Mapping of OMG IDL to C++ 20

This chapter explains how OMG IDL constructs are mapped to the constructs of the C++ programming language. It provides mapping information for:

- Interfaces
- Constants
- Basic data types
- Enums
- Types (string, structure, struct, union, sequence, array, typedefs, any, exception)
- Operations and attributes
- Arguments

Contents

This chapter contains the following sections.

20.1 Preliminary Information

20.1.1 Overview

Key Design Decisions

The design of the C++ mapping was driven by a number of considerations, including a design that achieves reasonable performance, portability, efficiency, and usability for OMG IDL-to-C++ implementations. Several other considerations are outlined in this section.

For more information about the general requirements of a mapping from OMG IDL to any programming language, refer to “Requirements for a Language Mapping” on page 19-2.

Compliance

The C++ mapping tries to avoid limiting the implementation freedoms of ORB developers. For each OMG IDL and CORBA construct, the C++ mapping explains the syntax and semantics of using the construct from C++. A client or server program conforms to this mapping (is CORBA-C++ compliant) if it uses the constructs as described in the C++ mapping chapters. An implementation conforms to this mapping if it correctly executes any conforming client or server program. A conforming client or server program is therefore portable across all conforming implementations. For more information about CORBA compliance, refer to the Preface, “Definition of CORBA Compliance” on page -xxvi.

C++ Implementation Requirements

The mapping proposed here assumes that the target C++ environment supports all the features described in *The Annotated C++ Reference Manual* (ARM) by Ellis and Stroustrup as adopted by the ANSI/ISO C++ standardization committees, including exception handling. In addition, it assumes that the C++ environment supports the **`namespace`** construct recently adopted into the language. Because C++ implementations vary widely in the quality of their support for templates, this mapping does not explicitly require their use, nor does it disallow their use as part of a CORBA-compliant implementation.

C Data Layout Compatibility

Some ORB vendors feel strongly that the C++ mapping should be able to work directly with the CORBA C mapping. This mapping makes every attempt to ensure compatibility between the C and C++ mappings, but it does not mandate such compatibility. In addition to providing better interoperability and portability, the C++ call style solves the memory management problems seen by C programmers who use the C call style. Therefore, the OMG has adopted the C++ call style for OMG IDL.

However, to provide continuity for earlier applications, an implementation might choose to support the C call style as an option. If an implementation supports both call styles, it is recommended that the C call style be phased out.

Note that the mapping in the C Language Mapping chapter has been modified to achieve compatibility between the C and C++ mappings.

No Implementation Descriptions

This mapping does not contain implementation descriptions. It avoids details that would constrain implementations, but still allows clients to be fully source compatible with any compliant implementation. Some examples show possible implementations, but these are not required implementations.

20.1.2 Scoped Names

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

- OMG IDL modules are mapped to C++ namespaces.
- OMG IDL interfaces are mapped to C++ classes (as described in “Mapping for Interfaces” on page 20-6).
- All OMG IDL constructs scoped to an interface are accessed via C++ scoped names. For example, if a type **mode** were defined in interface **printer** then the type would be referred to as **`printer::mode`**.

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

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

is mapped into:

```
// C++
namespace M
{
    struct E {
        Long L;
    };
}
```

and **`E`** can be referred outside of **`M`** as **`M::E`**. Alternatively, a C++ **`using`** statement for namespace **`M`** can be used so that **`E`** can be referred to simply as **`E`**:

```
// C++
using namespace M;
```

```
E e;
e.L = 3;
```

Another alternative is to employ a **using** statement only for **M::E**:

```
// C++
using M::E;
E e;
e.L = 3;
```

To avoid C++ compilation problems, every use in OMG IDL of a C++ keyword as an identifier is mapped into the same name preceded by the prefix "_cxx_". For example, an IDL interface named "try" would be named "_cxx_try" when its name is mapped into C++. The complete list of C++ keywords from the 2 December 1996 Working Paper of the ANSI/ISO C++ standardization committees (X3J16, WG21) can be found in the "C++ Keywords" appendix.

20.1.3 C++ Type Size Requirements

The sizes of the C++ types used to represent OMG IDL types are implementation-dependent. That is, this mapping makes no requirements as to the **sizeof(T)** for anything except basic types (see "Mapping for Basic Data Types" on page 20-15) and string (see "Mapping for String Types" on page 20-17).

20.1.4 CORBA Module

The mapping relies on some predefined types, classes, and functions that are logically defined in a module named **CORBA.** The module is automatically accessible from a C++ compilation unit that includes a header file generated from an OMG IDL specification. In the examples presented in this document, CORBA definitions are referenced without explicit qualification for simplicity. In practice, fully scoped names or C++ **using** statements for the **CORBA** namespace would be required in the application source. Appendix A contains standard OMG IDL types.

20.2 Mapping for Modules

As shown in "Scoped Names" on page 20-4, a module defines a scope, and as such is mapped to a C++ **namespace** with the same name:

```
// IDL
module M
{
    // definitions
};

// C++
namespace M
{
```

```
        // definitions
}
```

Because namespaces were only recently added to the C++ language, few C++ compilers currently support them. Alternative mappings for OMG IDL modules that do not require C++ namespaces are in the Appendix “Alternative Mappings for C++ Dialects.”

20.3 Mapping for Interfaces

An interface is mapped to a C++ class that contains public definitions of the types, constants, operations, and exceptions defined in the interface.

A CORBA–C++–compliant program cannot

- Create or hold an instance of an interface class
- Use a pointer (**A***) or a reference (**A&**) to an interface class.

The reason for these restrictions is to allow a wide variety of implementations. For example, interface classes could not be implemented as abstract base classes if programs were allowed to create or hold instances of them. In a sense, the generated class is like a namespace that one cannot enter via a **using** statement. This example shows the behavior of the mapping of an interface:

```
// IDL
interface A
{
    struct S { short field; };
};

// C++
// Conformant uses
A::S s; // declare a struct variable
s.field = 3; // field access

// Non-conformant uses:
// one cannot declare an instance of an interface class...
A a;
// ...nor declare a pointer to an interface class...
A *p;
// ...nor declare a reference to an interface class.
void f(A &r);
```

20.3.1 Object Reference Types

The use of an interface type in OMG IDL denotes an object reference. Because of the different ways an object reference can be used and the different possible implementations in C++, an object reference maps to two C++ types. For an interface **A**, these types are named **A_var** and **A_ptr**. For historical reasons, the type **ARef** is defined as a synonym for **A_ptr**, but usage of the **Ref** names is not portable and

is thus deprecated. These types need not be distinct—**A_var** may be identical to **A_ptr**, for example—so a compliant program cannot overload operations using these types solely.

An operation can be performed on an object by using an arrow ("**->**") on a reference to the object. For example, if an interface defines an operation **op** with no parameters and **obj** is a reference to the interface type, then a call would be written **obj->op()**. The arrow operator is used to invoke operations on both the **_ptr** and **_var** object reference types.

Client code frequently will use the object reference variable type (**A_var**) because a variable will automatically release its object reference when it is deallocated or when assigned a new object reference. The pointer type (**A_ptr**) provides a more primitive object reference, which has similar semantics to a C++ pointer. Indeed, an implementation may choose to define **A_ptr** as **A***, but is not required to. Unlike C++ pointers, however, conversion to **void***, arithmetic operations, and relational operations, including test for equality, are all non-compliant. A compliant implementation need not detect these incorrect uses because requiring detection is not practical.

For many operations, mixing data of type **A_var** and **A_ptr** is possible without any explicit operations or casts. However, one needs to be careful in doing so because of the implicit release performed when the variable is deallocated. For example, the assignment statement in the code below will result in the object reference held by **p** to be released at the end of the block containing the declaration of **a**.

```
// C++
A_var a;
A_ptr p = // ...somehow obtain an objref...
a = p;
```

20.3.2 Widening Object References

OMG IDL interface inheritance does not require that the corresponding C++ classes are related, though that is certainly one possible implementation. However, if interface B inherits from interface A, the following implicit widening operations for B must be supported by a compliant implementation:

- **B_ptr** to **A_ptr**
- **B_ptr** to **Object_ptr**
- **B_var** to **A_ptr**
- **B_var** to **Object_ptr**

Implicit widening from a **B_var** to **A_var** or **Object_var** need not be supported; instead, widening between **_var** types for object references requires a call to **_duplicate** (described in "Object Reference Operations" on page 20-8).[1] An attempt to implicitly widen from one **_var** type to another must cause a compile-time error.[2] Assignment between two **_var** objects of the same type is supported, but widening assignments are not and must cause a compile-time error. Widening assignments may be done using **_duplicate**.

```
// C++
B_ptr bp = ...
A_ptr ap = bp;          // implicit widening
Object_ptr objp = bp;   // implicit widening
objp = ap;              // implicit widening

B_var bv = bp;          // bv assumes ownership of bp
ap = bv;                // implicit widening, bv retains
                        // ownership of bp
obp = bv;               // implicit widening, bv retains
                        // ownership of bp

A_var av = bv;          // illegal, compile-time error
A_var av = B::_duplicate(bv);// av, bv both refer to bp
B_var bv2 = bv;         // implicit _duplicate
A_var av2;
av2 = av;               // implicit _duplicate
```

20.3.3 Object Reference Operations

Conceptually, the **Object** class in the **CORBA** module is the base interface type for all CORBA objects. Any object reference can therefore be widened to the type **`Object_ptr`**. As with other interfaces, the CORBA namespace also defines the type **`Object_var`**.

CORBA defines three operations on any object reference: **duplicate**, **release**, and **is_nil**. Note that these are operations on the object reference, not the object implementation. Because the mapping does not require object references to themselves be C++ objects, the "**`->`**" syntax cannot be employed to express the usage of these operations. Also, for convenience these operations are allowed to be performed on a nil object reference.

The **release** and **is_nil** operations depend only on type **Object**, so they can be expressed as regular functions within the CORBA namespace as follows:

1.When **`T_ptr`** is mapped to **`T*`**, it is impossible in C++ to provide implicit widening between **`T_var`** types while also providing the necessary duplication semantics for **`T_ptr`** types.

2.This can be achieved by deriving all **`T_var`** types for object references from a base **`_var`** class, then making the assignment operator for **`_var`** private within each **`T_var`** type.

```
// C++
void release(Object_ptr obj);
Boolean is_nil(Object_ptr obj);
```

The **release** operation indicates that the caller will no longer access the reference so that associated resources may be deallocated. If the given object reference is nil, **release** does nothing. The **is_nil** operation returns **TRUE** if the object reference contains the special value for a nil object reference as defined by the ORB. Neither the **release** operation nor the **is_nil** operation may throw CORBA exceptions.

The **duplicate** operation returns a new object reference with the same static type as the given reference. The mapping for an interface therefore includes a static member function named **_duplicate** in the generated class. For example:

```
// IDL
interface A { };
```

```
// C++
class A
{
   public:
    static A_ptr _duplicate(A_ptr obj);
};
```

If the given object reference is nil, **_duplicate** will return a nil object reference. The **_duplicate** operation can throw CORBA system exceptions.

20.3.4 Narrowing Object References

The mapping for an interface defines a static member function named **_narrow** that returns a new object reference given an existing reference. Like **_duplicate**, the **_narrow** function returns a nil object reference if the given reference is nil. Unlike **_duplicate**, the parameter to **_narrow** is a reference of an object of any interface type (**Object_ptr**). If the actual (runtime) type of the parameter object can be widened to the requested interface's type, then **_narrow** will return a valid object reference. Otherwise, **_narrow** will return a nil object reference. For example, suppose A, B, C, and D are interface types, and D inherits from C, which inherits from B, which in turn inherits from A. If an object reference to a C object is widened to an **A_ptr** variable called **ap**, the

- **A::_narrow(ap)** returns a valid object reference;
- **B::_narrow(ap)** returns a valid object reference;
- **C::_narrow(ap)** returns a valid object reference;
- **D::_narrow(ap)** returns a nil object reference.

Narrowing to A, B, and C all succeed because the object supports all those interfaces. The **D::_narrow** returns a nil object reference because the object does not support the D interface.

For another example, suppose A, B, C, and D are interface types. C inherits from B, and both B and D inherit from A. Now suppose that an object of type C is passed to a function as an A. If the function calls **B::_narrow** or **C::_narrow**, a new object reference will be returned. A call to **D::_narrow** will fail and return nil.

If successful, the **_narrow** function creates a new object reference and does not consume the given object reference, so the caller is responsible for releasing both the original and new references.

The **_narrow** operation can throw CORBA system exceptions.

20.3.5 Nil Object Reference

The mapping for an interface defines a static member function named **_nil** that returns a nil object reference of that interface type. For each interface A, the following call is guaranteed to return **TRUE**:

```
// C++
Boolean true_result = is_nil(A::_nil());
```

A compliant application need not call **release** on the object reference returned from the **_nil** function.

As described in "Object Reference Types" on page 20-6, object references may not be compared using **operator==**, so **is_nil** is the only compliant way an object reference can be checked to see if it is nil.

The **_nil** function may not throw any CORBA exceptions.

A compliant program cannot attempt to invoke an operation through a nil object reference, since a valid C++ implementation of a nil object reference is a null pointer.

20.3.6 Object Reference Out Parameter

When a **_var** is passed as an **out** parameter, any previous value it refers to must be implicitly released. To give C++ mapping implementations enough hooks to meet this requirement, each object reference type results in the generation of an **_out** type which is used solely as the **out** parameter type. For example, interface **A** results in the object reference type **A_ptr**, the helper type **A_var**, and the **out** parameter type **A_out**. The general form for object reference **_out** types is shown below.

```
// C++
class A_out
{
  public:
A_out(A_ptr& p) : ptr_(p) { ptr_ = A::_nil(); }
A_out(A_var& p) : ptr_(p.ptr_) {
release(ptr_); ptr_ = A::_nil();
}
A_out(A_out& a) : ptr_(a.ptr_) {}
A_out& operator=(A_out& a) {
ptr_ = a.ptr_; return *this;
}
A_out& operator=(const A_var& a) {
ptr_ = A::_duplicate(A_ptr(a)); return *this;
}
A_out& operator=(A_ptr p) { ptr_ = p; return *this; }
operator A_ptr&() { return ptr_; }
A_ptr& ptr() { return ptr_; }
A_ptr operator->() { return ptr_; }

  private:
A_ptr& ptr_;
};
```

The first constructor binds the reference data member with the **A_ptr&** argument. The second constructor binds the reference data member with the **A_ptr** object reference held by the **A_var** argument, and then calls **release()** on the object reference. The third constructor, the copy constructor, binds the reference data member to the same **A_ptr** object reference bound to the data member of its argument. Assignment from another **A_out** copies the **A_ptr** referenced by the argument **A_out** to the data member. The overloaded assignment operator for **A_ptr** simply assigns the **A_ptr** object reference argument to the data member. The overloaded assignment operator for **A_var** duplicates the **A_ptr** held by the **A_var** before assigning it to the data member. Note that assignment does not cause any previously-held object reference value to be released; in this regard, the **A_out** type behaves exactly as an **A_ptr**. The **A_ptr&** conversion operator returns the data member. The **ptr()** member function, which can be used to avoid having to rely on implicit conversion, also returns the data member. The overloaded arrow operator (**operator->()**) returns the data member to allow operations to be invoked on the underlying object reference after it has been properly initialized by assignment.

20.3.7 Interface Mapping Example

The example below shows one possible mapping for an interface. Other mappings are also possible, but they must provide the same semantics and usage as this example.

```
// IDL
interface A
{
  A op(in A arg1, out A arg2);
};

// C++
class A;
typedef A *A_ptr;
class A : public virtual Object
{
  public:
static A_ptr _duplicate(A_ptr obj);
static A_ptr _narrow(Object_ptr obj);
static A_ptr _nil();

virtual A_ptr op(A_ptr arg1, A_out arg2) = 0;

  protected:
A();
virtual ~A();

  private:
A(const A&);
void operator=(const A&);
};

class A_var : public _var
{
  public:
A_var() : ptr_(A::_nil()) {}
A_var(A_ptr p) : ptr_(p) {}
A_var(const A_var &a) : ptr_(A::_duplicate(A_ptr(a))) {}
~A_var() { free(); }

A_var &operator=(A_ptr p) {
reset(p); return *this;
}
A_var &operator=(const A_var& a) {
if (this != &a) {
free();
ptr_ = A::_duplicate(A_ptr(a));
}
return *this;
}
A_ptr in() const { return ptr_; }
A_ptr& inout() { return ptr_; }
A_ptr& out() {
reset(A::_nil());
return ptr_;
```

```
}
A_ptr _retn() {
// yield ownership of managed object reference
A_ptr val = ptr_;
ptr_ = A::_nil();
return val;
}

operator const A_ptr&() const { return ptr_; }
operator A_ptr&() { return ptr_; }
A_ptr operator->() const { return ptr_; }

  protected:
A_ptr ptr_;
void free() { release(ptr_); }
void reset(A_ptr p) { free(); ptr_ = p; }

  private:
// hidden assignment operators for var types to
// fulfill the rules specified in
// Section 19.3.2
void operator=(const _var &);
};
```

The definition for the **A_out** type is the same as the one shown in “Object Reference Out Parameter” on page 20-10.

