C Language Mapping 19

CORBA is independent of the programming language used to construct clients and implementations. In order to use the ORB, it is necessary for programmers to know how to access ORB functionality from their programming languages. This chapter defines the mapping of OMG IDL constructs to the C programming language.

Contents

This chapter contains the following sections.

19.1 Requirements for a Language Mapping

All language mappings have approximately the same structure. They must define the means of expressing in the language:

- All OMG IDL basic data types
- All OMG IDL constructed data types
- References to constants defined in OMG IDL
- References to objects defined in OMG IDL
- Invocations of operations, including passing parameters and receiving results
- Exceptions, including what happens when an operation raises an exception and how the exception parameters are accessed
- Access to attributes
- Signatures for the operations defined by the ORB, such as the dynamic invocation interface, the object adapters, and so forth.

A complete language mapping will allow a programmer to have access to all ORB functionality in a way that is convenient for the particular programming language. To support source portability, all ORB implementations must support the same mapping for a particular language.

19.1.1 Basic Data Types

A language mapping must define the means of expressing all of the data types defined in “Basic Types” on page 3-23. The ORB defines the range of values supported, but the language mapping defines how a programmer sees those values. For example, the C mapping might define TRUE as 1 and FALSE as 0, whereas the LISP mapping might define TRUE as T and FALSE as NIL. The mapping must specify the means to construct and operate on these data types in the programming language.

19.1.2 Constructed Data Types

A language mapping must define the means of expressing the constructed data types defined in “Constructed Types” on page 3-25. The ORB defines aggregates of basic data types that are supported, but the language mapping defines how a programmer sees those aggregates. For example, the C mapping might define an OMG IDL struct as a C struct, whereas the LISP mapping might define an OMG IDL struct as a list. The mapping must specify the means to construct and operate on these data types in the programming language.

19.1.3 Constants

OMG IDL definitions may contain named constant values that are useful as parameters for certain operations. The language mapping should provide the means to access these constants by name.

19.1.4 Objects

There are two parts of defining the mapping of ORB objects to a particular language. The first specifies how an object is represented in the program and passed as a parameter to operations. The second is how an object is invoked. The representation of an object reference in a particular language is generally opaque, that is, some language-specific data type is used to represent the object reference, but the program does not interpret the values of that type. The language-specific representation is independent of the ORB representation of an object reference, so that programs are not ORB-dependent. In an object-oriented programming language, it may be convenient to represent an ORB object as a programming language object. Any correspondence between the programming language object types and the OMG IDL types including inheritance, operation names, etc., is up to the language mapping.

There are only three uses that a program can make of an object reference: it may specify it as a parameter to an operation (including receiving it as an output parameter), it can invoke an operation on it, or it can perform an ORB operation (including object adapter operations) on it.

19.1.5 Invocation of Operations

An operation invocation requires the specification of the object to be invoked, the operation to be performed, and the parameters to be supplied. There are a variety of possible mappings, depending to a large extent on the procedure mechanism in the particular language. Some possible choices for language mapping of invocation include: interface-specific stub routines, a single general-purpose routine, a set of calls to construct a parameter list and initiate the operation, or mapping ORB operations to operations on objects defined in an object-oriented programming language.

The mapping must define how parameters are associated with the call, and how the operation name is specified. It is also necessary to specify the effect of the call on the flow of control in the program, including when an operation completes normally and when an exception is raised.

The most natural mapping would be to model a call on an ORB object as the corresponding call in the particular language. However, this may not always be possible for languages where the type system or call mechanism is not powerful enough to handle ORB objects. In this case, multiple calls may be required. For example, in C, it is necessary to have a separate interface for dynamic construction of calls, since C does not permit discovery of new types at runtime. In LISP, however, it may be possible to make a language mapping that is the same for objects whether or not they were known at compile time.

In addition to defining how an operation is expressed, it is necessary to specify the storage allocation policy for parameters, for example, what happens to storage of input parameters, and how and where output parameters are allocated. It is also necessary to describe how a return value is handled, for operations that have one.

19.1.6 Exceptions

There are two aspects to the mapping of exceptions into a particular language. First is the means for handling an exception when it occurs, including deciding which exception occurred. If the programming language has a model of exceptions that can accommodate ORB exceptions, that would likely be the most convenient choice; if it does not, some other means must be used, for example, passing additional parameters to the operations that receive the exception status.

It is commonly the case that the programmer associates specific code to handle each kind of exception. It is desirable to make that association as convenient as possible.

Second, when an exception has been raised, it must be possible to access the parameters of the exception. If the language exception mechanism allows for parameters, that mechanism could be used. Otherwise, some other means of obtaining the exception values must be provided.

19.1.7 Attributes

The ORB models attributes as a pair of operations, one to set and one to get the attribute value. The language mapping defines the means of expressing these operations. One reason for distinguishing attributes from pairs of operations is to allow the language mapping to define the most natural way for accessing them. Some possible choices include defining two operations for each attribute, defining two operations that can set or get, respectively, any attribute, defining operations that can set or get groups of attributes, and so forth.

19.1.8 ORB Interfaces

Most of a language mapping is concerned with how the programmer-defined objects and data are accessed. Programmers who use the ORB must also access some interfaces implemented directly by the ORB, for example, to convert an object reference to a string. A language mapping must also specify how these interfaces appear in the particular programming language.

Various approaches may be taken, including defining a set of library routines, allowing additional ORB-related operations on objects, or defining interfaces that are similar to the language mapping for ordinary objects.

The last approach is called defining pseudo-objects. A pseudo-object has an interface that can (with a few exceptions) be defined in IDL, but is not necessarily implemented as an ORB object. Using stubs a client of a pseudo-object writes calls to it in the same way as if it were an ordinary object. Pseudo-object operations cannot be invoked with the Dynamic Invocation Interface. However, the ORB may recognize such calls as special and handle them directly. One advantage of pseudo-objects is that the interface can be expressed in IDL independent of the particular language mapping, and the programmer can understand how to write calls by knowing the language mapping for the invocations of ordinary objects.

It is not necessary for a language mapping to use the pseudo-object approach. However, this document defines interfaces in subsequent chapters using OMG IDL wherever possible. A language mapping must define how these interfaces are accessed, either by defining them as pseudo-objects and supporting a mapping similar to ordinary objects, by defining language-specific interfaces for them, or in some other way.

19.2 Scoped Names

The C programmer must always use the global name for a type, constant, exception, or operation. The C global name corresponding to an OMG IDL global name is derived by converting occurrences of "**::**" to "_" (an underscore) and eliminating the leading underscore.

Consider the following example:

```
// IDL
typedef string<256> filename_t;
interface example0 {
    enum color {red, green, blue};
    union bar switch (enum foo {room, bell}) { ... };
    •••
};
```

Code to use this interface would look as follows:

```
/* C */
#include "example0.h"

filename_t FN;
example0_color C = example0_red;
example0_bar myUnion;

switch (myUnion._d) {
   case example0_bar_room: •••
   case example0_bar_bell: •••
};
```

Note that the use of underscores to replace the "**::**" separators can lead to ambiguity if the OMG IDL specification contains identifiers with underscores in them. Consider the following example:

```
// IDL
typedef long foo_bar;
interface foo {
    typedef short bar;  /* A legal OMG IDL statement,
    but ambiguous in C */
    •••
};
```

Due to such ambiguities, it is advisable to avoid the indiscriminate use of underscores in identifiers.

19.3 Mapping for Interfaces

All interfaces must be defined at global scope (*no* nested interfaces). The mapping for an interface declaration is as follows:

```
// IDL
interface example1 {
    long op1(in long arg1);
};
```

The preceding example generates the following C declarations[1]:

```
/* C */
typedef CORBA_Object example1;
extern CORBA_long example1_op1(
   example1 o,
   CORBA_long arg1,
   CORBA_Environment *ev
);
```

All object references (typed interface references to an object) are of the well-known, opaque type **CORBA_Object**. The representation of **CORBA_Object** is a pointer. To permit the programmer to decorate a program with typed references, a type with the name of the interface is defined to be a **CORBA_Object**. The literal

CORBA_OBJECT_NIL is legal wherever a **CORBA_Object** may be used; it is guaranteed to pass the **is_nil** operation defined in “Nil Object References” on page 4-5.

