce.

Note that such differences are completely invisible to clients as long as the mapping is used legally. For example, it is not allowed to declare an instance of the mapped class even though it is possible when non-virtual classes are used.

The following example:

```
// IDL
interface Example1
{
    long op1(in long arg1);
};
```

generates the following C++ declarations:

```
// C++
class Example1 : public virtual CORBA::Object
{
   public:
   ...
   long op1(long arg1,
            CORBA::Environment &env=CORBA::default_environment);
   Example1Ref _duplicate(CORBA::Environment &env=
                              CORBA::default_environment);
   static Example1Ref _narrow(CORBA::Object obj,
                              CORBA::Environment &env=
                                   CORBA::default_environment);
   ...
};
```

In the submitters' implementation, the C++ class **example1** is an abstract class; it does not provide implementations for its member functions which are all pure virtual. A client cannot create instances of **Example1** because it is an abstract class. A client might obtain an **Example1** object from an operation on another object, for example a name service. (**example1**'s inheritance of **CORBA::Object** is described in §4.3 on page 40.).

The example methods shown above have optional environment parameters which are explained in §3.14 on page 31. The environment parameter permits the return of exception information in the absence of C++ exceptions. §3.12 on page 27 describes the exception and environment structures and provides an example of handling exceptions in client code. Subsequent examples will omit these optional parameters for brevity. The Environment parameter is a reference parameter so that the client need not pass in a pointer.

The parameter is given a default value **CORBA::default_environment** in order to satisfy the requirement in C++ that an optional parameter be given a default value. The default value cannot be NULL because it is a reference parameter. The **CORBA::default_environment** is a special Environment parameter which cannot ever explicitly be used by the client as an Environment parameter. It always indicates no exception and is not used to pass back exception information when no environment parameter is provided by the client. Any use by the client of this default environment variable is undefined.

A C++ fragment for invoking an operation on an **Example1** object is:

```
// C++
#include "example1.hh"

Example1Ref ex1;
long y;

// Initialize ex1 ...

y = ex1->op1(3);
```

The object reference used by a client is named by the interface name and the suffix **Ref**, for example the **Example1Ref** in the above example. This allows some flexibility in the implementation. The reference is required to behave like a pointer to a C++ class in that method invocation uses the arrow notation and the reference may be implicitly widened. All of the methods of the class and its inherited parents must be directly accessible.

3.3.1 Inheritance and Operation Names

IDL permits the specification of interfaces that inherit the operations of other interfaces. Consider the following in which **Example3** inherits from **Example1**:

```
// IDL
interface Example3 : Example1
{
    void op3(in long arg3, out long arg4);
};
```

This is mapped to the following C++ declarations:

```
// C++
class Example3: public virtual Example1
{
   long op3(long arg3, long &arg4);
};
```

A client written in C++ can request **op1** (or an operation defined in **CORBA::Object**) as if it were directly declared in **Example3**. Public virtual inheritance ensures that base interfaces are only inherited once in the case of multiple inheritance.

3.3.2 Widening Object References

Widening is supported by C++ mechanisms (e.g. widening through C++ inheritance or by using cast operators):

- implicit widening through parameter passing and assignment
- explicit widening by using the `<type name>` operator in functional or cast syntax.

For instance, an **Example3** object could be widened to **Example1** by:

```
// C++
Example3Ref z;
Example1Ref x = Example1Ref(z);
```

3.3.3 Narrowing Object References

Often object references are typed at a more general level than desired. For example, a name service might return object references of type **Object** but a more specific type is needed in order to perform operations. The act of converting an object reference from a more general type to a more specific type is called narrowing.

The current version of C++ does not support narrowing in the presence of public virtual inheritance. Also, C++ can only narrow down to the most derived type of the C++ object being narrowed. It would be too much of an implementation restriction to require the C++ object representing the real (possibly remote) object to be the same type as the remote object. The narrow operation allows the ORB implementation to create a new C++ object if needed. Therefore, the mapping provides special narrow operations.

Object references can be narrowed by means of a static member function generated by the IDL compiler for each class. In the case of **Example3** this would be:

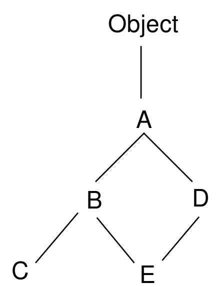
```
// C++
static Example3Ref _narrow(CORBA::ObjectRef x);
```

(The **`CORBA::ObjectRef`** parameter is described in §4.3 on page 40.)

Narrowing an object reference performs an implicit **CORBA::Object::_duplicate** operation; the client is responsible for releasing the narrowed (new) object reference.

If the narrow operation attempts to cast the object reference to a type that is not legal for that reference, the implementation may return a **BAD_PARAM** standard exception at the time of the narrow. However, to not overly constrain an implementation of the ORB, detection of improper narrows is not required. Checking the legality of a narrow may, in the worst case, cost as much as an invocation on the object.

Suppose **`obj`** really is a **`C`** (as defined in the diagram below) that has been widened to an **`A`**. Passing **`obj`** to **`B::_narrow`** or to **`C::_narrow`** does not raise **BAD_PARAM**; passing it to **`E::_narrow`** or to **`D::_narrow`** may raise **BAD_PARAM**. Although a **`C`**-widened-to-an-**`A`** may, in some implementations, be improperly narrowed to a **`D`** without raising a **BAD_PARAM** exception, requesting a **`D`** operation on the object nevertheless raises the **BAD_OPERATION** exception.