20.4 Mapping for Constants

OMG IDL constants are mapped directly to a C++ constant definition that may or may not define storage depending on the scope of the declaration. In the following example, a top-level IDL constant maps to a file-scope C++ constant whereas a nested constant maps to a class-scope C++ constant. This inconsistency occurs because C++ file-scope constants may not require storage (or the storage may be replicated in each compilation unit), while class-scope constants always take storage. As a side effect, this difference means that the generated C++ header file might not contain values for constants defined in the OMG IDL file.

```
// IDL
const string name = "testing";

interface A
{
    const float pi = 3.14159;
};

// C++
static const char *const name = "testing";

class A
{
  public:
static const Float pi;
};
```

In certain situations, use of a constant in OMG IDL must generate the constant's value instead of the constant's name.[3] For example,

```
// IDL
interface A
{
    const long n = 10;
    typedef long V[n];
};

// C++
class A
{
  public:
   static const long n;
   typedef long V[10];
};
```

Wide Character and Wide String Constants

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

3.A recent change made to the C++ language by the ANSI/ISO C++ standardization committees allows static integer constants to be initialized within the class declaration, so for some C++ compilers, the code generation issues described here may not be a problem.

```
static const CORBA::WChar *const ws = L"Hello World";
```

in C++.

20.5 Mapping for Basic Data Types

The basic data types have the mappings shown in Table 20-1[4]. Note that the mapping of the OMG IDL **boolean** type defines only the values 1 (TRUE) and 0 (FALSE); other values produce undefined behavior.

Table 20-1 Basic Data Type Mappings

OMG IDL	C++	C++ Out Type
short	CORBA::Short	CORBA::Short_out
long	CORBA::Long	CORBA::Long_out
long long	CORBA::LongLong	CORBA::LongLong_out
unsigned short	CORBA::UShort	CORBA::UShort_out
unsigned long	CORBA::ULong	CORBA::ULong_out
unsigned long long	CORBA::ULongLong	CORBA::ULongLong_out
float	CORBA::Float	CORBA::Float_out
double	CORBA::Double	CORBA::Double_out
long double	CORBA::LongDouble	CORBA::LongDouble_out
char	CORBA::Char	CORBA::Char_out
wchar	CORBA::WChar	CORBA::WChar_out
boolean	CORBA::Boolean	CORBA::Boolean_out
octet	CORBA::Octet	CORBA::Octet

Each OMG IDL basic type is mapped to a typedef in the CORBA module. This is because some types, such as **`short`** and **`long`**, may have different representations on different platforms, and the CORBA definitions will reflect the appropriate representation. For example, on a 64-bit machine where a long integer is 64 bits, the definition of **`CORBA::Long`** would still refer to a 32-bit integer. Requirements for the sizes of basic types are shown in “Basic Types” on page 3-23.

4. This mapping assumes that **`CORBA::LongLong`**, **`CORBA::ULongLong`**, and **`CORBA::LongDouble`** are mapped directly to native numeric C++ types (e.g., **`CORBA::LongLong`** to a 64-bit integer type) that support the required IDL semantics and that can be manipulated via built-in operators. If such native type support is not widely available, then an alternate mapping to C++ classes (that support appropriate creation, conversion, and manipulation operators) should also be provided by the C++ Mapping Revision Task Force.

Except for **boolean**, **char**, and **octet**, the mappings for basic types must be distinguishable from each other for the purposes of overloading. That is, one can safely write overloaded C++ functions on **Short**, **UShort**, **Long**, **ULong**, **Float**, and **Double**.

The **_out** types for the basic types are used to type **out** parameters within operation signatures, as described in "Operation Parameters and Signatures" on page 20-65. For the basic types, each **_out** type shall be a **typedef** to a reference to the corresponding C++ type. For example, the **Short_out** shall be defined in the **CORBA** namespace as follows:

```
// C++
typedef Short& Short_out;
```

The **_out** types for the basic types are provided for consistency with other **out** parameter types.

Programmers concerned with portability should use the CORBA types. However, some may feel that using these types with the CORBA qualification impairs readability. If the **CORBA** module is mapped to a namespace, a C++ **using** statement may help this problem. On platforms where the C++ data type is guaranteed to be identical to the OMG IDL data type, a compliant implementation may generate the native C++ type.

For the **Boolean** type, only the values 1 (representing **TRUE**) and 0 (representing **FALSE**) are defined; other values produce undefined behavior. Since many existing C++ software packages and libraries already provide their own preprocessor macro definitions of **TRUE** and **FALSE**, this mapping does not require that such definitions be provided by a compliant implementation. Requiring definitions for **TRUE** and **FALSE** could cause compilation problems for CORBA applications that make use of such packages and libraries. Instead, we recommend that compliant applications simply use the values 1 and 0 directly.[5] Alternatively, for those C++ compilers that support the **bool** type, the keywords **true** and **false** may be used.

20.6 Mapping for Enums

An OMG IDL **enum** maps directly to the corresponding C++ type definition. The only difference is that the generated C++ type may need an additional constant that is large enough to force the C++ compiler to use exactly 32 bits for values declared to be of the enumerated type.

5.Examples and descriptions in this document still use **TRUE** and **FALSE** for purposes of clarity.

```
// IDL
enum Color { red, green, blue };

// C++
enum Color { red, green, blue };
```

In addition, an **_out** type used to type **out** parameters within operation signatures is generated for each enumerated type. For enum **Color** shown above, the **Color_out** type shall be defined in the same scope as follows:

```
// C++
typedef Color& Color_out;
```

The **_out** types for enumerated types are generated for consistency with other **out** parameter types.

20.7 Mapping for String Types

As in the C mapping, the OMG IDL string type, whether bounded or unbounded, is mapped to **char*** in C++. String data is null-terminated. In addition, the **CORBA** module defines a class **String_var** that contains a **char*** value and automatically frees the pointer when a **String_var** object is deallocated. When a **String_var** is constructed or assigned from a **char***, the **char*** is consumed and thus the string data may no longer be accessed through it by the caller. Assignment or construction from a **const char*** or from another **String_var** causes a copy. The **String_var** class also provides operations to convert to and from **char*** values, as well as subscripting operations to access characters within the string. The full definition of the **String_var** interface is given in “String_var and String_out Class” on page 20-104. Because its mapping is **char***, the OMG IDL string type is the only non-basic type for which this mapping makes size requirements. For dynamic allocation of strings, compliant programs must use the following functions from the **CORBA** namespace:

```
// C++
namespace CORBA {
char *string_alloc(ULong len);
char *string_dup(const char*);
void string_free(char *);
...
}
```

The **string_alloc** function dynamically allocates a string, or returns a null pointer if it cannot perform the allocation. It allocates **len+1** characters so that the resulting string has enough space to hold a trailing NUL character. The **string_dup** function dynamically allocates enough space to hold a copy of its string argument, including the NUL character, copies its string argument into that memory, and returns a pointer to the new string. If allocation fails, a null pointer is returned. The **string_free** function deallocates a string that was allocated with **string_alloc** or **string_dup**. Passing a null pointer to **string_free** is acceptable and results in no action being performed. These functions allow ORB

implementations to use special memory management mechanisms for strings if necessary, without forcing them to replace global **`operator new`** and **`operator new[]`**.

The **`string_alloc`**, **`string_dup`**, and **`string_free`** functions may not throw CORBA exceptions.

Note that a static array of char in C++ decays to a **`char*`**[6], so care must be taken when assigning one to a **`String_var`**, since the **`String_var`** will assume the pointer points to data allocated via **`string_alloc`** and thus will eventually attempt to **`string_free`** it:

```
// C++
// The following is an error, since the char* should point to
// data allocated via string_alloc so it can be consumed
String_var s = "static string";// error

// The following are OK, since const char* are copied,
// not consumed
const char* sp = "static string";
s = sp;
s = (const char*)"static string too";
```


```
// C++
// The following is an error, since the char* should point to
// data allocated via string_alloc so it can be consumed
String_var s = "static string";// error

// The following are OK, since const char* are copied,
// not consumed
const char* sp = "static string";
s = sp;
s = (const char*)"static string too";
```


6. This has changed in ANSI/ISO C++, where string literals are const char*, not char*. However, since most C++ compilers do not yet implement this change, portable programs must heed the advice given here.

When a **String_var** is passed as an **out** parameter, any previous value it refers to must be implicitly freed. To give C++ mapping implementations enough hooks to meet this requirement, the string type also results in the generation of a **String_out** type in the **CORBA** namespace which is used solely as the string **out** parameter type. The general form for the **String_out** type is shown below.

```
// C++
class String_out
{
  public:
String_out(char*& p) : ptr_(p) { ptr_ = 0; }
String_out(String_var& p) : ptr_(p.ptr_) {
string_free(ptr_); ptr_ = 0;
}
String_out(String_out& s) : ptr_(s.ptr_) {}
String_out& operator=(String_out& s) {
ptr_ = s.ptr_; return *this;
}
String_out& operator=(char* p) {
ptr_ = p; return *this;
}
String_out& operator=(const char* p) {
ptr_ = string_dup(p); return *this;
}
operator char*&() { return ptr_; }
char*& ptr() { return ptr_; }

 private:
char*& ptr_;

// assignment from String_var disallowed
void operator=(const String_var&);
};
```

The first constructor binds the reference data member with the **char*&** argument. The second constructor binds the reference data member with the **char*** held by the **String_var** argument, and then calls **string_free()** on the string. The third constructor, the copy constructor, binds the reference data member to the same **char*** bound to the data member of its argument. Assignment from another **String_out** copies the **char*** referenced by the argument **String_out** to the **char*** referenced by the data member. The overloaded assignment operator for **char*** simply assigns the **char*** argument to the data member. The overloaded assignment operator for **const char*** duplicates the argument and assigns the result to the data member. Note that assignment does not cause any previously-held string to be freed; in this regard, the **String_out** type behaves exactly as a **char***. The **char*&** conversion operator returns the data member. The **ptr()** member function, which can be used to avoid having to rely on implicit conversion, also returns the data member.

Assignment from **String_var** to a **String_out** is disallowed because of the memory management ambiguities involved. Specifically, it is not possible to determine whether the string owned by the **String_var** should be taken over by the **String_out** without copying, or if it should be copied. Disallowing assignment from **String_var** forces the application developer to make the choice explicitly:

```
// C++
void
A::op(String_out arg)
{
String_var s = string_dup("some string");
...
arg = s;// disallowed; either
arg = string_dup(s);// 1: copy, or
arg = s._retn();// 2: adopt
}
```

On the line marked with the comment "1," the application writer is explicitly copying the string held by the **String_var** and assigning the result to the **arg** argument. Alternatively, the application writer could use the technique shown on the line marked with the comment "2" in order to force the **String_var** to give up its ownership of the string it holds so that it may be returned in the **arg** argument without incurring memory management errors.

20.8 Mapping for Wide String Types

Both bounded and unbounded wide string types are mapped to **CORBA::WChar*** in C++. In addition, the **CORBA** module defines **WString_var** and **WString_out** classes. Each of these classes provides the same member functions with the same semantics as their **string** counterparts, except of course they deal with wide strings and wide characters.

Dynamic allocation and deallocation of wide strings must be performed via the following functions:

```
// C++
namespace CORBA {
   // ...
   WChar *wstring_alloc(ULong len);
   WChar *wstring_dup(const WChar* ws);
   void wstring_free(WChar*);
};
```

These functions have the same semantics as the same functions for the **string** type, except they operate on wide strings.

20.9 Mapping for Structured Types

The mapping for **struct**, **union**, and **sequence** (but not **array**) is a C++ struct or class with a default constructor, a copy constructor, an assignment operator, and a destructor. The default constructor initializes object reference members to appropriately-typed nil object references and string members to NULL; all other members are initialized via their default constructors. The copy constructor performs a deep-copy from the existing structure to create a new structure, including calling **`_duplicate`** on all object reference members and performing the necessary heap allocations for all string members. The assignment operator first releases all object reference members and frees all string members, and then performs a deep-copy to create a new structure. The destructor releases all object reference members and frees all string members.

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
- 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.

As a convenience for managing pointers to variable-length data types, the mapping also provides a managing helper class for each variable-length type. This type, which is named by adding the suffix "_var" to the original type's name, automatically deletes the pointer when an instance is destroyed. An object of type **`T_var`** behaves similarly to the structured type **`T`**, except that members must be accessed indirectly. For a struct, this means using an arrow ("**`->`**") instead of a dot ("**`.`**").

```
// IDL
struct S { string name; float age; };
void f(out S p);

// C++
S a;
S_var b;
f(b);
a = b; // deep-copy
cout << "names " << a.name << ", " << b->name << endl;
```

20.9.1 T_var Types

The general form of the **T_var** types is shown below.

```
// C++
class T_var
{
  public:
T_var();
T_var(T *);
T_var(const T_var &);
~T_var();

T_var &operator=(T *);
T_var &operator=(const T_var &);

T* operator->();
const T* operator->() const;

/* in parameter type */ in() const;
/* inout parameter type */ inout();
/* out parameter type */ out();
/* return type */ _retn();

// other conversion operators to support
// parameter passing
};
```

The default constructor creates a **T_var** containing a null **T***. Compliant applications may not attempt to convert a **T_var** created with the default constructor into a **T*** nor use its overloaded **operator->** without first assigning to it a valid **T*** or another valid **T_var**. Due to the difficulty of doing so, compliant implementations are not required to detect this error. Conversion of a null **T_var** to a **T_out** is allowed, however, so that a **T_var** can legally be passed as an **out** parameter. Conversion of a null **T_var** to a **T*&** is also allowed so as to be compatible with earlier versions of this specification.

The **T*** constructor creates a **T_var** that, when destroyed, will **delete** the storage pointed to by the **T*** parameter. The parameter to this constructor should never be a null pointer. Compliant implementations are not required to detect null pointers passed to this constructor.

The copy constructor deep-copies any data pointed to by the **T_var** constructor parameter. This copy will be destroyed when the **T_var** is destroyed or when a new value is assigned to it. Compliant implementations may, but are not required to, utilize some form of reference counting to avoid such copies.

The destructor uses **delete** to deallocate any data pointed to by the **T_var**, except for strings and array types, which are deallocated using the **string_free** and **T_free** (for array type **T**) deallocation functions, respectively.

The **T*** assignment operator results in the deallocation of any old data pointed to by the **T_var** before assuming ownership of the **T*** parameter.

The normal assignment operator deep-copies any data pointed to by the **T_var** assignment parameter. This copy will be destroyed when the **T_var** is destroyed or when a new value is assigned to it.

The overloaded **operator->** returns the **T*** held by the **T_var**, but retains ownership of it. Compliant applications may not call this function unless the **T_var** has been initialized with a valid non-null **T*** or **T_var**.

In addition to the member functions described above, the **T_var** types must support conversion functions that allow them to fully support the parameter passing modes shown in “Basic Argument and Result Passing” on page 20-66. The form of these conversion functions is not specified so as to allow different implementations, but the conversions must be automatic (i.e., they must require no explicit application code to invoke them).

Because implicit conversions can sometimes cause problems with some C++ compilers and with code readability, the **T_var** types also support member functions that allow them to be explicitly converted for purposes of parameter passing. To pass a **T_var** as an **in** parameter, an application can call the **in()** member function of the **T_var**; for **inout** parameters, the **inout()** member function; for **out** parameters, the **out()** member function; and to obtain a return value from the **T_var**, the **_retn()** function.[7] For each **T_var** type, the return types of each of these functions shall match the types shown in Table 6 on page 19-59 for the **in**, **inout**, **out**, and return modes for underlying type **T** respectively.