OMG IDL permits specifications in which arguments, return results, or components of constructed types may be interface references. Consider the following example:

```
// IDL
#include "example1.idl"

interface example2 {
    example1 op2();
};
```

This is equivalent to the following C declaration:

```
/* C */
#include "example1.h"

typedef CORBA_Object example2;
extern example1 example2_op2(example2 o, CORBA_Environment
*ev);
```

A C fragment for invoking such an operation is as follows:

1. “Implicit Arguments to Operations” on page 19-21 describes the additional arguments added to an operation in the C mapping.

```
/* C */
#include "example2.h"

example1 ex1;
example2 ex2;
CORBA_Environment ev;

/* code for binding ex2 */

ex1 = example2_op2(ex2, &ev);
```

19.4 Inheritance and Operation Names

OMG IDL permits the specification of interfaces that inherit operations from other interfaces. Consider the following example:

```
// IDL
interface example3 : example1 {
    void op3(in long arg3, out long arg4);
};
```

This is equivalent to the following C declarations:

```
/* C */
typedef CORBA_Object example3;
extern CORBA_long example3_op1(
   example3 o,
   CORBA_long arg1,
   CORBA_Environment *ev
);
extern void example3_op3(
   example3 o,
   CORBA_long arg3,
   CORBA_long *arg4,
   CORBA_Environment *ev
);
```

As a result, an object written in C can access **op1** as if it was directly declared in **example3**. Of course, the programmer could also invoke **example1_op1** on an **Object** of type **example3**; the virtual nature of operations in interface definitions will cause invocations of either function to cause the same method to be invoked.

19.5 Mapping for Attributes

The mapping for attributes is best explained through example. Consider the following specification:

```
// IDL
interface foo {
    struct position_t {
        float x, y;
    };

    attribute float radius;
    readonly attribute position_t position;
};
```

This is exactly equivalent to the following illegal OMG IDL specification:

```
// IDL (illegal)
interface foo {
    struct position_t {
        float x, y;
    };

    float       _get_radius();
    void        _set_radius(in float r);
    position_t  _get_position();
};
```

This latter specification is illegal, since OMG IDL identifiers are not permitted to start with the underscore (_) character.

The language mapping for attributes then becomes the language mapping for these equivalent operations. More specifically, the function signatures generated for the above operations are as follows:

```
/* C */
typedef struct foo_position_t {
   CORBA_float x, y;
} foo_position_t;

extern CORBA_float foo__get_radius(foo o, CORBA_Environment
*ev);
extern void foo__set_radius(
   foo o,
   CORBA_float r,
   CORBA_Environment *ev
);
extern foo_position_t foo__get_position(foo o,
CORBA_Environment *ev);
```

Note that two underscore characters (__) separate the name of the interface from the words "**get**" or "**set**" in the names of the functions.

If the "**set**" accessor function fails to set the attribute value, the method should return one of the standard exceptions defined in "Standard Exceptions" on page 3-37.

19.6 Mapping for Constants

Constant identifiers can be referenced at any point in the user's code where a literal of that type is legal. In C, these constants are **`#define`**d.

The fact that constants are **`#define`**d may lead to ambiguities in code. All names which are mandated by the mappings for any of the structured types below start with an underscore.

The mappings for wide character and wide string constants is identical to character and string constants, except that IDL literals are preceded by **`L`** in C. For example, IDL constant:

`const wstring ws = "Hello World";`

would map to

```
#define ws L"Hello World"
```

in C.

19.7 Mapping for Basic Data Types

The basic data types have the mappings shown in Table 19-1 on page 19-10. Implementations are responsible for providing typedefs for CORBA_short, CORBA_long, and so forth. consistent with OMG IDL requirements for the corresponding data types.

Table 19-1 Data Type Mappings

OMG IDL	C
short	CORBA_short
long	CORBA_long
long long	CORBA_long_long
unsigned short	CORBA_unsigned_short
unsigned long	CORBA_unsigned_long
unsigned long long	CORBA_unsigned_long_long
float	CORBA_float
double	CORBA_double
long double	CORBA_long_double
char	CORBA_char
wchar	CORBA_wchar
boolean	CORBA_boolean
any	typedef struct CORBA_any { CORBA_TypeCode _type; void *_value; } CORBA_any;

The C mapping of the OMG IDL **boolean** types is **unsigned char** with only the values 1 (TRUE) and 0 (FALSE) defined; other values produce undefined behavior. CORBA_boolean is provided for symmetry with the other basic data type mappings.

The C mapping of OMG IDL **enum** types is an unsigned integer type capable of representing 2^{32} enumerations. Each enumerator in an **enum** is **#define**d with an appropriate unsigned integer value conforming to the ordering constraints described in "Enumerations" on page 3-27.

TypeCodes are described in "TypeCodes" on page 8-35. The **_value** member for an **any** is a pointer to the actual value of the datum.

The **any** type supports the notion of ownership of its **_value** member. By setting a release flag in the **any** when a value is installed, programmers can control ownership of the memory pointed to by **_value**. The location of this release flag is implementation-dependent, so the following two ORB-supplied functions allow for the setting and checking of the **any** release flag:

```
/* C */
void CORBA_any_set_release(CORBA_any*, CORBA_boolean);
   CORBA_boolean CORBA_any_get_release(CORBA_any*);
```

CORBA_any_set_release can be used to set the state of the release flag. If the flag is set to **TRUE**, the **any** effectively "owns" the storage pointed to by **_value**; if **FALSE**, the programmer is responsible for the storage. If, for example, an **any** is returned from an operation with its release flag set to **FALSE**, calling **CORBA_free()** on the returned **any*** will not deallocate the memory pointed to by **_value**. Before calling **CORBA_free()** on the **_value** member of an **any** directly, the programmer should check the release flag using **CORBA_any_get_release**. If it returns **FALSE**, the programmer should not invoke **CORBA_free()** on the **_value** member; doing so produces undefined behavior. Also, passing a null pointer to either **CORBA_any_set_release** or **CORBA_any_get_release** produces undefined behavior.

If **CORBA_any_set_release** is never called for a given instance of **any**, the default value of the release flag for that instance is **FALSE**.

19.8 Mapping Considerations for Constructed Types

The mapping for OMG IDL structured types (structs, unions, arrays, and sequences) can vary slightly depending on whether the data structure is *fixed-length* or *variable-length*. A type is *variable-length* if it is one of the following types:

- The type **any**
- A bounded or unbounded string or wide string
- A bounded or unbounded sequence
- An object reference or reference to a transmissible pseudo-object
- A struct or union that contains a member whose type is variable-length
- An array with a variable-length element type
- A typedef to a variable-length type

The reason for treating fixed- and variable-length data structures differently is to allow more flexibility in the allocation of **out** parameters and return values from an operation. This flexibility allows a client-side stub for an operation that returns a sequence of strings, for example, to allocate all the string storage in one area that is deallocated in a single call. The mapping of a variable-length type as an **out** parameter or operation return value is a pointer to the associated class or array, as shown in Table 19-2 on page 19-23.

For types whose parameter passing modes require heap allocation, an ORB implementation will provide allocation functions. These types include variable-length **struct**, variable-length **union**, **sequence**, **any**, **string**, **wstring** and array of a variable-length type. The return value of these allocation functions must be freed using **CORBA_free()**. For one of these listed types T, the ORB implementation will provide the following type-specific allocation function:

```
/* C */
T *T__alloc();
```

The functions are defined at global scope using the fully-scoped name of T converted into a C language name (as described in Section 19.2) followed by the suffix "__alloc" (note the double underscore). For **any, string,** and **wstring**, the allocation functions are:

```
/* C */

CORBA_any *CORBA_any_alloc();
char *CORBA_string_alloc();
CORBA_wchar* CORBA_wstring_alloc(CORBA_unsigned_long len);
```

respectively.

19.9 Mapping for Structure Types

OMG IDL structures map directly onto C **struct**s. Note that all OMG IDL types that map to C **struct**s may potentially include padding.