3.4 Mapping for Attributes

The mapping for each read-write attribute produces two member functions, both with the same name as the attribute: one to get the attribute's value and the other to set it. Consider the following example:

```
// IDL
interface DepositAccount : Account {
      attribute float balance;
      readonly attribute float over_draft_limit;
};
```

The equivalent C++ is:

```
// C++
class DepositAccount: public virtual Account {
   float balance();
   void balance(float x);
   float over_draft_limit();
};
```

Thus, attributes are mapped to two member functions (to get and set the value); **readonly** attributes are mapped to one member function (to get the value).

3.5 Mapping for Constants

IDL constants are mapped differently depending on whether they are global or occur in an IDL scope. A constant defined outside any scope:

```
// IDL
const long L = 4;
```

is mapped to:

```
// C++
const long L = 4;
```

The same constant defined in IDL interface S is mapped to:

```
// C++
class S ...
{
   ...
   static const long L;
   ...
}
```

in the header (.hh) file, and:

```
// C++
const long S::L = 4;
```

in the definition (.cc) file.

3.6 Mapping for Basic Data Types

The basic data types have the mappings shown in TBL. 1 on page 20. Note that the **`unsigned char`** mapping of the OMG IDL **boolean** type defines only the values 1 (TRUE) and 0 (FALSE); other values produce undefined behavior.

TBL. 1 Basic Data Type Mappings

IDL	C++
short	short
long	long
unsigned short	unsigned short
unsigned long	unsigned long
float	float
double	double
char	char
boolean	unsigned char
octet	unsigned char
enum	enum[1]

[1]Note that whereas IDL enums may have up to 2^32 values, C++ enums are allowed to be implemented with a limit of 127 values.

3.7 Mapping for Structure Types

OMG IDL structures map directly onto C++ structs. Note that all OMG IDL types that map to C++ structs may potentially include padding.

The IDL struct:

```
// IDL
struct str {
    long i;
    string j;
};
```

maps to:

```
// C++
struct str {
   long i;
   char *j;

   str();
   ~str();
   str(const str &);
   str &operator= (const str &);
};
```

Each struct has a default constructor, a copy constructor, a destructor, and an assignment operator. The default constructor creates a struct with default initial values. The destructor deletes any embedded storage (such as the string in the above example). The copy constructor does a deep copy of the struct by copying the struct then copying everything the struct points to. The assignment operator overwrites (and frees) the existing contents of the struct then deep copies the source.

Note the impact of this behavior of the destructor. The destructor will attempt to free any non-null storage contained within the struct. If the struct was provided with non-dynamic storage (e.g., static or stack storage) then the attempted destruction of that storage may cause problems. This is also true of the destructors for unions and sequences as described below.

It is an unspecified implementation detail how these operators are implemented, with one restriction. They may be implemented as non-virtual methods or as inlines, but they cannot be implemented as virtual methods. This is because of the desire to have compatibility between the C and C++ structures and the representation of a struct is changed if virtual methods are used.

3.8 Mapping for Union Types

OMG IDL discriminated unions are mapped onto C++ structs. Consider the following OMG IDL declaration:

```
// IDL
union Foo
switch (long) {
   case 1:      long x;
   case 2:      float y;
   default:     string z;
};
```

This is equivalent to the following struct in C++:

```
// C++
struct Foo {
   long _d;
   union {
      long x;
      float y;
      char *z;
   };

   Foo();
   ~Foo();
   Foo(const Foo &);
   Foo &operator= (const Foo &);
};
```

The discriminator in the struct is always referred to as **`_d`**; the union in the struct is always an anonymous union.

The constructor, destructor, copy constructor, and assignment operator serve the same purpose as in the struct mapping. The destructor checks the value of the **`_d`** field to decide what needs to be deleted. For example, if the discriminator indicates that the union contains a string, then the destructor is called on that string.

The C++ language definition states that an object of a class with a constructor or a destructor or a user-defined assignment operator cannot be a member of a union. The proposed definitions of struct, sequence, union, and object reference have such constructs and hence are not allowed in a union. To retain compatibility with C unions, it is necessary to define special versions of datatypes included in unions.

For example, if a union were defined to contain the struct defined in the previous section, a special version of that struct would have to be defined which did not contain any of the prohibited operations.

```
// IDL
union Foo
switch (long) {
   case 1:     long x;
   case 2:     str s;
};
```

The C++ union would have to refer to a struct without the methods:

```
// C++
struct _str {
   long i;
   char *j;
};
struct Foo {
   long _d;
   union {
      long l;
      _str s;
   };

   Foo();
   ~Foo();
   Foo(const Foo &);
   Foo &operator= (const Foo &);
};
```

Two additional operations would have to be defined on the original struct to allow implicit conversions:

```
// C++
struct str {
   // stuff as defined above
   str(const _str &);
   operator _str();
};
```

The additional operators allows an **`str`** to be put into a union and extracted from a union.

```
// C++
str s;
s.i = 1;
s.j = new char[20];
Foo f;
// set the value of the union
f._d = 1;
f.s = s;
// extract the struct in the union
if (f._d == 1) {
   s = f.s;
}
```

Note that a client is not required to refer to the special struct defined to accommodate the union.