For **T_var** types that return **T*&** from the **out()** member function, the **out()** member function calls **delete** on the **T*** owned by the **T_var**, sets it equal to the null pointer, and then returns a reference to it. This is to allow for proper management

7.A leading underscore is needed on the **_retn()** function to keep it from clashing with user-defined member names of constructed types, but leading underscores are not needed for the **in()**, **inout()**, and **out()** functions because their names are IDL keywords, so users can’t define members with those names.

of the **T*** owned by a **T_var** when passed as an **out** parameter, as described in "Mapping For Operations and Attributes" on page 20-61. An example implementation of such an **out()** function is shown below:

```
// C++
T*& T_var::out()
{
// assume ptr_ is the T* data member of the T_var
delete ptr_;
ptr_ = 0;
return ptr_;
}
```

Similarly, for **T_var** types whose corresponding type **T** is returned from IDL operations as **T*** (see Table 20-2 on page 20-66), the **_retn()** function stores the value of the **T*** owned by the **T_var** into a temporary pointer, sets the **T*** to the null pointer value, and then returns the temporary. The **T_var** thus yields ownership of its **T*** to the caller of **_retn()** without calling **delete** on it, and the caller becomes responsible for eventually deleting the returned **T***. An example implementation of such a **_retn()** function is shown below:

```
// C++
T* T_var::_retn()
{
// assume ptr_ is the T* data member of the T_var
T* tmp = ptr_;
ptr_ = 0;
return tmp;
}
```

This allows, for example, a method implementation to store a **T*** as a potential return value in a **T_var** so that it will be deleted if an exception is thrown, and yet be able to acquire control of the **T*** to be able to return it properly:

```
// C++
T_var t = new T;// t owns pointer to T
if (exceptional_condition) {
// t owns the pointer and will delete it
// as the stack is unwound due to throw
throw AnException();
}
...
return t._retn();// _retn() takes ownership of
// pointer from t
```

The **T_var** types are also produced for fixed-length structured types for reasons of consistency. These types have the same semantics as **T_var** types for variable-length types. This allows applications to be coded in terms of **T_var** types regardless of whether the underlying types are fixed- or variable-length.

Each **T_var** type must be defined at the same level of nesting as its **T** type.

T_var types do not work with a pointer to constant **T**, since they provide no constructor nor **operator=** taking a **const T*** parameter. Since C++ does not allow **delete** to be called on a **const T***[8], the **T_var** object would normally have to copy the const object; instead, the absence of the **const T*** constructor and assignment operators will result in a compile-time error if such an initialization or assignment is attempted. This allows the application developer to decide if a copy is really wanted or not. Explicit copying of **const T*** objects into **T_var** types can be achieved via the copy constructor for **T**:

```
// C++
const T *t = ...;
T_var tv = new T(*t);
```

20.9.2 T_out Types

When a **T_var** is passed as an **out** parameter, any previous value it referred to must be implicitly deleted. To give C++ mapping implementations enough hooks to meet this requirement, each **T_var** type has a corresponding **T_out** type which is used solely as the **out** parameter type. The general form for **T_out** types for variable-length types is shown below.

```
// C++

class T_out
{
  public:
T_out(T*& p) : ptr_(p) { ptr_ = 0; }
T_out(T_var& p) : ptr_(p.ptr_) {
delete ptr_;
ptr_ = 0;
}
T_out(T_out& p) : ptr_(p.ptr_) {}
T_out& operator=(T_out& p) {
ptr_ = p.ptr_;
return *this;
}
```

8. This too has changed in ANSI/ISO C++, but it not yet widely implemented by C++ compilers.

```
T_out& operator=(T* p) { ptr_ = p; return *this; }

operator T*&() { return ptr_; }
T*& ptr() { return ptr_; }

T* operator->() { return ptr_; }

  private:
T*& ptr_;

// assignment from T_var not allowed
void operator=(const T_var&):
};
```

The first constructor binds the reference data member with the **T*&** argument and sets the pointer to the null pointer value. The second constructor binds the reference data member with the pointer held by the **T_var** argument, and then calls **delete** on the pointer (or **string_free()** in the case of the **String_out** type or **T_free()** in the case of a **T_var** for an array type **T**). The third constructor, the copy constructor, binds the reference data member to the same pointer referenced by the data member of the constructor argument. Assignment from another **T_out** copies the **T*** referenced by the **T_out** argument to the data member. The overloaded assignment operator for **T*** simply assigns the pointer argument to the data member. Note that assignment does not cause any previously-held pointer to be deleted; in this regard, the **T_out** type behaves exactly as a **T***. The **T*&** conversion operator returns the data member. The **ptr()** member function, which can be used to avoid having to rely on implicit conversion, also returns the data member. The overloaded arrow operator (**operator->()**) allows access to members of the data structure pointed to by the **T*** data member. Compliant applications may not call the overloaded **operator->()** unless the **T_out** has been initialized with a valid non-null **T***.

Assignment to a **T_out** from instances of the corresponding **T_var** type is disallowed because there is no way to determine whether the application developer wants a copy to be performed, or whether the **T_var** should yield ownership of its managed pointer so it can be assigned to the **T_out**. To perform a copy of a **T_var** to a **T_out**, the application should use **new**:

```
// C++
T_var t = ...;
my_out = new T(t.in());// heap-allocate a copy
```

The **in()** function called on **t** typically returns a **const T&**, suitable for invoking the copy constructor of the newly-allocated **T** instance.

Alternatively, to make the **T_var** yield ownership of its managed pointer so it can be returned in a **T_out** parameter, the application should use the **T_var::_retn()** function:

```
// C++
T_var t = ...;
my_out = t._retn();// t yields ownership, no copy
```

Note that the **T_out** types are not intended to serve as general-purpose data types to be created and destroyed by applications; they are used only as types within operation signatures to allow necessary memory management side-effects to occur properly.

20.10 Mapping for Struct Types

An OMG IDL struct maps to C++ struct, with each OMG IDL struct member mapped to a corresponding member of the C++ struct. This mapping allows simple field access as well as aggregate initialization of most fixed-length structs. To facilitate such initialization, C++ structs must not have user-defined constructors, assignment operators, or destructors, and each struct member must be of self-managed type. With the exception of strings and object references, the type of a C++ struct member is the normal mapping of the OMG IDL member's type.

For a string or object reference member, the name of the C++ member's type is not specified by the mapping—a compliant program therefore cannot create an object of that type. The behavior[9] of the type is the same as the normal mapping (**char*** for string, **A_ptr** for an interface A) except the type's copy constructor copies the member's storage and its assignment operator releases the member's old storage. These types must also provide the **in()**, **inout()**, **out()**, and **_retn()** functions that their corresponding **T_var** types provide to allow them to support the parameter passing modes specified in "Basic Argument and Result Passing" on page 20-66.

Assignment between a string or object reference member and a corresponding **T_var** type (**String_var** or **A_var**) always results in copying the data, while assignment with a pointer does not. The one exception to the rule for assignment is when a **const char*** is assigned to a member, in which case the storage is copied.

When the old storage must not be freed (for example, it is part of the function's activation record), one can access the member directly as a pointer using the **_ptr** field accessor. This usage is dangerous and generally should be avoided.

```
// IDL
struct FixedLen { float x, y, z; };

// C++
FixedLen x1 = {1.2, 2.4, 3.6};
```

9.Those implementations concerned with data layout compatibility with the C mapping in this manual will also want to ensure that the sizes of these members match those of their C mapping counterparts.

```
FixedLen_var x2 = new FixedLen;
x2->y = x1.z;
```

The example above shows usage of the **T** and **T_var** types for a fixed-length struct. When it goes out of scope, **x2** will automatically free the heap-allocated **FixedLen** object using **delete**.

The following examples illustrate mixed usage of **T** and **T_var** types for variable-length types, using the following OMG IDL definition:

```
// IDL
interface A;
struct Variable { string name; };

// C++
Variable str1;// str1.name is initially NULL
Variable_var str2 = new Variable;// str2->name is
      // initially NULL
char *non_const;
const char *const2;
String_var string_var;
const char *const3 = "string 1";
const char *const4 = "string 2";

str1.name = const3;// 1: free old storage, copy
str2->name = const4;// 2: free old storage, copy
```

In the example above, the **name** components of variables **str1** and **str2** both start out as null. On the line marked 1, **const3** is assigned to the **name** component of **str1**; this results in the previous **str1.name** being freed, and since **const3** points to const data, the contents of **const3** being copied. In this case, **str1.name** started out as null, so no previous data needs to be freed before the copying of **const3** takes place. Line 2 is similar to line 1, except that **str2** is a **T_var** type.

Continuing with the example:

```
// C++
non_const = str1.name;// 3: no free, no copy
const2 = str2->name;// 4: no free, no copy
```

On the line marked 3, **str1.name** is assigned to **non_const**. Since **non_const** is a pointer type (**char***), **str1.name** is not freed, nor are the data it points to copied. After the assignment, **str1.name** and **non_const** effectively point to the same storage, with **str1.name** retaining ownership of that storage. Line 4 is identical to line 3, even though **const2** is a pointer to const char; **str2->name** is neither freed nor copied because **const2** is a pointer type.

```
// C++
str1.name = non_const;// 5: free, no copy
str1.name = const2;// 6: free old storage, copy
```

Line 5 involves assignment of a **char*** to **str1.name**, which results in the old **str1.name** being freed and the value of the **non_const** pointer, but not the data it points to, being copied. In other words, after the assignment **str1.name** points to the same storage as **non_const** points to. Line 6 is the same as line 5 except that because **const2** is a **const char***, the data it points to are copied.

```
// C++
str2->name = str1.name;// 7: free old storage, copy
str1.name = string_var;// 8: free old storage, copy
string_var = str2->name;// 9: free old storage, copy
```

On line 7, assignment is performed to a member from another member, so the original value is of the left-hand member is freed and the new value is copied. Similarly, lines 8 and 9 involve assignment to or from a **String_var**, so in both cases the original value of the left-hand side is freed and the new value is copied.

```
// C++
str1.name._ptr = str2.name;// 10: no free, no copy
```

Finally, line 10 uses the **_ptr** field accessor, so no freeing or copying takes place. Such usage is dangerous and generally should be avoided.

ORB implementations concerned with single-process interoperability with the C mapping may overload **operator new()** and **operator delete()** for structs so that dynamic allocation uses the same mechanism as the C language dynamic allocation functions. Whether these operators are overloaded by the implementation or not, compliant programs use **new** to dynamically allocate structs and **delete** to free them.

20.11 Mapping for Fixed

The C++ mapping for **fixed** is an abstract data type, with the following class and function templates:

```
// C++ class template
template<CORBA::UShort d, Short s>
class Fixed
{
 public:

   // Constructors...
   Fixed(int val = 0);
   Fixed(CORBA::LongDouble val);
   Fixed(const Fixed<d,s>& val);
   ~Fixed();

   // Conversions...
```

```
    operator LongDouble() const;

    // Operators...
    Fixed<d,s>& operator=(const Fixed<d,s>& val);
    Fixed<d,s>& operator++();
    Fixed<d,s>& operator++(int);
    Fixed<d,s>& operator--();
    Fixed<d,s>& operator--(int);
    Fixed<d,s>& operator+() const;
    Fixed<d,s>& operator-() const;
    int operator!() const;

    // Other member functions
    CORBA::UShort fixed_digits() const;
    CORBA::Short fixed_scale() const;
};

template<CORBA::UShort d, CORBA::Short s>
istream& operator>>(istream& is, Fixed<d,s> &val);
template<CORBA::UShort d, CORBA::Short s>
ostream& operator<<(ostream& os, const Fixed<d,s> &val);
```

The digits and scale, d_r and s_r, respectively, in the results of the binary arithmetic functions (**+**, **-**, ***** and **/**) are computed according to the rules in "Semantics" on page 3-20. One way to do this is to declare the result types with a macro that evaluates to the appropriate values, based on the digits and scale of the operands:

```
// Example of Fixed result type declaration
// Fixed<_FIXED_ADD_TYPE(d1,s1,d2,s2)> => Fixed<d_r,s_r>
```

The template specification below should be read as a prefix to each of the **operator** function declarations following.

```
// C++ function templates for operators...
template<unsigned short d1, short s1, unsigned short d2,
    short s2)
Fixed<d_r,s_r> operator + (const Fixed<d1,s1> &val1,
    const Fixed<d2,s2> &val2);
Fixed<d_r,s_r> operator - (const Fixed<d1,s1> &val1,
    const Fixed<d2,s2> &val2);
Fixed<d_r,s_r> operator * (const Fixed<d1,s1> &val1,
    const Fixed<d2,s2> &val2);
Fixed<d_r,s_r> operator / (const Fixed<d1,s1> &val1,
    const Fixed<d2,s2> &val2);
Fixed<d1,s1> operator += (const Fixed<d1,s1> &val1,
    const Fixed<d2,s2> &val2);
Fixed<d1,s1> operator -= (const Fixed<d1,s1> &val1,
    const Fixed<d2,s2> &val2);
Fixed<d1,s1> operator *= (const Fixed<d1,s1> &val1,
    const Fixed<d2,s2> &val2);
```

```
Fixed<d1,s1> operator /= (const Fixed<d1,s1> &val1,
   const Fixed<d2,s2> &val2);
int operator > (const Fixed<d1,s1> &val1,
   const Fixed<d2,s2> &val2);
int operator < (const Fixed<d1,s1> &val1,
   const Fixed<d2,s2> &val2);
int operator >= (const Fixed<d1,s1> &val1,
   const Fixed<d2,s2> &val2);
int operator <= (const Fixed<d1,s1> &val1,
   const Fixed<d2,s2> &val2);
int operator == (const Fixed<d1,s1> &val1,
   const Fixed<d2,s2> &val2);
int operator != (const Fixed<d1,s1> &val1,
   const Fixed<d2,s2> &val2);
```

20.11.1 Fixed T_var and T_out Types

Just as for other types, **`T_var`** types are defined for **`Fixed`** types. The semantics of the **`T_var`** types for **`Fixed`** types is similar to that for fixed-length structs.

A **`T_out`** type for a **`Fixed`** type is defined as typedef to a reference to the **`Fixed`** type, with the digits and scale added to the name to disambiguate it. For example, the name of the **`T_out`** type for the type **`Fixed<5,2>`** is **`Fixed_5_2_out`**[10]:

```
// C++
typedef Fixed<5, 2>& Fixed_5_2_out;
```

20.12 Mapping for Union Types

Unions map to C++ classes with access functions for the union members and discriminant. The default union constructor performs no application-visible initialization of the union. It does not initialize the discriminator, nor does it initialize any union members to a state useful to an application. (The implementation of the default constructor can do whatever type of initialization it wants to, but such initialization is implementation-dependent. No compliant application can count on a union ever being properly initialized by the default constructor alone.)

It is therefore an error for an application to access the union before setting it, but ORB implementations are not required to detect this error due to the difficulty of doing so. The copy constructor and assignment operator both perform a deep-copy of their parameters, with the assignment operator releasing old storage if necessary. The destructor releases all storage owned by the union.

10. Note that this naming scheme would not be necessary if fixed types, like sequences and arrays, were not allowed to be passed as anonymous types.

The union discriminant access functions have the name **_d** to both be brief and avoid name conflicts with the members. The **_d** discriminator modifier function can only be used to set the discriminant to a value within the same union member. In addition to the **_d** accessors, a union with an implicit default member provides a **_default()** member function that sets the discriminant to a legal default value. A union has an implicit default member if it does not have a default case and not all permissible values of the union discriminant are listed.

Setting the union value through an access function automatically sets the discriminant and may release the storage associated with the previous value. Attempting to get a value through an access function that does not match the current discriminant results in undefined behavior. If an access function for a union member with multiple legal discriminant values is used to set the value of the discriminant, the union implementation is free to set the discriminant to any one of the legal values for that member. The actual discriminant value chosen under these circumstances is implementation dependent.

The following example helps illustrate the mapping for union types:

```
// IDL
typedef octet Bytes[64];
struct S { long len; };
interface A;
union U switch (long) {
    case 1: long x;
    case 2: Bytes y;
    case 3: string z;
    case 4:
    case 5: S w;
    default: A obj;
};


// C++
typedef Octet Bytes[64];
typedef Octet Bytes_slice;
class Bytes_forany { ... };
struct S { Long len; };
typedef ... A_ptr;
class U
{
```

```
  public:
U();
U(const U&);
~U();
U &operator=(const U&);

void _d(Long);
Long _d() const;

void x(Long);
Long x() const;

void y(Bytes);
Bytes_slice *y() const;

void z(char*);// free old storage, no copy
void z(const char*);// free old storage, copy
void z(const String_var &);// free old storage, copy
const char *z() const;

void w(const S &);// deep copy
const S &w() const;// read-only access
S &w();// read-write access

void obj(A_ptr);// release old objref,
   // duplicate
A_ptr obj() const;// no duplicate
};
```

Accessor and modifier functions for union members provide semantics similar to that of struct data members. Modifier functions perform the equivalent of a deep-copy of their parameters, and their parameters should be passed by value (for small types) or by reference to const (for larger types). Accessors that return a reference to a non-const object can be used for read-write access, but such accessors are only provided for the following types: **struct**, **union**, **sequence**, and **any**.