19.10 Mapping for Union Types

OMG IDL discriminated unions are mapped onto C **struct**s. Consider the following OMG IDL declaration:

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

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

```
/* C */
typedef struct {
   CORBA_long _d;
   union {
      CORBA_long x;
      CORBA_float y;
      CORBA_char z;
   } _u;
} Foo;
```

The discriminator in the struct is always referred to as **_d**; the union in the struct is always referred to as **_u.**

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:  printf("x = %ld\n", v->_u.x); break;
   case 2:  printf("y = %f\n", v->_u.y); break;
   default: printf("z = %c\n", v->_u.z); break;
}
```

An ORB implementation need not use a C **union** to hold the OMG IDL **union** elements; a C struct may be used instead. In either case, the programmer accesses the union elements via the **_u** member.

19.11 Mapping for Sequence Types

The OMG IDL data type **sequence** permits passing of unbounded arrays between objects. Consider the following OMG IDL declaration:

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

In C, this is converted to:

```
/* C */
typedef struct {
   CORBA_unsigned_long _maximum;
   CORBA_unsigned_long _length;
   CORBA_long *_buffer;
} vec10;
```

An instance of this type is declared as follows:

```
/* C */
vec10 x = {10L, 0L, (CORBA_long *)NULL);
```

Prior to passing **&x** as an **in** parameter, the programmer must set the **_buffer** member to point to a **CORBA_long** array of 10 elements, and must set the **_length** member to the actual number of elements to transmit.

Prior to passing the address of a **vec10*** as an **out** parameter (or receiving a **vec10*** as the function return), the programmer does nothing. The client stub will allocate storage for the returned sequence; for bounded sequences, it also allocates a buffer of the specified size, while for unbounded sequences, it also allocates a buffer big enough to hold what was returned by the object. Upon successful return from the invocation, the **_maximum** member will contain the size of the allocated array, the **_buffer** member will point at allocated storage, and the **_length** member will contain the number of values that were returned in the **_buffer** member. The client is responsible for freeing the allocated sequence using **CORBA_free()**.

Prior to passing **&x** as an **inout** parameter, the programmer must set the **_buffer** member to point to a **CORBA_long** array of 10 elements. 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. If more data must be returned than the original buffer can hold, the callee can deallocate the original **_buffer** member using **CORBA_free()** (honoring the release flag) and assign **_buffer** to point to new storage.

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

Sequence types support the notion of ownership of their **_buffer** members. By setting a release flag in the sequence when a buffer is installed, programmers can control ownership of the memory pointed to by **_buffer**. The location of this release flag is implementation-dependent, so the following two ORB-supplied functions allow for the setting and checking of the sequence release flag:

```
/* C */
void CORBA_sequence_set_release(void*, CORBA_boolean);
CORBA_boolean CORBA_sequence_get_release(void*);
```

CORBA_sequence_set_release can be used to set the state of the release flag. If the flag is set to **TRUE**, the sequence effectively “owns” the storage pointed to by **_buffer**; if **FALSE**, the programmer is responsible for the storage. If, for example, a sequence is returned from an operation with its release flag set to **FALSE**, calling **CORBA_free()** on the returned sequence pointer will not deallocate the memory pointed to by **_buffer**. Before calling **CORBA_free()** on the **_buffer** member of a sequence directly, the programmer should check the release flag using **CORBA_sequence_get_release**. If it returns **FALSE**, the programmer should not invoke **CORBA_free()** on the **_buffer** member; doing so produces undefined behavior. Also, passing a null pointer or a pointer to something other than a sequence type to either **CORBA_sequence_set_release** or **CORBA_sequence_get_release** produces undefined behavior.

CORBA_sequence_set_release should only be used by the creator of a sequence. If it is not called for a given sequence instance, then the default value of the release flag for that instance is **FALSE**.

Two sequence types are the same type if their sequence element type and size arguments are identical. For example,

```
// IDL
const long SIZE = 25;
typedef long seqtype;

typedef sequence<long, SIZE> s1;
typedef sequence<long, 25> s2;
typedef sequence<seqtype, SIZE> s3;
typedef sequence<seqtype, 25> s4;
```

declares **s1**, **s2**, **s3**, and **s4** to be of the same type.

The OMG IDL type

```
// IDL
sequence<type,size>
```

maps to

```
/* C */
#ifndef _CORBA_sequence_type_defined
#define _CORBA_sequence_type_defined
typedef struct {
   CORBA_unsigned_long _maximum;
   CORBA_unsigned_long _length;
   type *_buffer;
} CORBA_sequence_type;
#endif /* _CORBA_sequence_type_defined */
```

The **ifdef**'s are needed to prevent duplicate definition where the same type is used more than once. The type name used in the C mapping is the type name of the effective type, e.g. in

```
/* C */
typedef CORBA_long FRED;
typedef sequence<FRED,10> FredSeq;
```

the sequence is mapped onto

```
struct { ... } CORBA_sequence_long;
```

If the **type** in

```
// IDL
sequence<type,size>
```

consists of more than one identifier (e.g, unsigned long), then the generated type name consists of the string “CORBA_sequence_” concatenated to the string consisting of the concatenation of each identifier separated by underscores (e.g, “unsigned_long”).

If the **type** is a **string**, the string “string” is used to generate the type name. If the **type** is a **sequence**, the string “sequence” is used to generate the type name, recursively. For example

```
// IDL
sequence<sequence<long> >
```

generates a type of

```
/* C */
CORBA_sequence_sequence_long
```

These generated type names may be used to declare instances of a sequence type.

In addition to providing a type-specific allocation function for each sequence, an ORB implementation must provide a buffer allocation function for each sequence type. These functions allocate vectors of type T for **sequence<T>**. They are defined at global scope and are named similarly to sequences:

```
/* C */
T *CORBA_sequence_T_allocbuf(CORBA_unsigned_long len);
```

Here, “T” refers to the type name. For the type

```
// IDL
sequence<sequence<long> >
```

for example, the sequence buffer allocation function is named

```
/* C */
T *CORBA_sequence_sequence_long_allocbuf
            (CORBA_unsigned_long len);
```

Buffers allocated using these allocation functions are freed using **CORBA_free()**.

19.12 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 through the placement of the 0-byte. Note that the storage for C strings is one byte longer than the stated OMG IDL bound. Consider the following OMG IDL declarations:

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

In C, this is converted to:

```
/* C */
typedef CORBA_char *sten;
typedef CORBA_char *sinf;
```

Instances of these types are declared as follows:

```
/* C */
sten s1 = NULL;
sinf s2 = NULL;
```

Two string types are the same type if their size arguments are identical. For example,

```
/* C */
const long SIZE = 25;

typedef string<SIZE> sx;
typedef string<25> sy;
```

declares **sx** and **sy** to be of the same type.

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. The caller cannot pass a null pointer as the string argument.

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, while 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 using **CORBA_free()**.

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. If the returned string is larger than the original buffer, the client stub will call **CORBA_free()** on the original string and allocate a new buffer for the new string. The client should therefore never pass an **inout** string parameter that was not allocated using **CORBA_string_alloc**. The client is responsible for freeing the allocated storage using **CORBA_free()**, regardless of whether or not a reallocation was necessary.

Strings are dynamically allocated using the following ORB-supplied function:

```
/* C */
CORBA_char *CORBA_string_alloc(CORBA_unsigned_long len);
```

This function allocates **len+1** bytes, enough to hold the string and its terminating NUL character.

Strings allocated in this manner are freed using **CORBA_free()**.

19.13 Mapping for Wide Strings

The mapping for wide strings is similar to that of strings, except that (1) wide strings are mapped to null-terminated (note: a wide null) wide-character arrays instead of 0-byte terminated character arrays; and (2) wide strings are dynamically allocated using the ORB-supplied function:

```
CORBA_wchar* CORBA_wstring_alloc(CORBA_unsigned_long len);
```

instead of **CORBA_string_alloc**. The length argument **len** is the number of CORBA::WChar units to be allocated, including one additional unit for the null terminator.

19.14 Mapping for Fixed

If an implementation has a native fixed-point decimal type, matching the CORBA specifications of the **fixed** type, then the OMG IDL **fixed** type may be mapped to the native type.

Otherwise, the mapping is as follows. Consider the following OMG IDL declarations:

```
fixed<15,5> dec1;                    // IDL
typedef fixed<9,2> money;
```

In C, these become

```
typedef struct {/* C */
CORBA_unsigned_short _digits;
CORBA_short _scale;
CORBA_char _value[(15+2)/2];
} CORBA_fixed_15_5;

CORBA_fixed_15_5 dec1 = {15u, 5};

typedef struct {
CORBA_unsigned_short _digits;
CORBA_short _scale;
 CORBA_char _value[(9+2)/2];
} CORBA_fixed_9_2;

typedef CORBA_fixed_9_2 money;
```

An instance of **money** is declared:

```
money bags = {9u, 2};
```

To permit application portability, the following minimal set of functions and operations on the **fixed** type must be provided by the mapping. Since C does not support parameterized types, the **fixed** arguments are represented as **void*** pointers. The type information is instead conveyed within the representation itself. Thus the **_digits** and **_scale** of every **fixed** operand must be set prior to invoking these functions. Indeed

only the **_value** field of the result, denoted by ***rp**, may be left unset. Otherwise the behavior of the functions is undefined.