Because the embedded struct does not have its own destructor, the destructor of the union must destruct the embedded struct.

Reference to union elements is as in normal C++:

```
// C++
Foo *v;

// make a call that returns a pointer to a Foo in v

switch(v->_d) {
   case 1:  cout << v->x; break;
   case 2:  cout << v->y; break;
   default: cout << v->z; break;
}
```

3.9 Mapping for Sequence Types

OMG IDL sequences are usually defined as typedefs, for example:

```
// IDL
typedef sequence<long,10> vec10;
```

The IDL compiler maps such a sequence to a struct containing three members:

- **`_maximum`** which describes the sequence's maximum number of elements
- **`_length`** which describes the number of elements actually in the sequence
- **`_buffer`** which points to an array containing the sequence data (i.e., an array of longs in the vec10 example above)

A C++ programmer can declare an instance of a **`vec10`** sequence in a variety of ways:

```
// C++
vec10 x;             // empty seq. with no allocated storage
vec10 x(20);         // seq. of maximum 20, length 20, and buffer of 20
vec10 x(10, 5, buf); // seq. of max 10, length 5, and buffer
```

Prior to passing **`x`** as an in parameter, the programmer must set the **`_buffer`** member to point to a **`long`** array of 10 elements, and must set the **`_length`** member to the actual number of elements to transmit. (**`_length`** denotes the number of elements in the sequence, not the size of the sequence.)

Prior to passing **`x`** as an out parameter, or receiving a **`vec10`** as a return value, the programmer does nothing. The client stub will allocate storage for the returned buffer; for bounded sequences it allocates a buffer of the specified size, while for unbounded sequences it allocates a buffer large enough to hold what the object returns. Upon successful return from an invocation, the **_maximum** member will contain the size of the allocated array, the **_buffer** member will point at the allocated storage, and the **_length** member will contain the number of values that were returned in the **_buffer** member.

Prior to passing **x** as an inout parameter, the programmer must set the **_buffer** member to point to a **long** array of 10 elements. For an unbounded sequence, the programmer must set the **_maximum** member to the actual size of the array. The **_length** member must be set to the actual number of elements to transmit. Upon successful return from the invocation, the **_length** member will contain the number of values that were copied into the buffer pointed to by the **_buffer** member. The number of values returned is constrained by the value of the **_maximum** member.

For bounded sequences, it is an error to set the **_length** or **_length** member to a value larger than the specified bound.

OMG IDL sequences are mapped with the C++ template facility. The IDL compiler generates the following template definition:

```
// C++
template <class T>
struct CORBA_sequence {
   unsigned long  _maximum;
   unsigned long  _length;
   T              *_buffer;

   CORBA_sequence();  // default constructor, no space allocated
   CORBA_sequence(unsigned long); // allocates storage
   CORBA_sequence(unsigned long, unsigned long, T *);
   CORBA_sequence(const CORBA_sequence &); // deep copy constructor
   ~CORBA_sequence(); // destructor, deallocates buffer
   CORBA_sequence operator=(const CORBA_sequence &); // assignment op
};
```

These methods may be implemented as non-virtual methods or as inlines. Virtual methods cannot be used because they would change the memory layout of the sequence so that it no longer matched the C sequence.

Suppose an interface defines a sequence of **S** as follows:

```
// IDL
interface I
{
   typedef string S;
   void op (in sequence <S> x);
};
```

A C++ programmer declares **x** as follows:

```
// C++
CORBA_sequence<I::S> x;
```

To declare a sequence of this sequence, the programmer writes:

```
// C++
CORBA_sequence< CORBA_sequence<I::S> > x;
```

3.10 Mapping for Strings

OMG IDL strings are mapped to 0-byte terminated character arrays; i.e. the length of the string is encoded in the character array itself by the placement of the 0-byte. Note that the storage for C++ strings is one byte longer than the stated IDL bound. Consider the following IDL declarations:

```
// IDL
typedef string<10> sten;
typedef string sinf;
```

In C++, this is converted to:

```
// C++
typedef char *sten;
typedef char *sinf;
```

Instances of these types are declared as follows:

```
// C++
sten s1 = 0;
sinf s2 = 0;
```

Prior to passing **s1** or **s2** as an in parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated string to the variable.

Prior to passing **s1** or **s2** as an out parameter (or receiving an **sten** or **sinf** as the return result), the programmer does nothing. The client stub will allocate storage for the returned buffer; for bounded strings, it allocates a buffer of the specified size; for unbounded strings, it allocates a buffer big enough to hold the returned string. Upon successful return from the invocation, the character pointer will contain the address of the allocated buffer. The client is responsible for freeing the allocated storage. If the pointer is non-NULL when a call is made, it is overwritten with a pointer to the storage allocated by the stub.

Prior to passing **s1** or **s2** as an inout parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated array to the variable. Upon successful return from the invocation, the returned 0-byte terminated array is copied into the same buffer. If it was a bounded string, then the size of the returned string is limited by the declared size of the string type; if it was an unbounded string, then the size of the returned string is limited by the size of the string passed as input. Due to this restriction, use of inout string parameters is deprecated.

3.11 Arrays

IDL arrays map directly to C++ arrays. Array indices run from 0 to <**size** - 1>.