For an array union member, the accessor returns a pointer to the array slice, where the slice is an array with all dimensions of the original except the first (array slices are described in detail in “Mapping For Array Types” on page 20-41). The array slice return type allows for read-write access for array members via regular subscript operators. For members of an anonymous array type, supporting typedefs for the array must be generated directly into the union. For example:

```
// IDL
union U switch (long) {
  default: long array[20][20];
};

// C++
class U
{
  public:
// ...
void array(long arg[20][20]);
typedef long _array_slice[20];
_array_slice * array();
// ...
};
```

The name of the supporting array slice typedef is created by prepending an underscore and appending "_slice" to the union member name. In the example above, the array member named "array" results in an array slice typedef called "_array_slice" nested in the union class.

For string union members, the **`char*`** modifier results in the freeing of old storage before ownership of the pointer parameter is assumed, while the **`const char*`** modifier and the **`String_var`** modifier[11] both result in the freeing of old storage before the parameter's storage is copied. The accessor for a string member returns a **`const char*`** to allow examination but not modification of the string storage.[12]

For object reference union members, object reference parameters to modifier functions are duplicated after the old object reference is released. An object reference return value from an accessor function is not duplicated because the union retains ownership of the object reference.

The restrictions for using the **`_d`** discriminator modifier function are shown by the following examples, based on the definition of the union **`U`** shown above:

---

11. A separate modifier for **`String_var`** is needed because it can automatically convert to both a **`char*`** and a **`const char*`**; since unions provide modifiers for both of these types, an attempt to set a string member of a union from a **`String_var`** would otherwise result in an ambiguity error at compile time.

12. A return type of **`char*`** allowing read-write access could mistakenly be assigned to a **`String_var`**, resulting in the **`String_var`** and the union both assuming ownership for the string's storage.

```
// C++
S s = {10};
U u;
u.w(s);// member w selected
u._d(4);// OK, member w selected
u._d(5);// OK, member w selected
u._d(1);// error, different member selected
A_ptr a = ...;
u.obj(a);// member obj selected
u._d(7);// OK, member obj selected
u._d(1);// error, different member selected
```

As shown here, the **_d** modifier function cannot be used to implicitly switch between different union members. The following shows an example of how the **_default()** member function is used:

```
// IDL
union Z switch(boolean) {
    case TRUE: short s;
};

// C++
Z z;
z._default(); // implicit default member selected
Boolean disc = z._d();// disc == FALSE
U u;// union U from previous example
u._default();// error, no _default() provided
```

For union **Z**, calling the **_default()** member function causes the union's value to be composed solely of the discriminator value of **FALSE**, since there is no explicit default member. For union **U**, calling **_default()** causes a compilation error because **U** has an explicitly declared default case and thus no **_default()** member function. A **_default()** member function is only generated for unions with implicit default members.

ORB implementations concerned with single-process interoperability with the C mapping may overload **operator new()** and **operator delete()** for unions so that dynamic allocation uses the same mechanism as the C language dynamic allocation functions. Whether these operators are overloaded by the implementation or not, compliant programs use **new** to dynamically allocate unions and **delete** to free them.

20.13 Mapping for Sequence Types

A sequence is mapped to a C++ class that behaves like an array with a current length and a maximum length. For a bounded sequence, the maximum length is implicit in the sequence's type and cannot be explicitly controlled by the programmer. For an

unbounded sequence, the initial value of the maximum length can be specified in the sequence constructor to allow control over the size of the initial buffer allocation. The programmer may always explicitly modify the current length of any sequence.

For an unbounded sequence, setting the length to a larger value than the current length may reallocate the sequence data. Reallocation is conceptually equivalent to creating a new sequence of the desired new length, copying the old sequence elements *zero through length-1* into the new sequence, and then assigning the old sequence to be the same as the new sequence. Setting the length to a smaller value than the current length does not affect how the storage associated with the sequence is manipulated. Note, however, that the elements orphaned by this reduction are no longer accessible and that their values cannot be recovered by increasing the sequence length to its original value.

For a bounded sequence, attempting to set the current length to a value larger than the maximum length given in the OMG IDL specification produces undefined behavior.

For each different named OMG IDL sequence type, a compliant implementation provides a separate C++ sequence type. For example:

```
// IDL
typedef sequence<long> LongSeq;
typedef sequence<LongSeq, 3> LongSeqSeq;


// C++
class LongSeq// unbounded sequence
{
  public:
LongSeq();// default constructor
LongSeq(ULong max);// maximum constructor
LongSeq(// T *data constructor
ULong max,
ULong length,
Long *value,
Boolean release = FALSE
);
LongSeq(const LongSeq&);
~LongSeq();
...
};
```

```
class LongSeqSeq// bounded sequence
{
  public:
LongSeqSeq();// default constructor
LongSeqSeq(// T *data constructor
ULong length,
LongSeq *value,
Boolean release = FALSE
);
LongSeqSeq(const LongSeqSeq&);
~LongSeqSeq();
...
};
```

For both bounded and unbounded sequences, the default constructor (as shown in the example above) sets the sequence length equal to 0. For bounded sequences, the maximum length is part of the type and cannot be set or modified, while for unbounded sequences, the default constructor also sets the maximum length to 0. The default constructor for a bounded sequence always allocates a contents vector, so it always sets the **`release`** flag to **`TRUE`**.

Unbounded sequences provide a constructor that allows only the initial value of the maximum length to be set (the “maximum constructor” shown in the example above). This allows applications to control how much buffer space is initially allocated by the sequence. This constructor also sets the length to 0 and the **`release`** flag to **`TRUE`**.

The “**`T *data`**” constructor (as shown in the example above) allows the length and contents of a bounded or unbounded sequence to be set. For unbounded sequences, it also allows the initial value of the maximum length to be set. For this constructor, ownership of the contents vector is determined by the **`release`** parameter—**`FALSE`** means the caller owns the storage, while **`TRUE`** means that the sequence assumes ownership of the storage. If **`release`** is **`TRUE`**, the contents vector must have been allocated using the sequence **`allocbuf`** function, and the sequence will pass it to **`freebuf`** when finished with it. The **`allocbuf`** and **`freebuf`** functions are described on “Additional Memory Management Functions” on page 20-40.

The copy constructor creates a new sequence with the same maximum and length as the given sequence, copies each of its current elements (items *zero* through *length–1*), and sets the **`release`** flag to **`TRUE`**.

The assignment operator deep-copies its parameter, releasing old storage if necessary. It behaves as if the original sequence is destroyed via its destructor and then the source sequence copied using the copy constructor.

If **`release=TRUE`**, the destructor destroys each of the current elements (items *zero* through *length–1*).

For an unbounded sequence, if a reallocation is necessary due to a change in the length and the sequence was created using the **`release=TRUE`** parameter in its constructor, the sequence will deallocate the old storage. If **`release`** is **`FALSE`** under these circumstances, old storage will not be freed before the reallocation is performed. After reallocation, the **`release`** flag is always set to **`TRUE`**.

For an unbounded sequence, the **maximum()** accessor function returns the total amount of buffer space currently available. This allows applications to know how many items they can insert into an unbounded sequence without causing a reallocation to occur. For a bounded sequence, **maximum()** always returns the bound of the sequence as given in its OMG IDL type declaration.

The overloaded subscript operators (**operator[]**) return the item at the given index. The non-const version must return something that can serve as an lvalue (i.e., something that allows assignment into the item at the given index), while the const version must allow read-only access to the item at the given index.

The overloaded subscript operators may not be used to access or modify any element beyond the current sequence length. Before either form of **operator[]** is used on a sequence, the length of the sequence must first be set using the **length(ULong)** modifier function, unless the sequence was constructed using the **T *data** constructor.

For strings and object references, **operator[]** for a sequence must return a type with the same semantics as the types used for string and object reference members of structs and arrays, so that assignment to the string or object reference sequence member via **operator=()** will release old storage when appropriate. Note that whatever these special return types are, they must honor the setting of the **release** parameter in the **T *data** constructor with respect to releasing old storage.

For the **T *data** sequence constructor, the type of **T** for strings and object references is **char*** and **T_ptr**, respectively. In other words, string buffers are passed as **char**** and object reference buffers are passed as **T_ptr***.

20.13.1 Sequence Example

The example below shows full declarations for both a bounded and an unbounded sequence.

```
// IDL
typedef sequence<T> V1;      // unbounded sequence
typedef sequence<T, 2> V2;   // bounded sequence
```

```
// C++
class V1// unbounded sequence
{
  public:
V1();
V1(ULong max);
V1(ULong max, ULong length, T *data,
Boolean release = FALSE);
V1(const V1&);
~V1();
V1 &operator=(const V1&);
```

```
ULong maximum() const;

void length(ULong);
ULong length() const;

T &operator[](ULong index);
const T &operator[](ULong index) const;
};

class V2// bounded sequence
{
  public:
V2();
V2(ULong length, T *data, Boolean release = FALSE);
V2(const V2&);
~V2();
V2 &operator=(const V2&);

ULong maximum() const;

void length(ULong);
ULong length() const;

T &operator[](ULong index);
const T &operator[](ULong index) const;
};
```

20.13.2 Using the "release" Constructor Parameter

Consider the following example:

```
// IDL
typedef sequence<string, 3> StringSeq;

// C++
char *static_arr[] = {"one", "two", "three"};
char **dyn_arr = StringSeq::allocbuf(3);
dyn_arr[0] = string_dup("one");
dyn_arr[1] = string_dup("two");
dyn_arr[2] = string_dup("three");

StringSeq seq1(3, static_arr);
StringSeq seq2(3, dyn_arr, TRUE);

seq1[1] = "2";// no free, no copy
char *str = string_dup("2");
seq2[1] = str;// free old storage, no copy
```

In this example, both **seq1** and **seq2** are constructed using user-specified data, but only **seq2** is told to assume management of the user memory (because of the **release=TRUE** parameter in its constructor). When assignment occurs into **seq1[1]**, the right-hand side is not copied, nor is anything freed because the sequence does not manage the user memory. When assignment occurs into **seq2[1]**, however, the old user data must be freed before ownership of the right-hand side can be assumed, since **seq2** manages the user memory. When **seq2** goes out of scope, it will call **string_free** for each of its elements and **freebuf** on the buffer given to it in its constructor.

When the **release** flag is set to **TRUE** and the sequence element type is either a string or an object reference type, the sequence will individually release each element before releasing the contents buffer. It will release strings using **string_free**, and it will release object references using the **release** function from the **CORBA** namespace.

In general, assignment should never take place into a sequence element via **operator[]** unless **release=TRUE** due to the possibility for memory management errors. In particular, a sequence constructed with **release=FALSE** should never be passed as an **inout** parameter because the callee has no way to determine the setting of the **release** flag, and thus must always assume that **release** is set to **TRUE**. Code that creates a sequence with **release=FALSE** and then knowingly and correctly manipulates it in that state, as shown with **seq1** in the example above, is compliant, but care should always be taken to avoid memory leaks under these circumstances.

As with other **out** and return values, **out** and return sequences must not be assigned to by the caller without first copying them. This is more fully explained in Section 20.20, “Argument Passing Considerations,” on page 20-62.

When a sequence is constructed with **release=TRUE**, a compliant application should make no assumptions about the continued lifetime of the data buffer passed to the constructor, since a compliant sequence implementation is free to copy the buffer and immediately free the original pointer.

20.13.3 Additional Memory Management Functions

ORB implementations concerned with single-process interoperability with the C mapping may overload **operator new()** and **operator delete()** for sequences so that dynamic allocation uses the same mechanism as the C language dynamic allocation functions. Whether these operators are overloaded by the implementation or not, compliant programs use **new** to dynamically allocate sequences and **delete** to free them.

Sequences also provide additional memory management functions for their buffers. For a sequence of type T, the following static member functions are provided in the sequence class public interface:

```
// C++
static T *allocbuf(ULong nelems);
static void freebuf(T *);
```

The **allocbuf** function allocates a vector of T elements that can be passed to the **T *data** constructor. The length of the vector is given by the **nelems** function argument. The **allocbuf** function initializes each element using its default constructor, except for strings, which are initialized to null pointers, and object references, which are initialized to suitably-typed nil object references. A null pointer is returned if **allocbuf** for some reason cannot allocate the requested vector. Vectors allocated by **allocbuf** should be freed using the **freebuf** function. The freebuf function ensures that the destructor for each element is called before the buffer is destroyed, except for string elements, which are freed using **string_free()**, and object reference elements, which are freed using **release()**. The **freebuf** function will ignore null pointers passed to it. Neither **allocbuf** nor **freebuf** may throw CORBA exceptions.

20.13.4 Sequence T_var and T_out Types

In addition to the regular operations defined for **T_var** and **T_out** types, the **T_var** and **T_out** for a sequence type also supports an overloaded **operator[]** that forwards requests to the **operator[]** of the underlying sequence.[13] This subscript operator should have the same return type as that of the corresponding operator on the underlying sequence type.

20.14 Mapping For Array Types

Arrays are mapped to the corresponding C++ array definition, which allows the definition of statically-initialized data using the array. If the array element is a string or an object reference, then the mapping uses the same type as for structure members. That is, assignment to an array element will release the storage associated with the old value.

```
// IDL
typedef float F[10];
typedef string V[10];
typedef string M[1][2][3];
void op(out F p1, out V p2, out M p3);
```

13.Note that since **T_var** and **T_out** types do not handle **const T***, there is no need to provide the const version of **operator[]** for **Sequence_var** and **Sequence_out** types.

```
// C++
typedef CORBA::Float F[10];
typedef ... V[10];// underlying type not shown because
typedef ... M[1][2][3];// it is implementation-dependent
F f1; F_var f2;
V v1; V_var v2;
M m1; M_var m2;
f(f2, v2, m2);
f1[0] = f2[1];
v1[1] = v2[1];// free old storage, copy
m1[0][1][2] = m2[0][1][2];// free old storage, copy
```

In the above example, the last two assignments result in the storage associated with the old value of the left-hand side being automatically released before the value from the right-hand side is copied.

As shown in “Basic Argument and Result Passing” on page 20-66, **out** and return arrays are handled via pointer to array *slice*, where a slice is an array with all the dimensions of the original specified except the first one. As a convenience for application declaration of slice types, the mapping also provides a typedef for each array slice type. The name of the slice typedef consists of the name of the array type followed by the suffix “_slice”. For example:

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

```
// C++
typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
```

Both the **T_var** type and the **T_out** type for an array should overload **operator[]** instead of **operator->**. The use of array slices also means that the **T_var** type and the **T_out** type for an array should have a constructor and assignment operator that each take a pointer to array slice as a parameter, rather than **T***. The **T_var** for the previous example would be:

```
// C++
class LongArray_var
{
  public:
LongArray_var();
LongArray_var(LongArray_slice*);
LongArray_var(const LongArray_var &);
~LongArray_var();
LongArray_var &operator=(LongArray_slice*);
LongArray_var &operator=(const LongArray_var &);

LongArray_slice &operator[](ULong index);
const LongArray_slice &operator[](Ulong index) const;

const LongArray_slice* in() const;
LongArray_slice* inout();
LongArray_slice* out();
LongArray_slice* _retn();

// other conversion operators to support
// parameter passing
};
```

Because arrays are mapped into regular C++ arrays, they present special problems for the type-safe **any** mapping described in “Mapping for the Any Type” on page 20-46. To facilitate their use with the **any** mapping, a compliant implementation must also provide for each array type a distinct C++ type whose name consists of the array name followed by the suffix _forany. These types must be distinct so as to allow functions to be overloaded on them. Like **Array_var** types, **Array_forany** types allow access to the underlying array type, but unlike **Array_var**, the **Array_forany** type does not **delete** the storage of the underlying array upon its own destruction. This is because the **Any** mapping retains storage ownership, as described in “Extraction from any” on page 20-49.