```
/* Conversions: all signs are the same.        */
CORBA_long CORBA_fixed_integer_part(const void *fp);
CORBA_long CORBA_fixed_fraction_part(const void *fp);
void CORBA_fixed_set(void *rp, const CORBA_long i,
   const CORBA_long f);

/* Operations, of the form: r = f1 op f2       */
void CORBA_fixed_add(void *rp, const void *f1p,
      const void *f2p);
void CORBA_fixed_sub(void *rp, const void *f1p,
      const void *f2p);
void CORBA_fixed_mul(void *rp, const void *f1p,
      const void *f2p);
void CORBA_fixed_div(void *rp, const void *f1p,
      const void *f2p);
```

These operations must maintain proper fixed-point decimal semantics, following the rules specified in "Semantics" on page 3-20 for the precision and scale of the intermediate results prior to assignment to the result variable. Truncation without rounding may occur if the result type cannot express the intermediate result exactly.

Instances of the **fixed** type are dynamically allocated using the ORB-supplied function:

```
CORBA_fixed_d_s* CORBA_fixed_alloc(CORBA_unsigned_short d);
```

19.15 Mapping for Arrays

OMG IDL arrays map directly to C arrays. All array indices run from 0 to <**size** - 1>.

For each named array type in OMG IDL, the mapping provides a C typedef for pointer to the array's *slice*. A slice of an array is another array with all the dimensions of the original except the first. For example, given the following OMG IDL definition:

```
// IDL
typedef long LongArray[4][5];
```

The C mapping provides the following definitions:

```
/* C */
typedef CORBA_long LongArray[4][5];
typedef CORBA_long LongArray_slice[5];
```

The generated name of the slice typedef is created by appending "_slice" to the original array name.

If the return result, or an **out** parameter for an array holding a variable-length type, of an operation is an array, the array storage is dynamically allocated by the stub; a pointer to the array slice of the dynamically allocated 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 return

the dynamically allocated storage by calling **CORBA_free()**.

An array T of a variable-length type is dynamically allocated using the following ORB-supplied function:

```
/* C */
T_slice *T__alloc();
```

This function is identical to the allocation functions described in Section 19.8, “Mapping Considerations for Constructed Types,” on page 19-11, except that the return type is pointer to array slice, not pointer to array.

19.16 Mapping for Exception Types

Each defined exception type is defined as a struct tag and a typedef with the C global name for the exception. An identifier for the exception, in string literal form, is also **#define**d, as is a type-specific allocation function. For example:

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

yields the following C declarations:

```
/* C */
typedef struct foo {
   CORBA_long dummy;
   /* ...may contain additional
    * implementation-specific members...
    */
} foo;
#define ex_foo <unique identifier for exception>
foo *foo__alloc();
```

The identifier for the exception uniquely identifies this exception type. For example, it could be the Interface Repository identifier for the exception (see “ExceptionDef” on page 8-26).

The allocation function dynamically allocates an instance of the exception and returns a pointer to it. Each exception type has its own dynamic allocation function. Exceptions allocated using a dynamic allocation function are freed using **CORBA_free()**.

Since IDL exceptions are allowed to have no members, but C structs must have at least one member, IDL exceptions with no members map to C structs with one member. This member is opaque to applications. Both the type and the name of the single member are implementation-specific.

19.17 Implicit Arguments to Operations

From the point of view of the C programmer, all operations declared in an interface have additional leading parameters preceding the operation-specific parameters:

1. The first parameter to each operation is a **`CORBA_Object`** input parameter; this parameter designates the object to process the request.
2. The last parameter to each operation is a **`CORBA_Environment*`** output parameter; this parameter permits the return of exception information.
3. If an operation in an OMG IDL specification has a context specification, then a **`CORBA_Context`** input parameter precedes the **`CORBA_Environment*`** parameter and follows any operation-specific arguments.

As described above, the **`CORBA_Object`** type is an opaque type. The **`CORBA_Environment`** type is partially opaque; "Handling Exceptions" on page 19-26 provides a description of the non-opaque portion of the exception structure and an example of how to handle exceptions in client code. The **`CORBA_Context`** type is opaque; see the Dynamic Invocation Interface chapter for more information on how to create and manipulate context objects.

19.18 Interpretation of Functions with Empty Argument Lists

A function declared with an empty argument list is defined to take *no* operation-specific arguments.

19.19 Argument Passing Considerations

For all OMG IDL types (except arrays), if the OMG IDL signature specifies that an argument is an **`out`** or **`inout`** parameter, then the caller must always pass the address of a variable of that type (or the value of a pointer to that type); the callee must dereference the parameter to get to the type. For arrays, the caller must pass the address of the first element of the array.

For **`in`** parameters, the value of the parameter must be passed for all of the basic types, enumeration types, and object references. For all arrays, the address of the first element of the array must be passed. For all other structured types, the address of a variable of that type must be passed, regardless of whether they are fixed- or variable-length. For strings, a **`char*`** and **`wchar*`** must be passed.

For **`inout`** parameters, the address of a variable of the correct type must be passed for all of the basic types, enumeration types, object references, and structured types. For strings, the address of a **`char*`** and the **`*`** of a **`wchar`** the must be passed. For all arrays, the address of the first element of the array must be passed.

Consider the following OMG IDL specification:

```
// IDL
interface foo {
    typedef long Vector[25];

    void bar(out Vector x, out long y);
};
```

Client code for invoking the **bar** operation would look like:

```
/* C */
foo object;
foo_Vector_slice x;
CORBA_long y;
CORBA_Environment ev;

/* code to bind object to instance of foo */

foo_bar(object, &x, &y, &ev);
```

For **out** parameters of type variable-length **struct**, variable-length **union**, **string**, **sequence**, an array holding a variable-length type, or **any**, the ORB will allocate storage for the output value using the appropriate type-specific allocation function. The client may use and retain that storage indefinitely, and must indicate when the value is no longer needed by calling the procedure **CORBA_free**, whose signature is:

```
/* C */
extern void CORBA_free(void *storage);
```

The parameter to **CORBA_free()** is the pointer used to return the **out** parameter. **CORBA_free()** releases the ORB-allocated storage occupied by the **out** parameter, including storage indirectly referenced, such as in the case of a sequence of strings or array of object reference. If a client does not call **CORBA_free()** before reusing the pointers that reference the **out** parameters, that storage might be wasted. Passing a null pointer to **CORBA_free()** is allowed; **CORBA_free()** simply ignores it and returns without error.

19.20 Return Result Passing Considerations

When an operation is defined to return a non-void return result, the following rules hold:

1. If the return result is one of the types **float**, **double**, **long**, **short**, **unsigned long**, **unsigned short**, **char**, **wchar, fixed, boolean**, **octet**, **Object**, or an **enumeration**, then the value is returned as the operation result.

2. If the return result is one of the fixed-length types **struct** or **union**, then the value of the C struct representing that type is returned as the operation result. If the return result is one of the variable-length types **struct**, **union**, **sequence**, or **any**, then a pointer to a C struct representing that type is returned as the operation result.

3. If the return result is of type **string** or **wstring**, then a pointer to the first character of the string is returned as the operation result.

4. If the return result is of type **array**, then a pointer to the slice of the array is returned as the operation result.

Consider the following interface:

```
// IDL
interface X {
    struct y {
        long a;
        float b;
    };

    long op1();
    y op2();
};
```

The following C declarations ensue from processing the specification:

```
/* C */
typedef CORBA_Object X;
typedef struct X_y {
   CORBA_long a;
   CORBA_float b;
} X_y;