If the return result to an operation is an array, the array storage is dynamically allocated by the stub; a pointer to the array is returned as the value of the client stub function. When the data is no longer needed, it is the programmer's responsibility to delete the dynamically allocated storage.

3.12 Mapping for Exception Types

Each defined OMG IDL exception type is mapped to a scoped struct containing the data and a string identifying the exception. For example:

```
// IDL
exception Foo {
    long          dummy;
};
```

yields the following C++ declarations:

```
// C ++
struct Foo : public CORBA::Exception {
   long      dummy;
};

static const char *ex_foo;
```

CORBA::Exception is the parent of all exceptions and is a struct defined as shown in the CORBA module:

```
// PIDL
struct Exception {
    char *_id;
};
```

This exception structure allows all CORBA exceptions, both system and user, to be caught with a single catch. It differs from the IDL module StExcep in that it must remain a type and the StExcep module will map to the C++ Namespace when compilers support it.

The value of the **_id** field is the exception identifier as shown above. This identifier allows the type of the exception to be inspected at run time. This identifier references static storage and is not freed.

The system exceptions are all derived from the special class **`CORBA::StExcep::SystemException`**. This allows all standard exceptions to be caught with a single C++ catch. See the example in the next section.

3.13 Handling Exceptions

Exceptions can be handled in one of two ways. If C++ exceptions are supported, then IDL exceptions are thrown as C++ exceptions. If C++ exceptions are not supported, the environment parameter is used to detect the exception.

3.13.1 C++ Exceptions

Consider the following interface:

```
// IDL
interface Bank {
      exception Reject {
         string reason;
      };
      Account new_account(in string name) raises (Reject);
};
```

When C++ exceptions are supported, exceptions raised by an invocation of an operation are handled as follows:

```
// C++
BankRef obj;
AccountRef a;
char *new_name;
...
try {
   a = obj->new_account(new_name);
}
catch (Bank::Reject exc) {
   report_bad_account(new_name, exc.reason);
}
catch (CORBA::StExcep::SystemException) {
   // system exception
}
```

3.13.2 Exceptions Passed through the Environment

The **CORBA::Environment** type passed as an implicit parameter in each request is partially opaque; the C++ declaration contains at least the following:

```
// C ++
struct Environment {
   StExcep::exception_type _major;
   Environment();
   Environment(Environment &);
   ~Environment();
   char *exception_id();
   void *exception_value();
   void exception_free();
   ...
};
```

An environment's value is undefined until it has been specified in a request; calling any of the methods on an undefined environment produces an undefined result.

Upon return from an invocation, the **_major** field indicates whether the invocation terminated successfully; **_major** can have one of these values:

- **CORBA::StExcep::NO_EXCEPTION**
- **CORBA::StExcep::USER_EXCEPTION**
- **CORBA::StExcep::SYSTEM_EXCEPTION**

If the value is either of the latter two, then any exception parameters signalled by the object can be accessed.

Three methods are defined on a **CORBA::Environment** structure for accessing and freeing exception information.

CORBA::Environment::exception_id() returns a pointer to the character string identifying the exception. If invoked on a **CORBA::Environment** that identifies a non-exception, a NULL is returned.

CORBA::Environment::exception_value() returns a pointer to the structure corresponding to this exception. If invoked on a **CORBA::Environment** that identifies a non-exception or an exception for which there is no associated information, a NULL is returned.

exception_free() returns any storage pertaining to the exception. It is permissible to call **exception_free()** regardless of the value of the **_major** field. However, if a request's outcome is an exception, **exception_free()** must be called before specifying the same environment in another request; failure to do so may cause a memory leak.

Consider the example in the section above. The following user code shows how to invoke the operation and recover from an exception:

```
// C++
CORBA::Environment ev;
BankRef obj;
Bank::reject *exc;
char *new_name;

...

obj->new_account(new_name, ev);
switch(ev._major) {
  case CORBA::StExcep::NO_EXCEPTION:
   break;
  case CORBA::StExcep::USER_EXCEPTION:
   if (strcmp(Bank::ex_reject, ev.exception_id())
          == 0) {
         exc = (Bank::reject *)ev.exception_value();
         report_bad_account(new_name, exc->reason);
   }
   break;
  default:  // standard (system) exception
    // system exception
   break;
}
ev.exception_free();       // free exception storage
```

3.14 Implicit Arguments to Operations

Every mapped operation includes an optional environment parameter which is the last parameter in the signature of the operation. If a **context** clause is present in an operation declaration, a context parameter is included after the operation parameters but before the environment parameter.

3.15 Argument Passing Considerations

For all OMG IDL types except arrays and strings, if the IDL signature specifies that an argument is an out or inout parameter, then the caller passes a variable of that type; for arrays and strings, the caller passes a pointer to the first element of the array or string. If the IDL argument is an in, the client passes a value of the correct type, except for arrays and strings, for which the client passes a pointer to the first element of the array or string.

Consider the following IDL specification:

```
// IDL
interface Foo
{
    typedef long Vector[25];
    struct Complex {
        long real;
        long imaginary;
    };

    void bar
    (
        in long x,
        out Vector y,
        inout complex z
    );
};
```

Client code for invoking the bar operation could look like:

```
// C++
Foo::Vector y;
Foo::Complex z;
CORBA::Environment ev;

// code to bind object and initialize z ...

object->bar(4, y, z, ev);
```

Clients are responsible for deleting ORB-allocated storage for out parameters and return results as described later in this section.