The interface of the **Array_forany** type is identical to that of the **Array_var** type, but it may not be implemented as a typedef to the **Array_var** type by a compliant implementation since it must be distinguishable from other types for purposes of function overloading. Also, the **Array_forany** constructor taking an **Array_slice*** parameter also takes a **Boolean** *nocopy* parameter which defaults to **FALSE**:

```
// C++
class Array_forany
{
  public:
Array_forany(Array_slice*, Boolean nocopy = FALSE);
...
};
```

The *nocopy* flag allows for a non-copying insertion of an **`Array_slice*`** into an **`Any`**.

Each **`Array_forany`** type must be defined at the same level of nesting as its **`Array`** type.

For dynamic allocation of arrays, compliant programs must use special functions defined at the same scope as the array type. For array T, the following functions will be available to a compliant program:

```
// C++
T_slice *T_alloc();
T_slice *T_dup(const T_slice*);
void T_free(T_slice *);
```

The **`T_alloc`** function dynamically allocates an array, or returns a null pointer if it cannot perform the allocation. The **`T_dup`** function dynamically allocates a new array with the same size as its array argument, copies each element of the argument array into the new array, and returns a pointer to the new array. If allocation fails, a null pointer is returned. The **`T_free`** function deallocates an array that was allocated with **`T_alloc`** or **`T_dup`**. Passing a null pointer to **`T_free`** is acceptable and results in no action being performed. These functions allow ORB implementations to utilize special memory management mechanisms for array types if necessary, without forcing them to replace global **`operator new`** and **`operator new[]`**.

The **`T_alloc`**, **`T_dup`**, and **`T_free`** functions may not throw CORBA exceptions.

20.15 Mapping For Typedefs

A typedef creates an alias for a type. If the original type maps to several types in C++, then the typedef creates the corresponding alias for each type. The example below illustrates the mapping.

```
// IDL
typedef long T;
interface A1;
typedef A1 A2;
typedef sequence<long> S1;
typedef S1 S2;

// C++
typedef Long T;

// ...definitions for A1...

typedef A1 A2;
typedef A1_ptr A2_ptr;
typedef A1_var A2_var;

// ...definitions for S1...

typedef S1 S2;
typedef S1_var S2_var;
```

For a typedef of an IDL type that maps to multiple C++ types, such as arrays, the typedef maps to all of the same C++ types and functions that its base type requires. For example:

```
// IDL
typedef long array[10];
typedef array another_array;


// C++
// ...C++ code for array not shown...
typedef array another_array;
typedef array_var another_array_var;
typedef array_slice another_array_slice;
typedef array_forany another_array_forany;

inline another_array_slice *another_array_alloc() {
return array_alloc();
}

inline another_array_slice*
another_array_dup(another_array_slice *a) {
return array_dup(a);
}
inline void another_array_free(another_array_slice *a) {
array_free(a);
}
```

20.16 Mapping for the Any Type

A C++ mapping for the OMG IDL type **any** must fulfill two different requirements:

- Handling C++ types in a type-safe manner.
- Handling values whose types are not known at implementation compile time.

The first item covers most normal usage of the **any** type—the conversion of typed values into and out of an **any**. The second item covers situations such as those involving the reception of a request or response containing an **any** that holds data of a type unknown to the receiver when it was created with a C++ compiler.

20.16.1 Handling Typed Values

To decrease the chances of creating an **any** with a mismatched **TypeCode** and value, the C++ function overloading facility is utilized. Specifically, for each distinct type in an OMG IDL specification, overloaded functions to insert and extract values of that type are provided by each ORB implementation. Overloaded operators are used for these functions so as to completely avoid any name space pollution. The nature of these functions, which are described in detail below, is that the appropriate **TypeCode** is implied by the C++ type of the value being inserted into or extracted from the **any**.

Since the type-safe **any** interface described below is based upon C++ function overloading, it requires C++ types generated from OMG IDL specifications to be distinct. However, there are special cases in which this requirement is not met:

- As noted in Section 20.5, “Mapping for Basic Data Types,” on page 20-15, the **boolean**, **octet**, and **char** OMG IDL types are not required to map to distinct C++ types, which means that a separate means of distinguishing them from each other for the purpose of function overloading is necessary. The means of distinguishing these types from each other is described in “Distinguishing boolean, octet, char, wchar, bounded string, and bounded wstring” on page 20-52.
- Since all strings and wide strings are mapped to **`char*`** and **`WChar*`**, respectively, regardless of whether they are bounded or unbounded, another means of creating or setting an **any** with a bounded string or wide string value is necessary. This is described in “Distinguishing boolean, octet, char, wchar, bounded string, and bounded wstring” on page 20-52.
- In C++, arrays within a function argument list decay into pointers to their first elements. This means that function overloading cannot be used to distinguish between arrays of different sizes. The means for creating or setting an **any** when dealing with arrays is described below and in “Mapping For Array Types” on page 20-41.

20.16.2 Insertion into *any*

To allow a value to be set in an **any** in a type-safe fashion, an ORB implementation must provide the following overloaded operator function for each separate OMG IDL type **T**.

```
// C++
void operator<<=(Any&, T);
```

This function signature suffices for types that are normally passed by value:

- **Short**, **UShort**, **Long**, **ULong**, **LongLong**, **ULongLong**, **Float**, **Double**, **LongDouble**
- Enumerations
- Unbounded strings and wide strings (**char*** and **WChar*** passed by value)
- Object references (**T_ptr**)

For values of type **T** that are too large to be passed by value efficiently, such as structs, unions, sequences, fixed types, **Any**, and exceptions, two forms of the insertion function are provided.

```
// C++
void operator<<=(Any&, const T&);// copying form
void operator<<=(Any&, T*);// non-copying form
```

Note that the copying form is largely equivalent to the first form shown, as far as the caller is concerned.

These “left-shift-assign” operators are used to insert a typed value into an **any** as follows.

```
// C++
Long value = 42;
Any a;
a <<= value;
```

In this case, the version of **operator<<=** overloaded for type **Long** must be able to set both the value and the **TypeCode** properly for the **any** variable.

Setting a value in an **any** using **operator<<=** means that:

- For the copying version of **operator<<=**, the lifetime of the value in the **any** is independent of the lifetime of the value passed to **operator<<=**. The implementation of the **any** may not store its value as a reference or pointer to the value passed to **operator<<=**.
- For the noncopying version of **operator<<=**, the inserted **T*** is consumed by the **any**. The caller may not use the **T*** to access the pointed-to data after insertion, since the **any** assumes ownership of it, and it may immediately copy the pointed-to data and destroy the original.
- With both the copying and non-copying versions of **operator<<=**, any previous value held by the **Any** is properly deallocated. For example, if the **Any(TypeCode_ptr,void*,TRUE)** constructor (described in “Handling Untyped Values” on page 20-56) was called to create the **Any**, the **Any** is responsible for deallocating the memory pointed to by the **void*** before copying the new value.

Copying insertion of a string type or wide string type causes one of the following functions to be invoked:

```
// C++
void operator<<=(Any&, const char*);
void operator<<=(Any&, const WChar*);
```

Since all string types are mapped to **`char*`**, and all wide string types are mapped to **`WChar*`** , these insertion functions assume that the value being inserted are unbounded. “Distinguishing boolean, octet, char, wchar, bounded string, and bounded wstring” on page 20-52 describes how bounded strings and bounded wide strings may be correctly inserted into an **`Any`**. Non-copying insertion of both bounded and unbounded strings can be achieved using the **`Any::from_string`** helper type. Similarly, non-copying insertion of bounded and unbounded wide strings strings can be achieved using the**`Any::from_wstring`** helper type. Both of these helper types are described in “Distinguishing boolean, octet, char, wchar, bounded string, and bounded wstring” on page 20-52.

Type-safe insertion of arrays uses the **`Array_forany`** types described in “Mapping For Array Types” on page 20-41. Compliant implementations must provide a version of **`operator<<=`** overloaded for each **`Array_forany`** type. For example:

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

```
// C++
typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
class LongArray_forany { ... };

void operator<<=(Any &, const LongArray_forany &);
```

The **`Array_forany`** types are always passed to **`operator<<=`** by reference to const. The *nocopy* flag in the **`Array_forany`** constructor is used to control whether the inserted value is copied (*nocopy* == **`FALSE`**) or consumed (*nocopy* == **`TRUE`**). Because the *nocopy* flag defaults to **`FALSE`**, copying insertion is the default.

Because of the type ambiguity between an array of **`T`** and a **`T*`**, it is highly recommended that portable code explicitly[14] use the appropriate **`Array_forany`** type when inserting an array into an **`any`**:

14. A mapping implementor may use the new C++ key word “explicit” to prevent implicit conversions through the **Array_forany** constructor, but this feature is not yet widely available in current C++ compilers.

```
// IDL
struct S {... };
typedef S SA[5];

// C++
struct S { ... };
typedef S SA[5];
typedef S SA_slice;
class SA_forany { ... };

SA s;
// ...initialize s...
Any a;
a <<= s;                // line 1
a <<= SA_forany(s);     // line 2
```

Line 1 results in the invocation of the noncopying **operator<<=(Any&, S*)** due to the decay of the **SA** array type into a pointer to its first element, rather than the invocation of the copying **SA_forany** insertion operator. Line 2 explicitly constructs the **SA_forany** type and thus results in the desired insertion operator being invoked.

The noncopying version of **operator<<=** for object references takes the address of the **T_ptr** type.

```
// IDL
interface T { ... };

// C++
void operator<<=(Any&, T_ptr);      // copying
void operator<<=(Any&, T_ptr*);     // non-copying
```

The noncopying object reference insertion consumes the object reference pointed to by **T_ptr***; therefore after insertion the caller may not access the object referred to by **T_ptr** since the **any** may have duplicated and then immediately released the original object reference. The caller maintains ownership of the storage for the **T_ptr** itself.

The copying version of **operator<<=** is also supported on the **Any_var** type. Note that due to the conversion operators that convert **Any_var** to **Any&** for parameter passing, only those **operator<<=** functions defined as member functions of **any** need to be explicitly defined for **Any_var**.

20.16.3 Extraction from *any*

To allow type-safe retrieval of a value from an **any**, the mapping provides the following operators for each OMG IDL type **T**:

```
// C++
Boolean operator>>=(const Any&, T&);
```

This function signature suffices for primitive types that are normally passed by value. For values of type **T** that are too large to be passed by value efficiently, such as structs, unions, sequences, fixed types, **Any**, and exceptions, this function may be prototyped as follows:

```
// C++
Boolean operator>>=(const Any&, T*&);
```

The first form of this function is used only for the following types:

- **Boolean**, **Char**, **Octet**, **Short**, **UShort**, **Long**, **ULong**, **LongLong**, **ULongLong**, **Float**, **Double**, **LongDouble**
- Enumerations
- Unbounded stringsand wide strings (**char*** and **WChar*** passed by reference, i.e., **char*&** and **WChar*&**)
- Object references (**T_ptr**)

For all other types, the second form of the function is used.

All versions of **operator>>=** implemented as member functions of class **Any**, such as those for primitive types, should be marked as **const**.

This “right-shift-assign” operator is used to extract a typed value from an **any** as follows:

```
// C++
Long value;
Any a;
a <<= Long(42);
if (a >>= value) {
      // ... use the value ...
}
```

In this case, the version of **operator>>=** for type **Long** must be able to determine whether the **Any** truly does contain a value of type **Long** and, if so, copy its value into the reference variable provided by the caller and return **TRUE**. If the **Any** does not contain a value of type **Long**, the value of the caller’s reference variable is not changed, and **operator>>=** returns **FALSE**.

For non-primitive types, such as struct, union, sequence, exception, **Any**, and fixed types, extraction is done by pointer. For example, consider the following IDL struct:

```
// IDL
struct MyStruct {
    long lmem;
    short smem;
};
```

Such a struct could be extracted from an **any** as follows:

```
// C++
Any a;
// ... a is somehow given a value of type MyStruct ...
MyStruct *struct_ptr;
if (a >>= struct_ptr) {
      // ... use the value ...
}
```

If the extraction is successful, the caller's pointer will point to storage managed by the **any**, and **operator>>=** will return **TRUE**. The caller must not try to **delete** or otherwise release this storage. The caller also should not use the storage after the contents of the **any** variable are replaced via assignment, insertion, or the **replace** function, or after the **any** variable is destroyed. Care must be taken to avoid using **T_var** types with these extraction operators, since they will try to assume responsibility for deleting the storage owned by the **any**.

If the extraction is not successful, the value of the caller's pointer is set equal to the null pointer, and **operator>>=** returns **FALSE**.

Correct extraction of array types relies on the **Array_forany** types described in"Mapping For Array Types" on page 20-41.

```
// IDL
typedef long A[20];
typedef A B[30][40][50];
```

```
// C++
typedef Long A[20];
typedef Long A_slice;
class A_forany { ... };
typedef A B[30][40][50];
typedef A B_slice[40][50];
class B_forany { ... };

Boolean operator>>=(const Any &, A_forany&);// for type A
Boolean operator>>=(const Any &, B_forany&);// for type B
```

The **Array_forany** types are always passed to **operator>>=** by reference.

For strings, wide strings, and arrays, applications are responsible for checking the **TypeCode** of the **any** to be sure that they do not overstep the bounds of the array, string, or wide string object when using the extracted value.

The **operator>>=** is also supported on the **Any_var** type. Note that due to the conversion operators that convert **Any_var** to **const Any&** for parameter passing, only those **operator>>=** functions defined as member functions of **any** need to be explicitly defined for **Any_var**.

20.16.4 Distinguishing boolean, octet, char, wchar, bounded string, and bounded wstring

Since the **boolean**, **octet**, **char**, and **wchar** OMG IDL types are not required to map to distinct C++ types, another means of distinguishing them from each other is necessary so that they can be used with the type-safe **any** interface. Similarly, since both bounded and unbounded strings map to **char***, and both bounded and unbounded wide strings map to **WChar***, another means of distinguishing them must be provided. This is done by introducing several new helper types nested in the **any** class interface. For example, this can be accomplished as shown next.

```
// C++
class Any
{
  public:
   // special helper types needed for boolean, octet, char,
   // and bounded string insertion
   struct from_boolean {
      from_boolean(Boolean b) : val(b) {}
      Boolean val;
   };
   struct from_octet {
      from_octet(Octet o) : val(o) {}
      Octet val;
   };
   struct from_char {
      from_char(Char c) : val(c) {}
      Char val;
   };
   struct from_wchar {
      from_wchar(WChar wc) : val(wc) {}
      WChar val;
   };
   struct from_string {
      from_string(char* s, ULong b,
            Boolean nocopy = FALSE) :
         val(s), bound(b) {}
      char *val;
      ULong bound;
   };
   struct from_wstring {
      from_wstring(WChar* s, ULong b,
            Boolean nocopy = FALSE) :
         val(s), bound(b) {}
      WChar *val;
      ULong bound;
```

```
};

void operator<<=(from_boolean);
void operator<<=(from_char);
void operator<<=(from_wchar);
void operator<<=(from_octet);
void operator<<=(from_string);
void operator<<=(from_wstring);

// special helper types needed for boolean, octet,
// char, and bounded string extraction
struct to_boolean {
   to_boolean(Boolean &b) : ref(b) {}
   Boolean &ref;
};
struct to_char {
   to_char(Char &c) : ref(c) {}
   Char &ref;
};
struct to_wchar {
   to_wchar(WChar &wc) : ref(wc) {}
   WChar &ref;
};
struct to_octet {
   to_octet(Octet &o) : ref(o) {}
   Octet &ref;
};
struct to_string {
   to_string(char *&s, ULong b) : val(s), bound(b) {}
   char *&val;
   ULong bound;
};
struct to_wstring {
   to_wstring(WChar *&s, ULong b) : val(s), bound(b) {}
   WChar *&val;
   ULong bound;
};

Boolean operator>>=(to_boolean) const;
Boolean operator>>=(to_char) const;
```

```
   Boolean operator>>=(to_wchar) const;
   Boolean operator>>=(to_octet) const;
   Boolean operator>>=(to_string) const;
   Boolean operator>>=(to_wstring) const;

   // other public Any details omitted

private:
   // these functions are private and not implemented
   // hiding these causes compile-time errors for
   // unsigned char
   void operator<<=(unsigned char);
   Boolean operator>>=(unsigned char &) const;
};
```

An ORB implementation provides the overloaded **operator<<=** and **operator>>=** functions for these special helper types. These helper types are used as shown next.

```
// C++
Boolean b = TRUE;
Any any;
any <<= Any::from_boolean(b);
// ...
if (any >>= Any::to_boolean(b)) {
   // ...any contained a Boolean...
}

char* p = "bounded";
any <<= Any::from_string(p, 8);
// ...
if (any >>= Any::to_string(p, 8)) {
   // ...any contained a string<8>...
}
```

A bound value of zero passed to the appropriate helper type indicates an unbounded string or wide string.