extern CORBA_long X_op1(X object, CORBA_Environment *ev);
extern X_y X_op2(X object, CORBA_Environment *ev);
```

For operation results of type variable-length **struct**, variable-length **union**, **wstring**, **string**, **sequence**, **array**, or **any**, the ORB will allocate storage for the return value using the appropriate type-specific allocation function. The client may use and retain that storage indefinitely, and must indicate when the value is no longer needed by calling the procedure **`CORBA_free()`** described in “Argument Passing Considerations” on page 19-21.

19.21 Summary of Argument/Result Passing

Table 19-3 on page 19-24 summarizes what a client passes as an argument to a stub and receives as a result. For brevity, the **`CORBA`**_prefix is omitted from type names in the tables.

Table 19-2 Basic Argument and Result Passing

Data Type	In	Inout	Out	Return
short	short	short*	short*	short
long	long	long*	long*	long
long long	long_long	long_long*	long_long*	long_long
unsigned short	unsigned_short	unsigned_short*	unsigned_short*	unsigned_short
unsigned long	unsigned_long	unsigned_long*	unsigned_long*	unsigned_long
unsigned long long	unsigned_long_long	unsigned_long_long*	unsigned_long_long*	unsigned_long_long

Table 19-2 Basic Argument and Result Passing *(Continued)*

Data Type	In	Inout	Out	Return
float	float	float*	float*	float
double	double	double*	double*	double
long double	long_double	long_double*	long_double*	long_double
fixed<d,s>	fixed_d_s*	fixed_d_s*	fixed_d_s*	fixed_d_s
boolean	boolean	boolean*	boolean*	boolean
char	char	char*	char*	char
wchar	wchar	wchar*	wchar*	wchar
octet	octet	octet*	octet*	octet
enum	enum	enum*	enum*	enum
object reference ptr[1]	objref_ptr	objref_ptr*	objref_ptr*	objref_ptr
struct, fixed	struct*	struct*	struct*	struct
struct, variable	struct*	struct*	struct**	struct*
union, fixed	union*	union*	union*	union
union, variable	union*	union*	union**	union*
string	char*	char**	char**	char*
wstring	wchar*	wchar**	wchar**	wchar*
sequence	sequence*	sequence*	sequence**	sequence*
array, fixed	array	array	array	array slice*[2]
array, variable	array	array	array slice**2	array slice*2
any	any*	any*	any**	any*

1. Including pseudo-object references.
2. A slice is an array with all the dimensions of the original except the first one.

A client is responsible for providing storage for all arguments passed as **in** arguments.

Table 19-3 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
enum	1	1	1
object reference ptr	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3

Table 19-3 Client Argument Storage Responsibilities *(Continued)*

Type	Inout Param	Out Param	Return Result
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3

Table 19-4 Argument Passing Cases

Case[1]	
1	Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller provides the initial value, and the callee may change that value. For out parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.
2	Caller allocates storage for the object reference. For inout parameters, the caller provides an initial value; if the callee wants to reassign the inout parameter, it will first call CORBA_Object_release on the original input value. To continue to use an object reference passed in as an inout, the caller must first duplicate the reference. The client is responsible for the release of all out and return object references. Release of all object references embedded in other out and return structures is performed automatically as a result of calling CORBA_free.
3	For out parameters, the caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned instance into a new instance, then modify the new instance.
4	For inout strings, the caller provides storage for both the input string and the char* pointing to it. The callee may deallocate the input string and reassign the char* to point to new storage to hold the output value. The size of the out string is therefore not limited by the size of the in string. The caller is responsible for freeing the storage for the out. The callee is not allowed to return a null pointer for an inout, out, or return value.
5	For inout sequences and anys, assignment or modification of the sequence or any may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the boolean release in the sequence or any.
6	For out parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance.

1. As listed in Table 19-3 on page 19-24

19.22 Handling Exceptions

Since the C language does not provide native exception handling support, applications pass and receive exceptions via the special **CORBA_Environment** parameter passed to each IDL operation. The **CORBA_Environment** type is partially opaque; the C declaration contains at least the following:

```
/* C */
typedef struct CORBA_Environment {
   CORBA_exception_type _major;
   ...
} CORBA_Environment;
```

Upon return from an invocation, the **_major** field indicates whether the invocation terminated successfully; **_major** can have one of the values **CORBA_NO_EXCEPTION**, **CORBA_USER_EXCEPTION**, or **CORBA_SYSTEM_EXCEPTION**; if the value is one of the latter two, then any exception parameters signalled by the object can be accessed.

Five functions are defined on a **CORBA_Environment** structure for accessing exception information. Their signatures are:

```
/* C */
extern void CORBA_exception_set(
                  CORBA_Environment    *ev,
                  CORBA_exception_type major,
                  CORBA_char           *except_repos_id,
                  void                 *param
               );
extern CORBA_char *CORBA_exception_id(
                              CORBA_Environment *ev
                     );
extern void *CORBA_exception_value(CORBA_Environment *ev);
extern void CORBA_exception_free(CORBA_Environment *ev);
extern CORBA_any* CORBA_exception_as_any(
                              CORBA_Environment *ev
                     );
```

CORBA_exception_set() allows a method implementation to raise an exception. The **ev** parameter is the environment parameter passed into the method. 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_NO_EXCEPTION**, this is a normal outcome to the operation. In this case, both **except_repos_id** and **param** must be NULL. Note that it is *not* necessary to invoke **CORBA_exception_set()** to indicate a normal outcome; it is the default behavior if the method simply returns.

- For any other value of **major** it specifies either a user-defined or system exception. The **except_repos_id** parameter is the repository ID representing the exception type. If the exception is declared to have members, the **param** parameter must be the address of an instance of the exception struct containing the parameters according to the C language mapping, coerced to a **void***. In this case, the exception struct must be allocated using the appropriate **T__alloc()** function, and the **CORBA_exception_set()** function adopts the allocated memory and frees it when it no longer needs it. Once the allocated exception struct is passed to **CORBA_exception_set()**, the application is not allowed to access it because it no longer owns it. If the exception takes no parameters, **param** must be NULL.

If the **CORBA_Environment** argument to **CORBA_exception_set()** already has an exception set in it, that exception is properly freed before the new exception information is set.

CORBA_exception_id() returns a pointer to the character string identifying the exception. The character string contains the repository ID for the exception. If invoked on a **CORBA_Environment** which identifies a non-exception, (**_major==CORBA_NO_EXCEPTION**) a null pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until **CORBA_exception_free()** is called.

CORBA_exception_value() returns a pointer to the structure corresponding to this exception. If invoked on a **CORBA_Environment** which identifies a non-exception or an exception for which there is no associated information, a null pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until **CORBA_exception_free()** is called.

CORBA_exception_free() frees any storage which was allocated in the construction of the **CORBA_Environment** or adopted by the **CORBA_Environment** when **CORBA_exception_set()** is called on it, and sets the **_major** field to **CORBA_NO_EXCEPTION**. It is permissible to invoke **CORBA_exception_free()** regardless of the value of the **_major** field.

CORBA_exception_as_any() returns a pointer to a **CORBA_any** containing the exception. This allows a C application to deal with exceptions for which it has no static (compile-time) information. If invoked on a **CORBA_Environment** which identifies a non-exception, a null pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until **CORBA_exception_free()** is called.

Consider the following example:

```
// IDL
interface exampleX {
    exception BadCall {
        string<80> reason;
    };

    void op() raises(BadCall);
};
```

This interface defines a single operation which returns no results and can raise a **BadCall** exception. The following user code shows how to invoke the operation and recover from an exception:

```
/* C */
#include "exampleX.h"

CORBA_Environment ev;
exampleX obj;
exampleX_BadCall *bc;

/*
 * some code to initialize obj to a reference to an object
 * supporting the exampleX interface
 */

exampleX_op(obj, &ev);
switch(ev._major) {
case CORBA_NO_EXCEPTION:/* successful outcome*/
   /* process out and inout arguments */
   break;
case CORBA_USER_EXCEPTION:/* a user-defined exception */
   if (strcmp(ex_exampleX_BadCall,
               CORBA_exception_id(&ev)) == 0) {
      bc = (exampleX_BadCall*)CORBA_exception_value(&ev);
      fprintf(stderr, "exampleX_op() failed - reason: %s\n",
         bc->reason);
   }
   else {   /* should never get here ... */
         fprintf( stderr,
                  "unknown user-defined exception -%s\n",
                  CORBA_exception_id(&ev));
   }
   break;
default:/* standard exception */
   /*
    * CORBA_exception_id() can be used to determine
    * which particular standard exception was
    * raised; the minor member of the struct
    * associated with the exception (as yielded by
    * CORBA_exception_value()) may provide additional
    * system-specific information about the exception
```

```
    */
   break;
}
/* free any storage associated with exception */
CORBA_exception_free(&ev);
```

19.23 Method Routine Signatures

The signatures of the methods used to implement an object depend not only on the language binding, but also on the choice of object adapter. Different object adapters may provide additional parameters to access object adapter-specific features.

Most object adapters are likely to provide method signatures that are similar in most respects to those of the client stubs. In particular, the mapping for the operation parameters expressed in OMG IDL should be the same as for the client side.

See “Mapping for Object Implementations” on page 19-30 for the description of method signatures for implementations using the Portable Object Adapter.

19.24 Include Files

Multiple interfaces may be defined in a single source file. By convention, each interface is stored in a separate source file. All OMG IDL compilers will, by default, generate a header file named **Foo.h** from **Foo.idl**. This file should be **#include**d by clients and implementations of the interfaces defined in **Foo.idl**.

Inclusion of **Foo.h** is sufficient to define all global names associated with the interfaces in **Foo.idl** and any interfaces from which they are derived.

19.25 Pseudo-objects

In the C language mapping, there are several interfaces that are defined as pseudo-objects; A client makes calls on a pseudo-object in the same way as an ordinary ORB object. However, the ORB may implement the pseudo-object directly, and there are restrictions on what a client may do with a pseudo-object.

The ORB itself is a pseudo-object with the following partial definition (see the ORB Interface chapter for the complete definition):

```
// IDL
interface ORB {
    string      object_to_string (in Object obj);
    Object      string_to_object (in string str);
};
```

This means that a C programmer may convert an object reference into its string form by calling:

```
/* C */
CORBA_Environment ev;
CORBA_char *str = CORBA_ORB_object_to_string(
                        orbobj, obj, &ev
                    );
```

just as if the ORB were an ordinary object. The C library contains the routine **`CORBA_ORB_object_to_string`**, and it does not do a real invocation. The **`orbobj`** is an object reference that specifies which ORB is of interest, since it is possible to choose which ORB should be used to convert an object reference to a string (see the ORB Interface chapter for details on this specific operation).

Although operations on pseudo-objects are invoked in the usual way defined by the C language mapping, there are restrictions on them. In general, a pseudo-object cannot be specified as a parameter to an operation on an ordinary object. Pseudo-objects are also not accessible using the dynamic invocation interface, and do not have definitions in the interface repository.

Because the programmer uses pseudo-objects in the same way as ordinary objects, some ORB implementations may choose to implement some pseudo-objects as ordinary objects. For example, assuming it could be efficient enough, a context object might be implemented as an ordinary object.

19.25.1 ORB Operations

The operations on the ORB defined in the ORB Interface chapter are used as if they had the OMG IDL definitions described in the document, and then mapped in the usual way with the C language mapping.

For example, the **string_to_object** ORB operation has the following signature:

```
/* C */
CORBA_Object CORBA_ORB_string_to_object(
   CORBA_Object      orb,
   CORBA_char        *objectstring,
   CORBA_Environment *ev
);
```

Although in this example, we are using an “object” that is special (an ORB), the method name is generated as **`interface_operation`** in the same way as ordinary objects. Also, the signature contains an **`CORBA_Environment`** parameter for error indications.

Following the same procedure, the C language binding for the remainder of the ORB and object reference operations may be determined.

19.26 Mapping for Object Implementations

This section describes the details of the OMG IDL-to-C language mapping that apply specifically to the Portable Object Adapter, such as how the implementation methods are connected to the skeleton.

19.26.1 Operation-specific Details

The C Language Mapping Chapter defines most of the details of binding methods to skeletons, naming of parameter types, and parameter passing conventions. Generally, for those parameters that are operation-specific, the method implementing the operation appears to receive the same values that would be passed to the stubs.

19.26.2 PortableServer Functions

Objects registered with POAs use sequences of octet, specifically the **PortableServer::POA::ObjectId** type, as object identifiers. However, because C programmers will often want to use strings as object identifiers, the C mapping provides several conversion functions that convert strings to **ObjectId** and vice-versa:

```
/* C */
extern CORBA_char* PortableServer_ObjectId_to_string(
                     PortableServer_ObjectId* id,
                     CORBA_Environment* env
                );
extern CORBA_wchar_t* PortableServer_ObjectId_to_wstring(
                     PortableServer_ObjectId* id
                     CORBA_Environment* env
                );

extern PortableServer_ObjectId*
                PortableServer_string_to_ObjectId(
                         CORBA_char* str,
                         CORBA_Environment* env
                );
extern PortableServer_ObjectId*
                PortableServer_wstring_to_ObjectId(
                         CORBA_wchar_t* str,
                         CORBA_Environment* env
                );
```

These functions follow the normal C mapping rules for parameter passing and memory management.

If conversion of an **ObjectId** to a string would result in illegal characters in the string (such as a NUL), the first two functions raise the **CORBA_BAD_PARAM** exception.

19.26.3 Mapping for PortableServer::ServantLocator::Cookie

Since **PortableServer::ServantLocator::Cookie** is an IDL **native** type, its type must be specified by each language mapping. In C, **Cookie** maps to **void***:

```
/* C */
typedef void* PortableServer_ServantLocator_Cookie;
```

For the C mapping of the **PortableServer::ServantLocator::preinvoke()** operation, the **Cookie** parameter maps to a **Cookie***, while for the **postinvoke()** operation, it is passed as a **Cookie**:

```
/* C */
extern PortableServer_ServantLocator_preinvoke(
            PortableServer_ObjectId* oid,
            PortableServer_POA adapter,
            CORBA_Identifier op_name,
            PortableServer_ServantLocator_Cookie* cookie
          );
extern PortableServer_ServantLocator_postinvoke(
            PortableServer_ObjectId* oid,
            PortableServer_POA adapter,
            CORBA_Identifier op_name,
            PortableServer_ServantLocator_Cookie cookie,
            PortableServer_Servant servant
          );
```

19.26.4 Servant Mapping

A *servant* is a language-specific entity that can incarnate a CORBA object. In C, a servant is composed of a data structure that holds the state of the object along with a collection of *method functions* that manipulate that state in order to implement the CORBA object.

The **PortableServer::Servant** type maps into C as follows:

```
/* C */
typedef void* PortableServer_Servant;
```

Servant is mapped to a **void*** rather than a pointer to **ServantBase** so that all servant types for derived interfaces can be passed to all the operations that take a **Servant** parameter without requiring casting. However, it is expected that an instance of **PortableServer_Servant** points to an instance of a **PortableServer_ServantBase** or its equivalent for derived interfaces, as described below.

Associated with a servant is a table of pointers to method functions. This table is called an *entry point vector*, or EPV. The EPV has the same name as the servant type with "__epv" appended (note the double underscore). The EPV for **PortableServer_Servant** is defined as follows:

```
/* C */
typedef struct PortableServer_ServantBase__epv {
   void* _private;
   void (*finalize)(PortableServer_Servant,
                    CORBA_Environment*);
   PortableServer_POA (*default_POA)(
                           PortableServer_Servant,
                           CORBA_Environment*);
} PortableServer_ServantBase__epv;

extern PortableServer_POA
PortableServer_ServantBase__default_POA(
         PortableServer_Servant,
         CORBA_Environment*
);
```

The **PortableServer_ServantBase__epv** "_private" member, which is opaque to applications, is provided to allow ORB implementations to associate data with each **ServantBase** EPV. Since it is expected that EPVs will be shared among multiple servants, this member is not suitable for per-servant data. The second member is a pointer to the finalization function for the servant, which is invoked when the servant is ethereal-ized. The other function pointers correspond to the usual **Servant** operations.

The actual **PortableServer_ServantBase** structure combines an EPV with per-servant data, as shown below:

```
/* C */
typedef PortableServer_ServantBase__epv*
               PortableServer_ServantBase__vepv;

typedef struct PortableServer_ServantBase {
   void* _private;
   PortableServer_ServantBase__vepv* vepv;
} PortableServer_ServantBase;
```

The first member is a **void*** that points to data specific to each ORB implementation. This member, which allows ORB implementations to keep per-servant data, is opaque to applications. The second member is a pointer to a pointer to a **PortableServer_ServantBase__epv**. The reason for the double level of indirection is that servants for derived classes contain multiple EPV pointers, one for each base interface as well as one for the interface itself. (This is explained further in the next-section.) The name of the second member, "vepv," is standardized to allow portable access through it.

19.26.5 Interface Skeletons

All C skeletons for IDL interfaces have essentially the same structure as ServantBase, with the exception that the second member has a type that allows access to all EPVs for the servant, including those for base interfaces as well as for the most-derived interface.

For example, consider the following IDL interface:

```
// IDL
interface Counter {
    long add(in long val);
};
```

The servant skeleton generated by the IDL compiler for this interface appears as follows (the type of the second member is defined further below):

```
/* C */
typedef struct POA_Counter {
   void* _private;
   POA_Counter__vepv* vepv;
} POA_Counter;
```

As with **PortableServer_ServantBase**, the name of the second member is standardized to "vepv" for portability.

The EPV generated for the skeleton is a bit more interesting. For the **Counter** interface defined above, it appears as follows:

```
/* C */
typedef struct POA_Counter__epv {
   void* _private;
   CORBA_Long (*add)(PortableServer_Servant servant,
                     CORBA_Long val,
                     CORBA_Environment* env);
} POA_Counter__epv;
```

Since all servants are effectively derived from **PortableServer_ServantBase**, the complete set of entry points has to include EPVs for both **PortableServer_ServantBase** and for **Counter** itself:

```
/* C */
typedef struct POA_Counter__vepv {
   PortableServer_ServantBase__epv* _base_epv;
   POA_Counter__epv* Counter_epv;
} POA_Counter__vepv;
```

The first member of the **POA_Counter__vepv** struct is a pointer to the **PortableServer_ServantBase** EPV. To ensure portability of initialization and access code, this member is always named "_base_epv." It must always be the first member. The second member is a pointer to a **POA_Counter__epv**.

The pointers to EPVs in the VEPV structure are in the order that the IDL interfaces appear in a top-to-bottom left-to-right traversal of the inheritance hierarchy of the most-derived interface. The base of this hierarchy, as far as servants are concerned, is always **PortableServer_ServantBase**. For example, consider the following complicated interface hierarchy:

```
// IDL
interface A {};
interface B : A {};
interface C : B {};
interface D : B {};
interface E : C, D {};
interface F {};
interface G : E, F {
    void foo();
};
```

The VEPV structure for interface **G** shall be generated as follows:

```
/* C */
typedef struct POA_G__epv {
   void* _private;
   void (*foo)(PortableServer_Servant, CORBA_Environment*);
};
typedef struct POA_G__vepv {
   PortableServer_ServantBase__epv* _base_epv;
   POA_A__epv* A_epv;
   POA_B__epv* B_epv;
   POA_C__epv* C_epv;
   POA_D__epv* D_epv;
   POA_E__epv* E_epv;
   POA_F__epv* F_epv;
   POA_G__epv* G_epv;
};
```

Note that each member other than the "_base_epv" member is named by appending "_epv" to the interface name whose EPV the member points to. These names are standardized to allow for portable access to these struct fields.

19.26.6 Servant Structure Initialization

Each servant requires initialization and etherealization, or finalization, functions. For **`PortableServer_ServantBase`**, the ORB implementation shall provide the following functions:

```
/* C */
void PortableServer_ServantBase__init(
         PortableServer_Servant,
         CORBA_Environment*);
void PortableServer_ServantBase__fini(
         PortableServer_Servant,
         CORBA_Environment*);
```

These functions are named by appending "__init" and "__fini" (note the double underscores) to the name of the servant, respectively.

The first argument to the init function shall be a valid **PortableServer_Servant** whose "vepv" member has already been initialized to point to a VEPV structure. The init function shall perform ORB-specific initialization of the **PortableServer_ServantBase**, and shall initialize the "finalize" struct member of the pointed-to **PortableServer_ServantBase__epv** to point to the **PortableServer_ServantBase_fini()** function if the "finalize" member is NULL. If the "finalize" member is not NULL, it is presumed that it has already been correctly initialized by the application, and is thus not modified. Similarly, if the the **default_POA** member of the **PortableServer_ServantBase__epv** structure is NULL when the init function is called, its value is set to point to the **PortableServer_ServantBase__default_POA()** function, which returns an object reference to the root POA.

If a servant pointed to by the **PortableServer_Servant** passed to an init function has a NULL "vepv" member, or if the **PortableServer_Servant** argument itself is NULL, no initialization of the servant is performed, and the **CORBA::BAD_PARAM** standard exception is raised via the **CORBA_Environment** parameter. This also applies to interface-specific init functions, which are described below.

The fini function only cleans up ORB-specific private data. It is the default finalization function for servants. It does not make any assumptions about where the servant is allocated, such as assuming that the servant is heap-allocated and trying to call **CORBA_free()** on it. Applications are allowed to "override" the fini function for a given servant by initializing the **PortableServer_ServantBase__epv** "finalize" pointer with a pointer to a finalization function made specifically for that servant; however, any such overriding function must always ensure that the **PortableServer_ServantBase_fini()** function is invoked for that servant as part of its implementation. The results of a finalization function failing to invoke **PortableServer_ServantBase_fini()** are implementation-specific, but may include memory leaks or faults that could crash the application.

If a servant passed to a fini function has a NULL "epv" member, or if the **PortableServer_Servant** argument itself is NULL, no finalization of the servant is performed, and the **CORBA::BAD_PARAM** standard exception is raised via the **CORBA_Environment** parameter. This also applies to interface-specific fini functions, which are described below.

Normally, the **PortableServer_ServantBase__init** and **PortableServer_ServantBase__fini** functions are not invoked directly by applications, but rather by interface-specific initialization and finalization functions generated by an IDL compiler. For example, the init and fini functions generated for the **Counter** skeleton are defined as follows:

```
/* C */
void POA_Counter__init(POA_Counter* servant,
                       CORBA_Environment* env)
{
   /*
    * first call immediate base interface init functions
    * in the left-to-right order of inheritance
    */
   PortableServer_ServantBase__init(
            (PortableServer_ServantBase*)servant,
            env
   );
   /* now perform POA_Counter initialization */
   ...
}

void POA_Counter__fini(POA_Counter* servant,
                       CORBA_Environment* env)
{
   /* first perform POA_Counter cleanup */
   ...
   /*
    * then call immediate base interface fini functions
    * in the right-to-left order of inheritance
    */
   PortableServer_ServantBase__fini(
            (PortableServer_ServantBase*)servant,
            env
   );
}
```

The address of a servant shall be passed to the init function before the servant is allowed to be activated or registered with the POA in any way. The results of failing to properly initialize a servant via the appropriate init function before registering it or allowing it to be activated are implementation-specific, but could include memory access violations that could crash the application.

19.26.7 Application Servants

It is expected that applications will create their own servant structures so that they can add their own servant-specific data members to store object state. For the **`Counter`** example shown above, an application servant would probably have a data member used to store the counter value:

```
/* C */
typedef struct AppServant {
   POA_Counter base;
   CORBA_Long value;
} AppServant;
```

The application might contain the following implementation of the **Counter::add** operation:

```
/* C */
CORBA_Long
app_servant_add(PortableServer_Servant _servant,
                CORBA_Long val,
                CORBA_Environment* _env)
{
   AppServant* self = (AppServant*)_servant;
   self->value += val;
   return self->value;
}
```

The application could initialize the servant statically as follows:

```
/* C */
PortableServer_ServantBase__epv base_epv = {
   NULL,          /* ignore ORB private data */
   NULL,          /* no servant-specific finalize
                     function needed */
   NULL,          /* use base default_POA function */
};

POA_Counter__epv counter_epv = {
   NULL,             /* ignore ORB private data */
   app_servant_add   /* point to our add function */
};

/* Vector of EPVs */
POA_Counter__vepv counter_vepv = {
   &base_epv,
   &counter_epv
};




};
AppServant my_servant = {
   /* initialize POA_Counter */
   {
      NULL,          /* ignore ORB private data */
      &counter_vepv  /* Counter vector of EPVs */
   },
   0           /* initialize counter value */
};
```

Before registering or activating this servant, the application shall call:

```
/* C */
CORBA_Environment env;
POA_Counter__init(&my_servant, &env);
```

If the application requires a special destruction function for **my_servant**, it shall set the value of the **PortableServer_ServantBase__epv** "finalize" member either before or after calling **POA_Counter__init()**:

```
/* C */
my_servant.epv._base_epv.finalize = my_finalizer_func;
```

Note that if the application statically initialized the "finalize" member before calling the servant initialization function, explicit assignment to the "finalize" member as shown here is not necessary, since the **PortableServer_ServantBase__init()** function will not modify it if it is non-NULL.

The example shown above illustrates static initialization of the EPV and VEPV structures. While portable, this method of initialization depends on the ordering of the VEPV struct members for base interfaces—if the top-to-bottom left-to-right ordering of the interface inheritance hierarchy is changed, the order of these fields is also changed. A less fragile way of initializing these fields is to perform the initialization at runtime, relying on assignment to the named struct fields. Since the names of the fields are used in this approach, it does not break if the order of base interfaces changes. Performing field initialization within a servant initialization function also provides a convenient place to invoke the servant initialization functions. In any case, both approaches are portable, and it is ultimately up to the developer to choose the one that is best for each application.

19.26.8 Method Signatures

With the POA, implementation methods have signatures that are identical to the stubs except for the first argument. If the following interface is defined in OMG IDL:

```
// IDL
interface example4 {
    long op5(in long arg6);
};
```

a method function for the **op5** operation must have the following function signature:

```
/* C */
CORBA_long example4_op5(
   PortableServer_Servant  _servant,
   CORBA_long              arg6,
   CORBA_Environment*      _env
);
```

The **_servant** parameter is the pointer to the servant incarnating the CORBA object on which the request was invoked. The method can obtain the object reference for the target CORBA object by using the **POA_Current** object. The **_env** parameter is

used for raising exceptions. Note that the names of the **_servant** and **_env** parameters are standardized to allow the bodies of method functions to refer to them portably.

The method terminates successfully by executing a **return** statement returning the declared operation value. Prior to returning the result of a successful invocation, the method code must assign legal values to all **out** and **inout** parameters.

The method terminates with an error by executing the **CORBA_exception_set** operation (described in "Handling Exceptions" on page 19-26) prior to executing a **return** statement. When raising an exception, the method code is *not* required to assign legal values to any **out** or **inout** parameters. Due to restrictions in C, however, it must return a legal function value.

19.27 Mapping of the Dynamic Skeleton Interface to C

For general information about mapping of the Dynamic Skeleton Interface to programming languages, refer to "DSI: Language Mapping" on page 6-4.

This section contains

- A mapping of the Dynamic Skeleton Interface's ServerRequest to C
- A mapping of the Portable Object Adapter's Dynamic Implementation Routine to C.

19.27.1 Mapping of ServerRequest to C

In the C mapping, a ServerRequest is a pseudo object in the CORBA module that supports the following operations:

```
/* C */
CORBA_Identifier  CORBA_ServerRequest_operation(
                     CORBA_ServerRequest req,
                     CORBA_Environment *env
                  );
```

This function returns the name of the operation being performed, as shown in the operation's OMG IDL specification.