TBL. 2 on page 32 summarizes parameter and result passing.

TBL. 2 Argument and Result Passing

Data Type	Pass In	Pass Out/Inout	Return Result
short	value	reference	receive value
long	value	reference	receive value
unsigned short	value	reference	receive value
unsigned long	value	reference	receive value
float	value	reference	receive value
double	value	reference	receive value
boolean	value	reference	receive value
char	value	reference	receive value
octet	value	reference	receive value
enumeration	value	reference	receive value
object reference[1]	object reference	reference	receive value of object reference
struct	reference[2]	reference	receive value of struct
union	reference[2]	reference	receive value of struct
string	pointer to 1st char[2]	reference to (char *) variable	receive char *
sequence	reference[2]	reference	receive value of sequence struct
array	pointer to 1st element[2]	address of 1st element	pointer to array

1. Including pseudo-object references.
2. Passed as const to provide compiler check for inadvertent modification by implementation.

A client is responsible for providing storage for all arguments passed as in arguments. All simple data types (shown as short through enum in TBL. 2 on page 32) are always passed in by value and are never allocated or freed. TBL. 3 on page 33 and TBL. 4 on page 34 describe the client’s responsibility for storage associated with inout and out parameters and for return results. A client is responsible for providing storage for all out arguments and return results except in the cases noted in TBL. 3 on page 33 and TBL. 4 on page 34.

TBL. 3 Client Argument Storage Responsibilities

Type	Inout Param	Out Param	Return Result
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enumeration	1	1	1
object reference	2	2	2
struct	1	1	1
union	1	1	1
string	1	3	3
sequence	1	4	4
array	1	1	3
any	5	5	5
TypeCode	5	5	5

TBL. 4 Argument Passing Cases

Case[1]	
1	The client is responsible for providing storage and managing release of the storage. That is, the system allocates storage which must be freed for the following types and directions: string/out, string/return, sequence/out, sequence/return, array/return. For inout strings and sequences, the out result is constrained by the size of the type on input.
2	The client is responsible for releasing the returned object reference, using CORBA::Object::release, it should not be deleted. To continue to use an object reference passed in as an inout, a client must first duplicate the reference. To release storage occupied by an object reference passed as an inout, the client must maintain a reference to the "in" reference, and use this pointer in the CORBA::Object::release operation following completion of the request. Storage freed using the C++ delete operation automatically performs a CORBA::Object::release for all object references embedded in ORB-allocated storage; clients should not release these object references.
3	The ORB provides the storage for these returned parameters and results. The client is responsible for releasing the storage.
4	The client provides the storage for the structure which contains the description of the sequence and the client manages release of the storage and the descriptor. The ORB provides storage for the values returned and puts the pointers to this storage in the descriptor structures. The client is responsible for releasing the ORB-allocated storage.
5	The client is responsible for releasing the returned pseudo-object reference using the relevant operation defined in the pseudo-object interface. An inout any is constrained to be the same type and size on output as input. This means that the typecode field cannot be modified and the value field must be exactly the same size (the existing value field can just be overwritten). To continue to use an any or typecode passed as an inout, the client must copy the parameter by creating a new instance and copying the data making up the pseudo-object, e.g., by copying the typecode and value of an any.

1. As listed in TBL. 3 on page 33

Note that the size of an unbounded string or sequence whose value is set by an operation is undefined until the operation completes.

3.16 Pseudo-objects

Pseudo-object interfaces are mapped like those of real objects, with some minor differences described in the next chapter.

3.17 Include Files

By convention, each OMG IDL interface or module is stored in a separate source file. Compiling an IDL file named Foo.idl yields a header file named Foo.hh. This file must be

#included by clients and implementations of the interfaces defined in Foo.idl. Inclusion of Foo.hh is sufficient to define all symbols associated with the interfaces in Foo.idl and any interfaces from which they are derived.

4 The Mapping of CORBA Operations

This chapter describes the mapping of OMG CORBA operations to C++. In general, the mapping is straightforward, using the language mapping of IDL to C++. Each section of this chapter describes the mapping of a section of the CORBA.

4.1 Dynamic Invocation Interface

The Dynamic Invocation Interface as defined in chapter 6 of the CORBA is primarily mapped to C++ using the mapping rules described in the previous chapter. Since these are CORBA defined objects and types, their names are scoped within the CORBA module.

4.1.1 Request, Send, and List Routines

The IDL defining these interfaces described in sections 6.2 through 6.4 of the CORBA are mapped according to the normal C++ mapping rules.

4.1.2 Context Object Routines

A client can maintain one or more Context objects, which provide a mapping from Identifiers (in effect, strings) to string values. An IDL operation can specify that it is to be provided with the client's mapping for particular Identifiers—it does this by listing these Identifiers following the operation declaration. For example, in the following interface definition, the operation **op1** specifies that it is to receive the mapping for Identifiers **accuracy** and **base**.

```
// IDL
interface f {
  operation void op1 (int s) raises (exc1)
    context (accuracy, base);
};
```

The set of Identifiers is not defined by the infrastructure, but each Identifier name must begin with an alphanumeric character and can only contain alphanumerics, digits, _ and .. An identifier specified in a context clause can also contain the character *, but this character must appear at the end—it indicates that the operation is to receive the mapping for all identifiers in the client context with matching leading names. For example, an identifier **sys_*** in a context clause would match entries such as **sys_printer** and **sys_quality** in the client's context.