For noncopying insertion of a bounded or unbounded string into an **any**, the **nocopy** flag on the **from_string** constructor should be set to **TRUE**.

```
// C++
char* p = string_alloc(8);
// ...initialize string p...
any <<= Any::from_string(p, 8, 1);  // any consumes p
```

The same rules apply for bounded and unbounded wide strings and the **from_wstring** helper type.

Assuming that **boolean**, **char**, and **octet** all map the C++ type **unsigned char**, the private and unimplemented **operator<<=** and **operator>>=** functions for **unsigned char** will cause a compile-time error if straight insertion or extraction of any of the **boolean**, **char**, or **octet** types is attempted.

```
// C++
Octet oct = 040;
Any any;
any <<= oct;// this line will not compile
any <<= Any::from_octet(oct);// but this one will
```

It is important to note that the previous example is only one possible implementation for these helpers, not a mandated one. Other compliant implementations are possible, such as providing them via in-lined static **any** member functions if **boolean**, **char**, and **octet** are in fact mapped to distinct C++ types. All compliant C++ mapping implementations must provide these helpers, however, for purposes of portability.

20.16.5 Widening to Object

Sometimes it is desirable to extract an object reference from an **Any** as the base **Object** type. This can be accomplished using a helper type similar to those required for extracting **Boolean**, **Char**, and **Octet**:

```
// C++
class Any
{
  public:
...
struct to_object {
to_object(Object_ptr &obj) : ref(obj) {}
Object_ptr &ref;
};
Boolean operator>>=(to_object) const;
...
};
```

The **to_object** helper type is used to extract an object reference from an **Any** as the base **Object** type. If the **Any** contains a value of an object reference type as indicated by its **TypeCode**, the extraction function **operator>>=(to_object)** explicitly widens its contained object reference to **Object** and returns true, otherwise it returns false. This is the only object reference extraction function that performs widening on the extracted object reference. As with regular object reference extraction, no duplication of the object reference is performed by the **to_object** extraction operator.

20.16.6 Handling Untyped Values

Under some circumstances the type-safe interface to **Any** is not sufficient. An example is a situation in which data types are read from a file in binary form and used to create values of type **Any**. For these cases, the **Any** class provides a constructor with an explicit **TypeCode** and generic pointer:

```
// C++
Any(TypeCode_ptr tc, void *value, Boolean release = FALSE);
```

The constructor is responsible for duplicating the given **TypeCode** pseudo object reference. If the **release** parameter is **TRUE**, then the **Any** object assumes ownership of the storage pointed to by the **value** parameter. A compliant application should make no assumptions about the continued lifetime of the **value** parameter once it has been handed to an **Any** with **release=TRUE**, since a compliant **Any** implementation is allowed to copy the **value** parameter and immediately free the original pointer. If the **release** parameter is **FALSE** (the default case), then the **Any** object assumes the caller will manage the memory pointed to by **value**. The **value** parameter can be a null pointer.

The **Any** class also defines three unsafe operations:

```
// C++
void replace(
TypeCode_ptr,
void *value,
Boolean release = FALSE
);
TypeCode_ptr type() const;
const void *value() const;
```

The **replace** function is intended to be used with types that cannot be used with the type-safe insertion interface, and so is similar to the constructor described above. The existing **TypeCode** is released and value storage deallocated, if necessary. The **TypeCode** function parameter is duplicated. If the **release** parameter is **TRUE**, then the **Any** object assumes ownership for the storage pointed to by the **value** parameter. A compliant application should make no assumptions about the continued lifetime of the **value** parameter once it has been handed to the **Any::replace** function with **release=TRUE**, since a compliant **Any** implementation is allowed to copy the **value** parameter and immediately free the original pointer. If the **release** parameter is **FALSE** (the default case), then the **Any** object assumes the caller will manage the memory occupied by the value. The **value** parameter of the **replace** function can be a null pointer.

For C++ mapping implementations that use **Environment** parameters to pass exception information, the default **release** argument can be simulated by providing two overloaded **replace** functions, one that takes a non-defaulted **release** parameter and one that takes no **release** parameter. The second function simply invokes the first with the **release** parameter set to **FALSE**.

Note that neither the constructor shown above nor the **replace** function is type-safe. In particular, no guarantees are made by the compiler or runtime as to the consistency between the **TypeCode** and the actual type of the **void*** argument. The behavior of an ORB implementation when presented with an **Any** that is constructed with a mismatched **TypeCode** and value is not defined.

The **type** function returns a **TypeCode_ptr** pseudo-object reference to the **TypeCode** associated with the **Any**. Like all object reference return values, the caller must release the reference when it is no longer needed, or assign it to a **TypeCode_var** variable for automatic management.

The **value** function returns a pointer to the data stored in the **Any**. If the **Any** has no associated value, the **value** function returns a null pointer. The type to which the **void*** returned by the **value** function may be cast depends on the ORB implementation; thus, use of the **value** function is not portable across ORB implementations and its usage is therefore deprecated. Note that ORB implementations are allowed to make stronger guarantees about the **void*** returned from the **value** function, if so desired.

20.16.7 Any Constructors, Destructor, Assignment Operator

The default constructor creates an **Any** with a **TypeCode** of type **tk_null**, and no value. The copy constructor calls **_duplicate** on the **TypeCode_ptr** of its **Any** parameter and deep-copies the parameter's value. The assignment operator releases its own **TypeCode_ptr** and deallocates storage for the current value if necessary, then duplicates the **TypeCode_ptr** of its **Any** parameter and deep-copies the parameter's value. The destructor calls **release** on the **TypeCode_ptr** and deallocates storage for the value, if necessary.

Other constructors are described in Section 20.16.6, "Handling Untyped Values," on page 20-56.

ORB implementations concerned with single-process interoperability with the C mapping may overload **operator new()** and **operator delete()** for **Any**s so that dynamic allocation uses the same mechanism as the C language dynamic allocation functions. Whether these operators are overloaded by the implementation or not, compliant programs use **new** to dynamically allocate anys and **delete** to free them.

20.16.8 The Any Class

The full definition of the **Any** class can be found in "The Any Class" on page 20-57.

20.16.9 The Any_var Class

Since **Any**s are returned via pointer as **out** and return parameters (see Table 20-2 on page 20-66), there exists an **Any_var** class similar to the **T_var** classes for object references. **Any_var** obeys the rules for **T_var** classes described in "Mapping for

Structured Types” on page 20-21, calling **delete** on its **Any*** when it goes out of scope or is otherwise destroyed. The full interface of the **Any_var** class is shown in “Any_var Class” on page 20-107.

20.17 Mapping for Exception Types

An OMG IDL exception is mapped to a C++ class that derives from the standard **UserException** class defined in the **CORBA** module (see “CORBA Module” on page 20-5). The generated class is like a variable-length struct, regardless of whether or not the exception holds any variable-length members. Just as for variable-length structs, each exception member must be self-managing with respect to its storage.

The copy constructor, assignment operator, and destructor automatically copy or free the storage associated with the exception. For convenience, the mapping also defines a constructor with one parameter for each exception member—this constructor initializes the exception members to the given values. For exception types that have a string member, this constructor should take a **const char*** parameter, since the constructor must copy the string argument. Similarly, constructors for exception types that have an object reference member must call **_duplicate** on the corresponding object reference constructor parameter. The default constructor performs no explicit member initialization.

```
// C++
class Exception
{
  public:
virtual ~Exception();

virtual void _raise() = 0;
};
```

The **Exception** base class is abstract and may not be instantiated except as part of an instance of a derived class. It supplies one pure virtual function to the exception hierarchy: the **_raise()** function which can be used to tell an exception instance to **throw** itself so that a **catch** clause can catch it by a more derived type. Each class derived from **Exception** shall implement **_raise()** as follows:

```
// C++
void SomeDerivedException::_raise()
{
throw *this;
}
```

For environments that do not support exception handling, please refer to “Without Exception Handling” on page 20-116 for information about the **_raise()** function.

The **UserException** class is derived from a base **Exception** class, which is also defined in the **CORBA** module.

All standard exceptions are derived from a **SystemException** class, also defined in the **CORBA** module. Like **UserException**, **SystemException** is derived from the base **Exception** class. The **SystemException** class interface is shown below.

```
// C++
enum CompletionStatus {
COMPLETED_YES,
COMPLETED_NO,
COMPLETED_MAYBE
};

class SystemException : public Exception
{
  public:
SystemException();
SystemException(const SystemException &);
SystemException(ULong minor, CompletionStatus status);
~SystemException();
SystemException &operator=(const SystemException &);

ULong minor() const;
void minor(ULong);

void _raise();

CompletionStatus completed() const;
void completed(CompletionStatus);
};
```

The default constructor for **SystemException** causes **minor()** to return 0 and **completed()** to return **COMPLETED_NO.**

Each specific system exception (described in “Exceptions” on page 19-4) is derived from **SystemException**:

```
// C++
class UNKNOWN : public SystemException { ... };
class BAD_PARAM : public SystemException { ... };
// etc.
```

All specific system exceptions are defined within the **CORBA** module.

This exception hierarchy allows any exception to be caught by simply catching the **Exception** type:

```
// C++
try {
...
} catch (const Exception &exc) {
...
}
```

Alternatively, all user exceptions can be caught by catching the **UserException** type, and all system exceptions can be caught by catching the **SystemException** type:

```
// C++
try {
...
} catch (const UserException &ue) {
...
} catch (const SystemException &se) {
...
}
```

Naturally, more specific types can also appear in **catch** clauses.

Exceptions are normally thrown by value and caught by reference. This approach lets the exception destructor release storage automatically.

The **Exception** class provides for narrowing within the exception hierarchy:

```
// C++
class UserException : public Exception
{
  public:
static UserException *_narrow(Exception *);
// ...
};

class SystemException : public Exception
{
  public:
static SystemException *_narrow(Exception *);
// ...
};
```

Each exception class supports a static member function named **_narrow**. The parameter to the **_narrow** call is a pointer to the base class **Exception**. If the parameter is a null pointer, the return type of **_narrow** is a null pointer. If the actual (runtime) type of the parameter exception can be widened to the requested exception's type, then **_narrow** will return a valid pointer to the parameter **Exception**. Otherwise, **_narrow** will return a null pointer.

Unlike the **_narrow** operation on object references, the **_narrow** operation on exceptions returns a suitably-typed pointer to the same exception parameter, not a pointer to a new exception. If the original exception goes out of scope or is otherwise destroyed, the pointer returned by **_narrow** is no longer valid.

For application portability, conforming C++ mapping implementations built using C++ compilers that support the standard C++ Run Time Type Information (RTTI) mechanisms still need to support narrowing for the **Exception** hierarchy. RTTI supports, among other things, determination of the run-time type of a C++ object. In particular, the **dynamic_cast<T*>** operator[15] allows for narrowing from a base pointer to a more derived pointer if the object pointed to really is of the more derived type. This operator is not useful for narrowing object references, since it cannot determine the actual type of remote objects, but it can be used by the C++ mapping implementation to narrow within the exception hierarchy.

Request invocations made through the DII may result in user-defined exceptions that cannot be fully represented in the calling program because the specific exception type was not known at compile-time. The mapping provides the **UnknownUserException** so that such exceptions can be represented in the calling process:

```
// C++
class UnknownUserException : public UserException
{
  public:
Any &exception();
};
```

As shown here, **UnknownUserException** is derived from **UserException**. It provides the **exception()** accessor that returns an **Any** holding the actual exception. Ownership of the returned **Any** is maintained by the **UnknownUserException**—the **Any** merely allows access to the exception data. Conforming applications should never explicitly throw exceptions of type **UnknownUserException**—it is intended for use with the DII.

20.18 Mapping For Operations and Attributes

An operation maps to a C++ function with the same name as the operation. Each read-write attribute maps to a pair of overloaded C++ functions (both with the same name), one to set the attribute's value and one to get the attribute's value. The *set* function takes an **in** parameter with the same type as the attribute, while the *get* function takes no parameters and returns the same type as the attribute. An attribute marked **readonly** maps to only one C++ function, to get the attribute's value. Parameters and return types for attribute functions obey the same parameter passing rules as for regular operations.

15.It is unlikely that a compiler would support RTTI without supporting exceptions, since much of a C++ exception handling implementation is based on RTTI mechanisms.

OMG IDL **oneway** operations are mapped the same as other operations; that is, there is no way to know by looking at the C++ whether an operation is **oneway** or not.

The mapping does not define whether exceptions specified for an OMG IDL operation are part of the generated operation's type signature or not.

```
// IDL
interface A
{
    void f();
    oneway void g();
    attribute long x;
};

// C++
A_var a;
a->f();
a->g();
Long n = a->x();
a->x(n + 1);
```

Unlike the C mapping, C++ operations do not require an additional **Environment** parameter for passing exception information—real C++ exceptions are used for this purpose. See "Mapping for Exception Types" on page 20-58 for more details.

20.19 Implicit Arguments to Operations

If an operation in an OMG IDL specification has a context specification, then a **`Context_ptr`** input parameter (see "Context Interface" on page 20-80) follows all operation-specific arguments. In an implementation that does not support real C++ exceptions, an output **Environment** parameter is the last argument, following all operation-specific arguments, and following the context argument if present. The parameter passing mode for **Environment** is described in "Without Exception Handling" on page 20-116.

20.20 Argument Passing Considerations

The mapping of parameter passing modes attempts to balance the need for both efficiency and simplicity. For primitive types, enumerations, and object references, the modes are straightforward, passing the type *P* for primitives and enumerations and the type **`A_ptr`** for an interface type *A*.

Aggregate types are complicated by the question of when and how parameter memory is allocated and deallocated. Mapping **in** parameters is straightforward because the parameter storage is caller-allocated and read-only. The mapping for **out** and **inout** parameters is more problematic. For variable-length types, the callee must allocate some if not all of the storage. For fixed-length types, such as a *Point* type represented as a struct containing three floating point members, caller allocation is preferable (to allow stack allocation).

To accommodate both kinds of allocation, avoid the potential confusion of split allocation, and eliminate confusion with respect to when copying occurs, the mapping is **T&** for a fixed-length aggregate **T** and **T*&** for a variable-length **T**. This approach has the unfortunate consequence that usage for structs depends on whether the struct is fixed- or variable-length; however, the mapping is consistently **T_var&** if the caller uses the managed type **T_var**.

The mapping for **out** and **inout** parameters additionally requires support for deallocating any previous variable-length data in the parameter when a **T_var** is passed. Even though their initial values are not sent to the operation, we include **out** parameters because the parameter could contain the result from a previous call. There are many ways to implement this support. The mapping does not require a specific implementation, but a compliant implementation must free the inaccessible storage associated with a parameter passed as a **T_var** managed type. The provision of the **T_out** types is intended to give implementations the hooks necessary to free the inaccessible storage while converting from the **T_var** types. The following examples demonstrate the compliant behavior:

```
// IDL
struct S { string name; float age; };
void f(out S p);

// C++
S_var s;
f(s);
// use s
f(s); // first result will be freed

S *sp; // need not initialize before passing to out
f(sp);
// use sp
delete sp; // cannot assume next call will free old value
f(sp);
```

Note that implicit deallocation of previous values for **out** and **inout** parameters works only with **T_var** types, not with other types:

```
// IDL
void q(out string s);

// C++
char *s;
for (int i = 0; i < 10; i++)
q(s);// memory leak!
```

Each call to the **q** function in the loop results in a memory leak because the caller is not invoking **string_free** on the **out** result. There are two ways to fix this, as shown below:

```
// C++
char *s;
String_var svar;
for (int i = 0 ; i < 10; i++) {
q(s);
string_free(s);// explicit deallocation
// OR:
q(svar);// implicit deallocation
}
```

Using a plain **`char*`** for the **out** parameter means that the caller must explicitly deallocate its memory before each reuse of the variable as an **out** parameter, while using a **`String_var`** means that any deallocation is performed implicitly upon each use of the variable as an **out** parameter.