```
/* C */
CORBA_Context CORBA_ServerRequest_ctx (
                  CORBA_ServerRequest req,
                  CORBA_Environment *env
               );
```

This function may be used to determine any context values passed as part of the operation. Context will only be available to the extent defined in the operation's OMG IDL definition; for example, attribute operations have none.

```
/* C */
void CORBA_ServerRequest_arguments(
         CORBA_ServerRequest req,
         CORBA_NVList* parameters,
         CORBA_Environment *env
      );
```

This function is used to retrieve parameters from the **ServerRequest**, and to find the addresses used to pass pointers to result values to the ORB. It must always be called by each DIR, even when there are no parameters.

The caller passes ownership of the **parameters** NVList to the ORB. Before this routine is called, that NVList should be initialized with the TypeCodes and direction flags for each of the parameters to the operation being implemented: *in*, *out*, and *inout* parameters inclusive. When the call returns, the **parameters** NVList is still usable by the DIR, and all *in* and *inout* parameters will have been unmarshalled. Pointers to those parameter values will at that point also be accessible through the **parameters** NVList.

The implementation routine will then process the call, producing any result values. If the DIR does not need to report an exception, it will replace pointers to *inout* values in parameters with the values to be returned, and assign pointers to *out* values in that NVList appropriately as well. When the DIR returns, all the parameter memory is freed as appropriate, and the NVList itself is freed by the ORB.

```
/* C */
void  CORBA_ServerRequest_set_result(
         CORBA_ServerRequest req,
         CORBA_any* value,
         CORBA_Environment *env
      );
```

This function is used to report any result **value** for an operation. If the operation has no result, it must either be called with a tk_void TypeCode stored in **value**, or not be called at all.

```
/* C */
void  CORBA_ServerRequest_set_exception(
         CORBA_ServerRequest req,
         CORBA_exception_type major,
         CORBA_any* value,
         CORBA_Environment *env
      );
```

This function is used to report exceptions, both user and system, to the client who made the original invocation. The parameters are as follows:

major indicates whether the exception is a user exception or system exception

value is the value of the exception, including an exceptionTypeCode.

19.27.2 Mapping of Dynamic Implementation Routine to C

In C, a DIR is a function with this signature:

```
/* C */
typedef void (*PortableServer_DynamicImplRoutine)(
   PortableServer_Servant     servant,
   CORBA_ServerRequest        request
);
```

Such a function will be invoked by the Portable Object Adapter when an invocation is received on an object reference whose implementation has registered a dynamic skeleton.

servant is the C implementation object incarnating the CORBA object to which the invocation is directed.

request is the ServerRequest used to access explicit parameters and report results (and exceptions).

Unlike other C object implementations, the DIR does not receive a **`CORBA_Environment*`** parameter, and so the **`CORBA_exception_set`** API is not used. Instead, **`CORBA_ServerRequest_set_exception`** is used; this provides the TypeCode for the exception to the ORB, so it does not need to consult the Interface Repository (or rely on compiled stubs) to marshal the exception value.

To register a Dynamic Implementation Routine with a POA, the proper EPV structure and servant must first be created. DSI servants are expected to supply EPVs for both **`PortableServer_ServantBase`** and for **`PortableServer_DynamicImpl`**, which is conceptually derived from **`PortableServer_ServantBase`**, as shown below.

```
/* C */
typedef struct PortableServer_DynamicImpl__epv {
   void* _private;
   PortableServer_DynamicImplRoutine invoke;
   CORBA_RepositoryId (*primary_interface)(
                              PortableServer_Servant svt,
                              PortableServer_ObjectId id,
                              PortableServer_POA poa,
                              CORBA_Environment* env);
} PortableServer_DynamicImpl__epv;

typedef struct PortableServer_DynamicImpl__vepv {
   PortableServer_ServantBase__epv* _base_epv;
   PortableServer_DynamicImpl__epv*
               PortableServer_DynamicImpl_epv;
} PortableServer_DynamicImpl__vepv;

typedef struct PortableServer_DynamicImpl {
   void* _private;
   PortableServer_DynamicImpl__vepv* vepv;
} PortableServer_DynamicImpl;
```

As for other servants, initialization and finalization functions for **`PortableServer_DynamicImpl`** are also provided, and must be invoked as described in “Servant Structure Initialization” on page 19-35.

To properly initialize the EPVs, the application must provide implementations of the **`invoke`** and the **`primary_interface`** functions required by the **`PortableServer_DynamicImpl`** EPV. The **`invoke`** method, which is the DIR, receives requests issued to any CORBA object it represents and performs the processing necessary to execute the request.

The **`primary_interface`** method receives an **ObjectId** value and a POA as input parameters and returns a valid Interface Repository Id representing the most-derived interface for that **oid**.

It is expected that these methods will be only invoked by the POA, in the context of serving a CORBA request. Invoking these methods in other circumstances may lead to unpredictable results.

An example of a DSI-based servant is shown below:

```
/* C */

/* This function serves as the DIR */
void my_invoke(PortableServer_Servant servant,
                  CORBA_ServerRequest req)
{
   /* details omitted */
}

CORBA_RepositoryId my_primary_intf(
```

```
                        PortableServer_Servant svt,
                        PortableServer_ObjectId id,
                        PortableServer_POA poa,
                        CORBA_Environment* env)
{
   /* details omitted */
}

/* Application-specific DSI servant type */
typedef struct MyDSIServant {
   POA_DynamicImpl base;
   /* other application-specific data members */
} MyDSIServant;

PortableServer_ServantBase__epv base_epv = {
   NULL,             /* ignore ORB private data */
   NULL,             /* no servant-specific finalize */
   NULL,             /* use base default_POA function */
};
PortableServer_DynamicImpl__epv dynimpl_epv = {
   NULL,             /* ignore ORB private data */
   my_invoke,        /* invoke() function */
   my_primary_intf,  /* primary_interface() function */
};
PortableServer_DynamicImpl__vepv dynimpl_vepv = {
   &base_epv,        /* ServantBase EPV */
   &dynimpl_epv,     /* DynamicImpl EPV */
};

MyDSIServant my_servant = {
   /* initialize PortableServer_DynamicImpl */
   {
      NULL,          /* ignore ORB private data */
      &dynimpl_vepv  /* DynamicImpl vector of EPVs */
   };
   /* initialize application-specific data members */
};
```

Registration of the **`my_servant`** data structure via the **`PortableServer_POA_set_servant()`** function on a suitably initialized POA makes the **`my_invoke`** DIR function available to handle DSI requests.

19.28 ORB Initialization Operations

ORB Initialization

The following PIDL specifies initialization operations for an ORB; this PIDL is part of the CORBA module (not the ORB interface) and is described in “ORB Initialization” on page 4-8.

```
// PIDL
module CORBA {
    typedef string ORBid;
    typedef sequence <string> arg_list;

    ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

The mapping of the preceding PIDL operations to C is as follows:

```
/* C */
typedef char* CORBA_ORBid;
extern CORBA_ORB CORBA_ORB_init(int *argc,
                                char **argv,
                                CORBA_ORBid orb_identifier,
                                CORBA_Environment *env);
```

The C mapping for **ORB_init** deviates from the OMG IDL PIDL in its handling of the **arg_list** parameter. This is intended to provide a meaningful PIDL definition of the initialization interface, which has a natural C binding. To this end, the **arg_list** structure is replaced with **`argv`** and **`argc`** parameters.

The **`argv`** parameter is defined as an unbound array of strings (**`char **`**) and the number of strings in the array is passed in the **`argc`** (**`int*`**) parameter.

If an empty ORBid string is used then argc arguments can be used to determine which ORB should be returned. This is achieved by searching the **`argv`** parameters for one tagged *ORBid*, e.g., *-ORBid "ORBid_example."* If an empty ORBid string is used and no ORB is indicated by the **`argv`** parameters, the default ORB is returned.

Regardless of whether an empty or non-empty ORBid string is passed to **`ORB_init`**, the **`argv`** arguments are examined to determine if any ORB parameters are given. If a non-empty ORBid string is passed to **`ORB_init`**, all *-ORBid* parameters in the **`argv`** are ignored. All other *-ORB<suffix>* parameters may be of significance during the ORB initialization process.

For C, the order of consumption of **`argv`** parameters may be significant to an application. In order to ensure that applications are not required to handle **`argv`** parameters they do not recognize the ORB initialization function must be called before the remainder of the parameters are consumed. Therefore, after the **`ORB_init`** call the **`argv`** and **`argc`** parameters will have been modified to remove the ORB understood arguments. It is important to note that the ORB_init call can only reorder or remove references to parameters from the argv list; this restriction is made in order to avoid potential memory management problems caused by trying to free parts of the argv list or extending the argv list of parameters. This is why **`argv`** is passed as a **`char**`** and not a **`char***`**.