An operation that specifies a context clause is mapped to a C++ member function that takes an extra parameter, just before the environment parameter:

```
// C++
class f {
    virtual void op1 (int s, Context &c,
                CORBA::Environment &ev=CORBA::default_environment);
};
```

This order allows the environment parameter to have a default value.

A client can create a Context by defining a variable of type **`Context`**:

```
// C++
Context c;
```

This creates an initially empty Context object, to which identifier:value mappings can be added, and which can be passed to a function that takes a Context parameter.

Because the infrastructure on the client side does not delete the context passed to it, it is also the responsibility of the client programmer to eventually delete the context if it was dynamically allocated.

The infrastructure on the server side constructs a new Context from the value received in the incoming operation request, and calls the target object's operation. The infrastructure deletes the context when the call returns. Should the server programmer require that the context be retained after the call, he should copy the context argument passed in the call.

Class **`Context`** is defined by the infrastructure as follows:

```
// C++
class Context {
 public:
 // constructors
 Context(Context *parent=NULL);
 Context(const char *name, Context *parent=NULL);

 //destructor (equivalent to calling the Delete fn
 // below - with zero flag).
 ~Context();

 // add property name and value to Context:
 ORBStatus set_one_value (
    Identifier prop_name,//add property name
    const char *value );//value

 // set a number of property values. Any prior
 // property values are forgotten:
 ORBStatus set_values (
   NVList &values ); // property_name:values to change:

 // retrieve matching values
 ORBStatus get_values (
   const Identifier start_scope,//search scope.
   Flags op_flags,//flags.
   const Identifier prop_name,//property name
   //output property_name:value list:
   NVList *&values );

 // delete specified property from context
 ORBStatus delete_values (
   const Identifier prop_name );//property name

 // create child context with specified name
 ORBStatus create_child (
   const Identifier ctx_name,//Context name
   Context *&child_ctx );//newly created

 // delete the context
 ORBStatus _delete (
   Flags &del_flags );//how to act
};
```

4.2 The Interface Repository

The Interface Repository as described in chapter 7 of the CORBA is mapped according to the normal C++ mapping rules.

4.3 Object Reference Operations

Object is the base class from which all object references derive. It is defined mainly for typing purposes i.e., to provide a root for the IDL type system on which widen/narrow mechanisms can be built. It also allows generic functions which work on any type of object to be written. The interface **Object** is defined as:

```
// PIDL
interface Object
{
    ImplementationDef       get_implementation();
    InterfaceDef            get_interface();
    boolean                 is_nil();
    Object                  duplicate();
    void                    release();

    ORBStatus create_request(
        in Context          ctx,
        in Identifier       operation,
        in NVList           arg_list,
        inout NamedValue    result,
        out Request         request,
        in Flags            req_flags
    );
};
```

It is generally mapped to C++ like any other interface:

```
// C++
class Object
{
   public:
      ImplementationDefRef _get_implementation();
      InterfaceDefRef _get_interface();
      static unsigned char _is_nil(CORBA::ObjectRef obj);
      void _release();
      ObjectRef _duplicate();
      ORBStatus _create_request
      (
         CORBA::ContextRef ctx,
         CORBA::Identifier &operation,
         NVList &arg_list,
         CORBA::NamedValue &result,
         CORBA::RequestRef &request,
         CORBA::Flags req_flags
      );
};
```

Unlike the mapping of ordinary interfaces, the **Object** mapped operation names start with an underscore. Because IDL reserves names starting with an underscore, **Object** operations names cannot collide with the names of user defined operations. For C++ compilers that do not support covariant return types, **_duplicate** must be nonvirtual to allow its redefinition in classes derived from **Object**. It is required that **_duplicate** return a reference to an object of the same type as that being duplicated. The **_is_nil** operation is defined as static so as not to fail if passed an OBJECT_NIL value.

All C++ abstract classes generated from IDL interfaces inherit **CORBA::Object**. As a result, a programmer can request **Object** operations on an **example3Ref** object in either of these ways, for example:

```
// C++
Example3Ref x;
unsigned short a_boolean ;
x = CORBA::OBJECT_NIL;
a_boolean = x->_is_nil(x, ev);
a_boolean = CORBA::Object::_is_nil(x, ev);
```

4.4 Null Object References

The CORBA defines OBJECT_NIL to represent a null object reference. The C++ mapping is:

```
//C++
enum {OBJECT_NIL = 0};
```

CORBA::OBJECT_NIL is valid in all places where an Object is allowed. Narrowing an **OBJECT_NIL** yields an **OBJECT_NIL**. An object reference whose value is **OBJECT_NIL** is guaranteed to pass the **CORBA::Object::_is_nil** operation.

4.5 The Basic Object Adapter

4.5.1 C++ Language Mapping for Object Implementations

This section describes how BOA-based object implementations written in C++ interact with the BOA.

4.5.2 Method Signatures

For each interface **Intf** implemented in C++ the IDL compiler generates a class **IntfBOAImpl** which has pure virtual functions for the operations defined in the OMG IDL interface specification. An implementation of **Intf** inherits from **IntfBOAImpl** and implements the virtual functions.