Variable-length data must be explicitly released before being overwritten. For example, before assigning to an **inout** string parameter, the implementor of an operation may first delete the old character data. Similarly, an **inout** interface parameter should be released before being reassigned. One way to ensure that the parameter storage is released is to assign it to a local **`T_var`** variable with an automatic release, as in the following example:

```
// IDL
interface A;
void f(inout string s, inout A obj);
```

```
// C++
void Aimpl::f(char *&s, A_ptr &obj) {
String_var s_tmp = s;
s = /* new data */;
A_var obj_tmp = obj;
obj = /* new reference */
}
```

To allow the callee the freedom to allocate a single contiguous area of storage for all the data associated with a parameter, we adopt the policy that the callee-allocated storage is not modifiable by the caller. However, trying to enforce this policy by returning a **`const`** type in C++ is problematic, since the caller is required to release the storage, and calling **`delete`** on a **`const`** object is an error[16]. A compliant mapping therefore is not required to detect this error.

For parameters that are passed or returned as a pointer (**`T*`**) or reference to pointer (**`T*&`**), a compliant program is not allowed to pass or return a null pointer; the result of doing so is undefined. In particular, a caller may not pass a null pointer under any of the following circumstances:

16.The upcoming ANSI/ISO C++ standard allows **`delete`** on a pointer to **`const`** object, but many C++ compilers do not yet support this feature.

- **in** and **inout** string
- **in** and **inout** array (pointer to first element)

A caller may pass a reference to a pointer with a null value for **out** parameters, however, since the callee does not examine the value but rather just overwrites it. A callee may not return a null pointer under any of the following circumstances:

- **out** and return variable-length struct
- **out** and return variable-length union
- **out** and return string
- **out** and return sequence
- **out** and return variable-length array, return fixed-length array
- **out** and return any

Since OMG IDL has no concept of pointers in general or null pointers in particular, allowing the passage of null pointers to or from an operation would project C++ semantics onto OMG IDL operations.[17] A compliant implementation is allowed but not required to raise a **BAD_PARAM** exception if it detects such an error.

20.20.1 Operation Parameters and Signatures

Table 20-2 on page 20-66 displays the mapping for the basic OMG IDL parameter passing modes and return type according to the type being passed or returned, while Table 20-3 on page 20-66 displays the same information for **`T_var`** types. “T_var Argument and Result Passing” is merely for informational purposes; it is expected that operation signatures for both clients and servers will be written in terms of the parameter passing modes shown in Table 20-2 on page 20-66, with the exception that the **`T_out`** types will be used as the actual parameter types for all **out** parameters. It is also expected that **`T_var`** types will support the necessary conversion operators to allow them to be passed directly. Callers should always pass instances of either **`T_var`** types or the base types shown in Table 20-2 on page 20-66, and callees should treat their **`T_out`** parameters as if they were actually the corresponding underlying types shown in “Basic Argument and Result Passing”.

In Table 20-2 on page 20-66, fixed-length arrays are the only case where the type of an **out** parameter differs from a return value, which is necessary because C++ does not allow a function to return an array. The mapping returns a pointer to a *slice* of the array, where a slice is an array with all the dimensions of the original specified except the first one. A caller is responsible for providing storage for all arguments passed as **in** arguments.

17.When real C++ exceptions are not available, however, it is important that null pointers are returned whenever an **`Environment`** containing an exception is returned; see “Without Exception Handling” on page 20-116 for more details.

Table 20-2 Basic Argument and Result Passing

Data Type	In	Inout	Out	Return
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
long long	LongLong	LongLong&	LongLong&	LongLong
unsigned short	UShort	UShort&	UShort&	UShort
unsigned long	ULong	ULong&	ULong&	ULong
unsigned long long	ULongLong	ULongLong&	ULongLong&	ULongLong
float	Float	Float&	Float&	Float
double	Double	Double&	Double&	Double
long double	LongDouble	LongDouble&	LongDouble&	LongDouble
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	const struct&	struct&	struct&	struct
struct, variable	const struct&	struct&	struct*&	struct*
union, fixed	const union&	union&	union&	union
union, variable	const union&	union&	union*&	union*
string	const char*	char*&	char*&	char*
wstring	const WChar*	WChar*&	WChar*&	WChar*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, fixed	const array	array	array	array slice*[2]
array, variable	const array	array	array slice*&[2]	array slice*[2]
any	const any&	any&	any*&	any*
fixed	const fixed&	fixed&	fixed&	fixed&

1. Including pseudo-object references.

2. A slice is an array with all the dimensions of the original except the first one.

Table 20-3 T_var Argument and Result Passing

Data Type	In	Inout	Out	Return
object reference var[1]	const objref_var&	objref_var&	objref_var&	objref_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
sequence_var	const sequence_var&	sequence_var&	sequence_var&	sequence_var

Table 20-3 T_var Argument and Result Passing

Data Type	In	Inout	Out	Return
array_var	const array_var&	array_var&	array_var&	array_var
any_var	const any_var&	any_var&	any_var&	any_var

1. Including pseudo-object references.

Table 20-4 on page 20-67 and Table 20-5 on page 20-68 describe the caller's responsibility for storage associated with **inout** and **out** parameters and for return results

Table 20-4 Caller Argument Storage Responsibilities

Type	Inout Param	Out Param	Return Result
short	1	1	1
long	1	1	1
long long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
unsigned long long	1	1	1
float	1	1	1
double	1	1	1
long double	1	1	1
boolean	1	1	1
char	1	1	1
wchar	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
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3
wstring	4	3	3
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3
fixed	1	1	1

.

Table 20-5 Argument Passing Cases

Case	
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::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 caller is responsible for the release of all out and return object references. Release of all object references embedded in other structures is performed automatically by the structures themselves.
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. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. 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*`** or **`wchar*`** pointing to it. Since the callee may deallocate the input string and reassign the **`char*`** or **`wchar*`** to point to new storage to hold the output value, the caller should allocate the input string using **`string_alloc()`** or **`wstring_alloc()`**. The size of the out string is therefore not limited by the size of the in string. The caller is responsible for deleting the storage for the out **`using string_free()`** or **`wstring_free()`**. 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 parameter with which the sequence or any was constructed.
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. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following 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.

20.21 Mapping of Pseudo Objects to C++

CORBA pseudo objects may be implemented either as normal CORBA objects or as *serverless objects*. In the CORBA specification, the fundamental differences between these strategies are:

- Serverless object types do not inherit from **CORBA::Object**
- Individual serverless objects are not registered with any ORB
- Serverless objects do not necessarily follow the same memory management rules as for regular IDL types.

References to serverless objects are not necessarily valid across computational contexts; for example, address spaces. Instead, references to serverless objects that are passed as parameters may result in the construction of independent functionally-identical copies of objects used by receivers of these references. To support this, the otherwise hidden representational properties (such as data layout) of serverless objects are made known to the ORB. Specifications for achieving this are not contained in this chapter: making serverless objects known to the ORB is an implementation detail.

This chapter provides a standard mapping algorithm for all pseudo object types. This avoids the need for piecemeal mappings for each of the nine CORBA pseudo object types, and accommodates any pseudo object types that may be proposed in future revisions of *CORBA*. It also avoids representation dependence in the C mapping while still allowing implementations that rely on C-compatible representations.

20.22 Usage

Rather than C-PIDL, this mapping uses an augmented form of full OMG IDL to describe serverless object types. Interfaces for pseudo object types follow the exact same rules as normal OMG IDL interfaces, with the following exceptions:

- They are prefaced by the keyword **pseudo.**
- Their declarations may refer to other[18] serverless object types that are not otherwise necessarily allowed in OMG IDL.

As explained in “Pseudo-objects” on page 19-29, the **pseudo** prefix means that the interface may be implemented in either a normal or serverless fashion. That is, apply either the rules described in the following sections or the normal mapping rules described in this chapter.

20.23 Mapping Rules

Serverless objects are mapped in the same way as normal interfaces, except for the differences outlined in this section.

Classes representing serverless object types are *not* subclasses of **CORBA::Object**, and are not necessarily subclasses of any other C++ class. Thus, they do not necessarily support, for example, the **Object::create_request** operation.

For each class representing a serverless object type T, overloaded versions of the following functions are provided in the **CORBA** namespace:

18.In particular, **exception** used as a data type and a function name.

```
// C++
void release(T_ptr);
Boolean is_nil(T_ptr p);
```

The mapped C++ classes are not guaranteed to be usefully subclassable by users, although subclasses can be provided by implementations. Implementations are allowed to make assumptions about internal representations and transport formats that may not apply to subclasses.

The member functions of classes representing serverless object types do not necessarily obey the normal memory management rules. This is due to the fact that some serverless objects, such as **`CORBA::NVList`**, are essentially just containers for several levels of other serverless objects. Requiring callers to explicitly free the values returned from accessor functions for the contained serverless objects would be counter to their intended usage.

All other elements of the mapping are the same. In particular:

1. The types of references to serverless objects, **`T_ptr`**, may or may not simply be a typedef of **`T*`**.

2. Each mapped class supports the following static member functions:

```
// C++
static T_ptr _duplicate(T_ptr p);
static T_ptr _nil();
```

Legal implementations of **`_duplicate`** include simply returning the argument or constructing references to a new instance. Individual implementations may provide stronger guarantees about behavior.

3. The corresponding C++ classes may or may not be directly instantiable or have other instantiation constraints. For portability, users should invoke the appropriate constructive operations.

4. As with normal interfaces, assignment operators are not supported.

5. Although they can transparently employ “copy-style” rather than “reference-style” mechanics, parameter passing signatures and rules as well as memory management rules are identical to those for normal objects, unless otherwise noted.

20.24 Relation to the C PIDL Mapping

All serverless object interfaces and declarations that rely on them have direct analogs in the C mapping. The mapped C++ classes can, but need not be, implemented using representations compatible to those chosen for the C mapping. Differences between the pseudo object specifications for C-PIDL and C++ PIDL are as follows:

- C++-PIDL calls for removal of representation dependencies through the use of interfaces rather than structs and typedefs.
- C++-PIDL calls for placement of operations on pseudo objects in their interfaces, including a few cases of redesignated functionality as noted.

- In C++-PIDL, the **release** performs the role of the associated **free** and **delete** operations in the C mapping, unless otherwise noted.

Brief descriptions and listings of each pseudo-interface and its C++ mapping are provided in the following sections. Further details, including definitions of types referenced but not defined below, may be found in the relevant sections of this document.

20.25 Environment

Environment provides a vehicle for dealing with exceptions in those cases where true exception mechanics are unavailable or undesirable (for example in the DII). They may be set and inspected using the **exception** attribute.

As with normal OMG IDL attributes, the **exception** attribute is mapped into a pair of C++ functions used to set and get the exception. The semantics of the **set** and **get** functions, however, are somewhat different than those for normal OMG IDL attributes. The **set** C++ function assumes ownership of the **Exception** pointer passed to it. The **Environment** will eventually call **delete** on this pointer, so the **Exception** it points to must be dynamically allocated by the caller. The **get** function returns a pointer to the **Exception**, just as an attribute for a variable-length struct would, but the pointer refers to memory owned by the **Environment**. Once the **Environment** is destroyed, the pointer is no longer valid. The caller must not call **delete** on the **Exception** pointer returned by the **get** function. The **Environment** is responsible for deallocating any **Exception** it holds when it is itself destroyed. If the **Environment** holds no exception, the **get** function returns a null pointer.

The **clear()** function causes the **Environment** to **delete** any **Exception** it is holding. It is not an error to call **clear()** on an **Environment** holding no exception. Passing a null pointer to the **set** exception function is equivalent to calling **clear()**. If an **Environment** contains exception information, the caller is responsible for calling **clear()** on it before passing it to an operation.

20.25.1 Environment Interface

```
// IDL
pseudo interface Environment
{    attribute exception exception;

     void clear();
};
```

20.25.2 Environment C++ Class

```
// C++
class Environment
{
  public:
void exception(Exception*);
Exception *exception() const;
void clear();
};
```

20.25.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Defines an interface rather than a struct.
- Supports an attribute allowing operations on exception values as a whole rather than on major numbers and/or identification strings.
- Supports a **`clear()`** function that is used to destroy any **`Exception`** the **`Environment`** may be holding.
- Supports a default constructor that initializes it to hold no exception information.

20.25.4 Memory Management

`Environment` has the following special memory management rules:

- The **`void exception(Exception*)`** member function adopts the **`Exception*`** given to it.
- Ownership of the return value of the **`Exception *exception()`** member function is maintained by the **`Environment`**; this return value must not be freed by the caller.

20.26 NamedValue

NamedValue is used only as an element of **NVList**, especially in the DII. **NamedValue** maintains an (optional) name, an **any** value, and labelling flags. Legal flag values are **ARG_IN**, **ARG_OUT**, and **ARG_INOUT**.

The value in a **NamedValue** may be manipulated via standard operations on **any**.

20.26.1 NamedValue Interface

```
// IDL
pseudo interface NamedValue
{
    readonly attribute Identifier name;
    readonly attribute any value;
    readonly attribute Flags flags;
};
```

20.26.2 NamedValue C++ Class

```
// C++
class NamedValue
{
  public:
const char *name() const;
Any *value() const;
Flags flags() const;
};
```

20.26.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Defines an interface rather than a struct.
- Provides no analog of the **len** field.

20.26.4 Memory Management

`NamedValue` has the following special memory management rules:

- Ownership of the return values of the **`name()`** and **`value()`** functions is maintained by the **`NamedValue`**; these return values must not be freed by the caller.

20.27 NVList

NVList is a list of **NamedValue**s. A new **NVList** is constructed using the **ORB::create_list** operation (see “ORB” on page 20-83). New **NamedValue**s may be constructed as part of an **NVList**, in any of three ways:

- **add**—creates an unnamed value, initializing only the flags.
  - **add_item**—initializes name and flags.
  - **add_value**—initializes name, value, and flags.
  - **add_item_consume**—initializes name and flags, taking over memory management responsibilities for the **`char*`** name parameter.

- **add_value_consume**—initializes name, value, and flags, taking over memory management responsibilities for both the **char*** name parameter and the **Any*** value parameter. Each of these operations returns the new item.

Elements may be accessed and deleted via zero-based indexing. The **add**, **add_item**, **add_value**, **add_item_consume**, and **add_value_consume** functions lengthen the **NVList** to hold the new element each time they are called. The **item** function can be used to access existing elements.

20.27.1 NVList Interface

```
// IDL
pseudo interface NVList
{
    readonly attribute unsigned long count;
    NamedValue add(in Flags flags);
    NamedValue add_item(in Identifier item_name, in Flags flags);
    NamedValue add_value(
                        in Identifier item_name,
                        in any val,
                        in Flags flags
    );
    NamedValue item(in unsigned long index) raises(Bounds);

    Status remove(in unsigned long index) raises(Bounds);
};
```

20.27.2 NVList C++ Class

```
// C++
class NVList
{
  public:
ULong count() const;
NamedValue_ptr add(Flags);
NamedValue_ptr add_item(const char*, Flags);
NamedValue_ptr add_value(
const char*,
const Any&,
Flags
);
NamedValue_ptr add_item_consume(
char*,
Flags
);
```

```
NamedValue_ptr add_value_consume(
char*,
Any *,
Flags
);
NamedValue_ptr item(ULong);
Status remove(ULong);
};
```

20.27.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Defines an interface rather than a typedef
- Provides different signatures for operations that add items in order to avoid representation dependencies
- Provides indexed access methods

20.27.4 Memory Management

`NVList` has the following special memory management rules:

- Ownership of the return values of the **`add`**, **`add_item`**, **`add_value`**, **`add_item_consume`**, **`add_value_consume`**, and **`item`** functions is maintained by the **`NVList`**; these return values must not be freed by the caller.
- The **`char*`** parameters to the **`add_item_consume`** and **`add_value_consume`** functions and the **`Any*`** parameter to the **`add_value_consume`** function are consumed by the **`NVList`**. The caller may not access these data after they have been passed to these functions because the **`NVList`** may copy them and destroy the originals immediately. The caller should use the **`NamedValue::value()`** operation in order to modify the **`value`** attribute of the underlying **`NamedValue`**, if desired.
- The **`remove`** function also calls **`CORBA::release`** on the removed **`NamedValue`**.