It is important for implementation flexibility that the class that a BOA implementation inherits from is not the same class used by a client. The class described here must have virtual methods so that the implementation can override the methods to provide the actual implementation of the object. As described before, it is important to allow the classes used for object references to not have virtual methods.

Suppose an interface is specified as:

```
// IDL
interface ex1
{
    long op1(in long l);
};
```

The IDL compiler generates this abstract class (other definitions in the class are described shortly):

```
// C++
class ex1BOAImpl
{
  public:
   virtual long op1
   (
      long l,
      BOA::MethodEnvironmentRef me
   ) = 0;
};
```

The implementation of this class could have the following signature (the name of the class could be different, but must not be **ex1BOAImpl**):

```
// C++
class acmeEx1 : public virtual ex1BOAImpl
{
  public:
   long op1
   (
      long l,
      BOA::MethodEnvironmentRef me
   );
};
```

The **me** argument is the client's environment and is described in §4.5.4 on page 45.

If an interface is contained in an OMG IDL module hierarchy, the IDL compiler generates an analogous class hierarchy. For example, if an interface **Intf** is defined in a module **m1** which is defined in a module **m0**, then the class corresponding to the interface is **m0BOAImpl::m1BOAImpl::IntfBOAImpl**. **BOAImpl** classes inherit from each other in the same way that the client abstract classes do (see §3.3.1 on page 16).

All of the symbols defined in the IDL hierarchy are defined with the same names in this analogous class hierarchy. For example, if the interfaces defines a type **foo**, then that type can be referenced from the **BOAImpl** class directly, without using a fully scoped name. This use pertains to all generated symbols such as exception ids.

4.5.3 Raising Exceptions

4.5.3.1 C++ Exceptions

When C++ exceptions are supported, an IDL exception is raised by raising the corresponding C++ exception. Following the example in §3.13 on page 28, the method code would be:

```
// C++
if (! legal_account_name(name)) {
   reject exc;
   exc.reason = new char[bad_name_msg_len+1];
   strcpy(exc.reason, bad_name_msg);
   throw(exc);
}
```

The exception is allocated locally but all of the storage in the exception must be allocated since it will be freed after the exception is caught and handled. In the example above, the actual exception **exc** is declared local but the storage within the exception, the **reason**, is dynamically allocated. For more details, see the description of C++ exceptions in the *Annotated C++ Reference Manual*.

4.5.3.2 Set Exception

The method terminates with an error by requesting the **set_exception** operation prior to executing a return statement. The **set_exception** operation is defined on the **MethodEnvironment** and has the following C++ language definition:

```
// C++
class MethodEnvironment {
  public:
   void set_exception
   (
      CORBA::StExcep::exception_type   major,
      const char                       *user_id,
      const void                       *param
   );
};
```

Like all methods, this one has an optional trailing environment parameter used to report an exception (the method is acting as a client here). The **MethodEnvironment** object is the environment parameter passed into the method (see §4.5.4 on page 45). The caller must supply a value for the **major** parameter. The value of the **major** parameter constrains the other parameters in the call as follows:

- If the **major** parameter has the value **CORBA::StExcep::NO_EXCEPTION,** then it specifies that this is a normal outcome to the operation. In this case, both **use-**

r_id and `param` must be NULL. Note that it is *not* necessary to invoke **set_exception** to indicate a normal outcome; it is the default behavior if the method simply returns.

- If the **major** parameter has the value **CORBA::StExcep::SYSTEM_EXCEPTION**, then it specifies that the outcome is a standard exception. The **user_id** parameter is a string representing the exception type identifier. The **param** parameter must be the address of a struct containing the parameters according to the C++ language mapping, coerced to a **void ***.
- For any other value of **major** it specifies either a user-defined or standard exception. If the `major` parameter has the value **CORBA::StExcep::USER_EXCEPTION**, then it specifies that the outcome is a user-defined exception. The **user_id** parameter is a string representing the exception type identifier. If the exception is declared to take parameters, the `param` parameter must be the address of a struct containing the parameters according to the C++ language mapping, coerced to a **void ***; if the exception takes no parameters, `param` must be NULL.

When raising an exception, the method code is *not* required to assign legal values to any `out` or `inout` parameters.

4.5.4 MethodEnvironment

A **MethodEnvironment** represents the **CORBA::Environment** passed by the method's client. The method can do nothing with a **MethodEnvironment** except pass it to operations that require it as a parameter, such as **BOA::set_completion**.

The parameter through which a BOA based implementation method receives call specific information (such as the principal) and specifies call specific information (such as exceptions) is typed differently from the environment parameter used by a client. This is to avoid the confusion of the two flavors of environment used by the server. The two environments are used completely differently and so should be typed differently.

4.6 Implementation Parameter Memory Management

The ORB does not constrain an implementation's private use of dynamically allocated memory. In contrast, an implementation must coordinate with the ORB to properly reclaim parameter memory: memory whose contents are returned to a client. This section describes how parameter memory is allocated, and how it can be reclaimed by either of two techniques, one simple, the other flexible.

4.6.1 Allocation

Parameter memory must be allocated for everything returned to clients: out and inout parameters, and results. Note that the environment parameter is treated as an in. TBL. 5 on page 46 and TBL. 6 on page 46 describe who (ORB or implementation) is responsible for allocating and freeing each kind of parameter memory.

TBL. 5 Parameter Storage Responsibilities

Type	Inout Param	Out Param	Return Result
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enumeration	1	1	1
object reference	2	2	2
struct	1	1	1
union	1	1	1
string	1	3	3
sequence	1	4	4
array	1	1	3
any	1	5	5
TypeCode	1	5	5

TBL. 6 Parameter Passing Cases

Case	Description
1[1]	The ORB provides and releases the storage. For inout strings and sequences, the out result is constrained by the size of the type on input. For inout any, the typecode field must be exactly the same size (i.e., the existing storage is reused). For inout TypeCode, the size of the TypeCode must remain the same. The use of inout strings and TypeCodes is deprecated.
2	The implementation provides the object reference and is responsible for releasing the object reference with CORBA::Object::release when it is no longer needed.
3	The implementation allocates storage for the string/out, string/return, and array/return. The object implementation must free this storage when it is no longer needed.

TBL. 6 Parameter Passing Cases

Case	Description
4	The ORB provides the storage for the structure that contains the description of the sequence. The implementation provides storage for the values returned and puts the pointers to this storage in the descriptor structures. The implementation is responsible for releasing this storage when it is no longer needed.
5	For an out any or TypeCode, the ORB provides the pseudo-object initialized to be empty. The implementation must set the fields of the any or TypeCode appropriately and then release the storage when no longer needed. For a return any or TypeCode, the implementation is responsible for creating the new pseudo-object and releasing it when no longer needed.

1. As listed in TBL. 5 on page 46.

4.6.2 Implicit Deallocation

The implementation is responsible for deallocating the parameter memory it allocates (i.e., cases 2, 3, and (partially) 4 in TBL. 5 on page 46). However, it is the ORB and not the implementation that knows *when* parameter memory can be deallocated. The implementation can let the ORB implicitly deallocate parameter memory, or can direct the ORB to inform the implementation when deallocation is safe (when the parameters have been transmitted to the client). Most implementations will be able to use implicit deallocation; alternatively, implementations can deallocate memory explicitly, as described in §4.6.3 on page 47. It is also possible to do both, and to switch between the techniques dynamically.

Implicit deallocation of parameter storage is the default. After a method returns (without raising an exception) and the ORB is finished with the parameters, the ORB automatically deallocates the parameter storage. The ORB frees datatypes other than Object with the C++ **`delete`** operator (i.e., the ORB assumes that the storage was allocated using the corresponding **`new`** operator.) The ORB releases object references with **CORBA::Object::release**. The ORB frees all of the subcomponents of complex data structures, assuming that each subcomponent was individually allocated. For example, for an `out` sequence of structs with each struct containing a string and an Object, each string is freed and each Object is released, then each struct is freed, then the sequence value array is freed.

If a method using implicit deallocation wants to retain a parameter value, it must make a copy of the parameter before returning. For an Object, this copy is made using **CORBA::Object::duplicate**. The implementation is responsible for freeing the copy when it is no longer needed. An Object is freed with **CORBA::Object::release**.

4.6.3 Explicit Deallocation

A method can explicitly deallocate the parameter memory it has allocated by registering a completion function with the ORB. (Such a method might use something other than **`new`** and **delete** for memory management, but there are no requirements for choosing explicit deallocation.) The ORB calls the method's completion function when the memory can be deallocated; the completion function should deallocate the memory and return to the ORB.

Explicit deallocation is in effect when a completion function is registered. A completion function must be registered by the method for each invocation. If no completion function is registered then implicit deallocation is in effect for the method invocation. A completion function must be registered at most once per method invocation.

If the completion function registered is a NULL pointer, this means that explicit (not implicit) deallocation is in effect, but no completion function will be called. For example, if part of the state of the activated object is being returned, it need not be freed because it is retained as long as the object is activated.

Note that if a completion function is registered, it is guaranteed to be called, even if there is no parameter storage to be freed (e.g. there were no out, inout, or result parameters).

A method registers a completion function by requesting the **BOA::set_completion** operation (other object adapters may provide analogous operations):

```
// C++
typedef void (*Completion_Fn)
(
   MethodEnvironmentRef mev,
   void                 *param
);

virtual void set_completion
(
   MethodEnvironmentRef mev,
   Completion_Fn        fn,
   const void           *param
) = 0;
```

The **mev** parameter identifies the method invocation to the ORB. The **fn** parameter is a pointer to the completion function or a NULL pointer. The **param** parameter is a pointer that the ORB will pass back to the completion function. The pointer must enable the completion function to deallocate (and release, in the case of object references) all implementation-allocated parameter memory associated with the method invocation. For example, **param** might point to an array of pointers to the allocated parameters. The information associated with **param** must be sufficient to deallocate parameter memory for both normal and exceptional outcomes; the ORB calls the completion function regardless of the outcome.

4.6.4 Exception Deallocation

If a BOA-based method raises an exception by requesting **set_exception**, the method does not allocate storage for the **mev**, **major**, or **user_id** arguments. It does allocate storage for the **ev** and the **param** arguments; the latter is a pointer to a struct that contains the exception value that is copied to the client.

For deallocation purposes, the ORB treats the exception struct as if it were embedded in an ORB-allocated sequence. The struct can be deallocated implicitly or explicitly. If the method has not specified a completion function, the ORB implicitly deallocates the struct with **delete**; however, it does not deallocate the method's parameters (it assumes the method did not allocate them due to the exception). If the method specifies a completion function, that function should deallocate the struct.