20.28 Request

Request provides the primary support for DII. A new request on a particular target object may be constructed using the short version of the request creation operation shown in “Object” on page 20-86:

```
// C++
Request_ptr Object::_request(Identifier operation);
```

Arguments and contexts may be added after construction via the corresponding attributes in the **Request** interface. Results, output arguments, and exceptions are similarly obtained after invocation. The following C++ code illustrates usage:

```
// C++
Request_ptr req = anObj->_request("anOp");
*(req->arguments()->add(ARG_IN)->value()) <<= anArg;
// ...
req->invoke();
if (req->env()->exception() == NULL) {
*(req->result()->value()) >>= aResult;
}
```

While this example shows the semantics of the attribute-based accessor functions, the following example shows that it is much easier and preferable to use the equivalent argument manipulation helper functions:

```
// C++
Request_ptr req = anObj->_request("anOp");
req->add_in_arg() <<= anArg;
// ...
req->invoke();
if (req->env()->exception() == NULL) {
req->return_value() >>= aResult;
}
```

Alternatively, requests can be constructed using one of the long forms of the creation operation shown in the Object interface in “Object” on page 20-86:

```
// C++
Status Object::_create_request(
Context_ptr ctx,
const char *operation,
NVList_ptr arg_list,
NamedValue_ptr result,
Request_out request,
Flags req_flags
);
Status Object::_create_request(
Context_ptr ctx,
const char *operation,
NVList_ptr arg_list,
NamedValue_ptr result,
ExceptionList_ptr,
ContextList_ptr,
Request_out request,
Flags req_flags
);
```

Usage is the same as for the short form except that all invocation parameters are established on construction. Note that the **OUT_LIST_MEMORY** and **IN_COPY_VALUE** flags can be set as flags in the **`req_flags`** parameter, but they are meaningless and thus ignored because argument insertion and extraction are done via the **`Any`** type.

Request also allows the application to supply all information necessary for it to be invoked without requiring the ORB to utilize the Interface Repository. In order to deliver a request and return the response, the ORB requires:

- a target object reference
- an operation name
- a list of arguments (optional)
- a place to put the result (optional)
- a place to put any returned exceptions

- a **Context** (optional)
  - a list of the user-defined exceptions that can be thrown (optional)
- a list of **Context** strings that must be sent with the operation (optional)

Since the **Object::create_request** operation allows all of these except the last two to be specified, an ORB may have to utilize the Interface Repository in order to discover them. Some applications, however, may not want the ORB performing potentially expensive Interface Repository lookups during a request invocation, so two new serverless objects have been added to allow the application to specify this information instead:

- **ExceptionList**: allows an application to provide a list of **TypeCode**s for all user-defined exceptions that may result when the **Request** is invoke.
- **ContextList**: allows an application to provide a list of **Context** strings that must be supplied with the **Request** invocation.

The **ContextList** differs from the **Context** in that the former supplies only the context strings whose values are to be looked up and sent with the request invocation (if applicable), while the latter is where those values are obtained.

The IDL descriptions for **ExceptionList**, **ContextList**, and **Request** are shown below.

20.28.1 Request Interface

```
// IDL
pseudo interface ExceptionList
{
    readonly attribute unsigned long count;
    void add(in TypeCode exc);
    TypeCode item(in unsigned long index) raises(Bounds);
    Status remove(in unsigned long index) raises(Bounds);
};

pseudo interface ContextList
{
    readonly attribute unsigned long count;
    void add(in string ctxt);
```

```
    string item(in unsigned long index) raises(Bounds);
    Status remove(in unsigned long index) raises(Bounds);
};

pseudo interface Request
{
    readonly attribute Object target;
    readonly attribute Identifier operation;
    readonly attribute NVList arguments;
    readonly attribute NamedValue result;
    readonly attribute Environment env;
    readonly attribute ExceptionList exceptions;
    readonly attribute ContextList contexts;

    attribute context ctx;

    Status invoke();
    Status send_oneway();
    Status send_deferred();
    Status get_response();
    boolean poll_response();
};
```

20.28.2 Request C++ Class

```
// C++
class ExceptionList
{
  public:
ULong count();
void add(TypeCode_ptr tc);
void add_consume(TypeCode_ptr tc);
TypeCode_ptr item(ULong index);
Status remove(ULong index);
};

class ContextList
{
  public:
ULong count();
void add(const char* ctxt);
void add_consume(char* ctxt);
const char* item(ULong index);
Status remove(ULong index);
};
```

```
class Request
{
  public:
Object_ptr target() const;
const char *operation() const;
NVList_ptr arguments();
NamedValue_ptr result();
Environment_ptr env();
ExceptionList_ptr exceptions();
ContextList_ptr contexts();

void ctx(Context_ptr);
Context_ptr ctx() const;

// argument manipulation helper functions
Any &add_in_arg();
Any &add_in_arg(const char* name);
Any &add_inout_arg();
Any &add_inout_arg(const char* name);
Any &add_out_arg();
Any &add_out_arg(const char* name);
void set_return_type(TypeCode_ptr tc);
Any &return_value();
Status invoke();
Status send_oneway();
Status send_deferred();
Status get_response();
Boolean poll_response();
};
```

20.28.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Replacement of **add_argument**, and so forth, with attribute-based accessors.
- Use of **env** attribute to access exceptions raised in DII calls.
- The **invoke** operation does not take a flag argument, since there are no flag values that are listed as legal in *CORBA*.
- The **send_oneway** and **send_deferred** operations replace the single **send** operation with flag values, in order to clarify usage.
- The **get_response** operation does not take a flag argument, and an operation **poll_response** is defined to immediately return with an indication of whether the operation has completed. This was done because in *CORBA*, if the type **Status** is **void**, the version with **RESP_NO_WAIT** does not enable the caller to determine if the operation has completed.
- The **add_*_arg**, **set_return_type**, and **return_value** member functions are added as shortcuts for using the attribute-based accessors.

20.28.4 Memory Management

Request has the following special memory management rules:

- Ownership of the return values of the **target**, **operation**, **arguments**, **result**, **env**, **exceptions**, **contexts**, and **ctx** functions is maintained by the **Request**; these return values must not be freed by the caller.

ExceptionList has the following special memory management rules:

- The **add_consume** function consumes its **TypeCode_ptr** argument. The caller may not access the object referred to by the **TypeCode_ptr** after it has been passed in because the **add_consume** function may copy it and release the original immediately.
- Ownership of the return value of the **item** function is maintained by the **ExceptionList**; this return value must not be released by the caller.

ContextList has the following special memory management rules:

- The **add_consume** function consumes its **char*** argument. The caller may not access the memory referred to by the **char*** after it has been passed in because the **add_consume** function may copy it and free the original immediately.
- Ownership of the return value of the **item** function is maintained by the **ContextList**; this return value must not be released by the caller.

20.29 Context

A **Context** supplies optional context information associated with a method invocation.

20.29.1 Context Interface

```
// IDL
pseudo interface Context
{
    readonly attribute Identifier context_name;
    readonly attribute context parent;

    Status create_child(in Identifier child_ctx_name, out Context child_ctx);

    Status set_one_value(in Identifier propname, in any propvalue);
    Status set_values(in NVList values);
    Status delete_values(in Identifier propname);
    Status get_values(
                    in Identifier start_scope,
                    in Flags op_flags,
                    in Identifier pattern,
                    out NVList values
    );
};
```

20.29.2 Context C++ Class

```
// C++
class Context
{
  public:
const char *context_name() const;
Context_ptr parent() const;

Status create_child(const char *, Context_out);

Status set_one_value(const char *, const Any &);
Status set_values(NVList_ptr);
Status delete_values(const char *);
Status get_values(
const char*,
Flags,
const char*,
NVList_out
);
};
```

20.29.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Introduction of attributes for context name and parent.
- The signatures for values are uniformly set to **any**.
- In the C mapping, **`set_one_value`** used strings, while others used **NamedValue**s containing **any**. Even though implementations need only support strings as values, the signatures now uniformly allow alternatives.
- The **release** operation frees child contexts.

20.29.4 Memory Management

`Context` has the following special memory management rules:

- Ownership of the return values of the **`context_name`** and **`parent`** functions is maintained by the **`Context`**; these return values must not be freed by the caller.

20.30 TypeCode

A **TypeCode** represents OMG IDL type information.

No constructors for **TypeCode**s are defined. However, in addition to the mapped interface, for each basic and defined OMG IDL type, an implementation provides access to a **TypeCode** pseudo object reference (**`TypeCode_ptr`**) of the form **`_tc_<type>`** that may be used to set types in **Any**, as arguments for **equal**, and so

on. In the names of these **TypeCode** reference constants, **<type>** refer to the local name of the type within its defining scope. Each C++ **_tc_<type>** constant must be defined at the same scoping level as its matching type.

In all C++ **TypeCode** pseudo object reference constants, the prefix "_tc_" should be used instead of the "TC_" prefix prescribed in "TypeCode" on page 20-81. This is to avoid name clashes for CORBA applications that simultaneously use both the C and C++ mappings.

Like all other serverless objects, the C++ mapping for **TypeCode** provides a **_nil()** operation that returns a nil object reference for a **TypeCode**. This operation can be used to initialize **TypeCode** references embedded within constructed types. However, a nil **TypeCode** reference may never be passed as an argument to an operation, since **TypeCode**s are effectively passed as values, not as object references.

20.30.1 TypeCode Interface

The **TypeCode** IDL interface is fully defined in "The TypeCode Interface" on page 8-36 and is thus is not duplicated here.

20.30.2 TypeCode C++ Class

```
// C++
class TypeCode
{
  public:
class Bounds { ... };
class BadKind { ... };

Boolean equal(TypeCode_ptr) const;
TCKind kind() const;

const char* id() const;
const char* name() const;

ULong member_count() const;
const char* member_name(ULong index) const;

TypeCode_ptr member_type(ULong index) const;
```

```
Any *member_label(ULong index) const;
TypeCode_ptr discriminator_type() const;
Long default_index() const;

ULong length() const;

TypeCode_ptr content_type() const;

UShort fixed_digits() const;
Short fixed_scale() const;

Long param_count() const;
Any *parameter(Long) const;
};
```

20.30.3 Differences from C-PIDL

For C++, use of prefix "_tc_" instead of "TC_" for constants.

20.30.4 Memory Management

`TypeCode` has the following special memory management rules:

- Ownership of the return values of the **`id`**, **`name`**, and **`member_name`** functions is maintained by the **`TypeCode`**; these return values must not be freed by the caller.

20.31 ORB

An **ORB** is the programmer interface to the Object Request Broker.

20.31.1 ORB Interface

```
// IDL
pseudo interface ORB
{
    typedef sequence<Request> RequestSeq;
    string object_to_string(in Object obj);
    Object string_to_object(in string str);
    Status create_list(in long count, out NVList new_list);
    Status create_operation_list(in OperationDef oper, out NVList new_list);
    Status create_named_value(out NamedValue nmval);
    Status create_exception_list(out ExceptionList exclist);
    Status create_context_list(out ContextList ctxtlist);

    Status get_default_context(out Context ctx);
    Status create_environment(out Environment new_env);

    Status send_multiple_requests_oneway(in RequestSeq req);
```

```
    Status send_multiple_requests_deferred(in RequestSeq req);
    boolean poll_next_response();

    Status get_next_response(out Request req);
};
    Boolean work_pending();
    void perform_work();
    void shutdown(in Boolean wait_for_completion);
    void run();

    Boolean get_service_information (
                in ServiceType service_type,
                out ServiceInformation service_information
            );
};
```

20.31.2 ORB C++ Class

```
// C++
class ORB
{
  public:
class RequestSeq {...};
char *object_to_string(Object_ptr);
Object_ptr string_to_object(const char *);
Status create_list(Long, NVList_out);
Status create_operation_list(
OperationDef_ptr,
NVList_out
);
Status create_named_value(NamedValue_out);
Status create_exception_list(ExceptionList_out);
Status create_context_list(ContextList_out);

Status get_default_context(Context_out);
Status create_environment(Environment_out);

Status send_multiple_requests_oneway(
const RequestSeq&
);
Status send_multiple_requests_deferred(
const RequestSeq &
);
Boolean poll_next_response();
Status get_next_response(Request_out);

Boolean work_pending();
void perform_work();
void shutdown(Boolean wait_for_completion);
void run();
```

```
Boolean get_service_information(
ServiceType svc_type,
ServiceInformation_out svc_info
);
};
```

20.31.3 Differences from C-PIDL

- Added **create_environment**. Unlike the struct version, **Environment** requires a construction operation. (Since this is overly constraining for implementations that do not support real C++ exceptions, these implementations may allow **Environment** to be declared on the stack. See "Without Exception Handling" on page 20-116 for details.)
- Assigned multiple request support to ORB, made usage symmetrical with that in **Request**, and used a sequence type rather than otherwise illegal unbounded arrays in signatures.
- Added **create_named_value**, which is required for creating **NamedValue** objects to be used as return value parameters for the **Object::create_request** operation.
- Added **create_exception_list** and **create_context_list** (see "Request" on page 20-75 for more details).

20.31.4 Mapping of ORB Initialization Operations

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++
namespace CORBA {
typedef char* ORBid;
static ORB_ptr ORB_init(
int& argc,
 char** argv,
const char* orb_identifier = ""
);
}
```

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**&`**.

20.32 Object

The rules in this section apply to OMG IDL interface **Object**, the base of the OMG IDL interface hierarchy. Interface **Object** defines a normal CORBA object, not a pseudo object. However, it is included here because it references other pseudo objects.

20.32.1 Object Interface

```
// IDL
interface Object
{
    boolean is_nil();
    Object duplicate();
    void release();
    ImplementationDef get_implementation();
    InterfaceDef get_interface();
    boolean is_a(in string logical_type_id);
    boolean non_existent();
    boolean is_equivalent(in Object other_object);
    unsigned long hash(in unsigned long maximum);
    Status create_request(
                in Context ctx,
                in Identifier operation,
                in NVList arg_list,
                in NamedValue result,
                out Request request,
                in Flags req_flags
    );
    Status create_request2(
                in Context ctx,
                in Identifier operation,
                in NVList arg_list,
                in NamedValue result,
                in ExceptionList exclist,
                in ContextList ctxtlist,
                out Request request,
                in Flags req_flags
    );
    Policy_ptr get_policy(in PolicyType policy_type);
    DomainManagerList get_domain_managers();
    Object set_policy_override(in PolicyList policies,
                               in SetOverrideType set_or_add);
};
```

20.32.2 Object C++ Class

In addition to other rules, all operation names in interface **Object** have leading underscores in the mapped C++ class. Also, the mapping for **create_request** is split into three forms, corresponding to the usage styles described in “create_request” on page 5-5 and in“Request” on page 20-75 of this document. The **is_nil** and **release** functions are provided in the **CORBA** namespace, as described in “Object Reference Operations” on page 20-8.

```
// C++
class Object
{
  public:
static Object_ptr _duplicate(Object_ptr obj);
static Object_ptr _nil();
ImplementationDef_ptr _get_implementation();
InterfaceDef_ptr _get_interface();
Boolean _is_a(const char* logical_type_id);
Boolean _non_existent();
Boolean _is_equivalent(Object_ptr other_object);
ULong _hash(ULong maximum);
Status _create_request(
Context_ptr ctx,
const char *operation,
NVList_ptr arg_list,
NamedValue_ptr result,
Request_out request,
Flags req_flags
);
Status _create_request(
Context_ptr ctx,
const char *operation,
NVList_ptr arg_list,
NamedValue_ptr result,
ExceptionList_ptr,
ContextList_ptr,
Request_out request,
Flags req_flags
);
Request_ptr _request(const char* operation);
Policy_ptr _get_policy(PolicyType policy_type);
    DomainManagerList* _get_domain_managers();
    Object_ptr _set_policy_override(
                      const PolicyList&,
                      SetOverrideType
};
```

20.33 Server-Side Mapping

Server-side mapping refers to the portability constraints for an object implementation written in C++. The term *server* is not meant to restrict implementations to situations in which method invocations cross address space or machine boundaries. This mapping addresses any implementation of an OMG IDL interface.

20.34 Implementing Interfaces

To define an implementation in C++, one defines a C++ class with any valid C++ name. For each operation in the interface, the class defines a non-static member function with the mapped name of the operation (the mapped name is the same as the OMG IDL identifier except when the identifier is a C++ keyword, in which case the string "_cxx_" is prepended to the identifier, as noted in "Preliminary Information" on page 20-3). Note that the ORB implementation may allow one implementation class to derive from another, so the statement "the class defines a member function" does not mean the class must explicitly define the member function—it could inherit the function.

The mapping specifies two alternative relationships between the application-supplied implementation class and the generated class or classes for the interface. Specifically, the mapping requires support for both *inheritance-based* relationships and *delegation-based* relationships. CORBA-compliant ORB implementations are required to provide both of these alternatives. Conforming applications may use either or both of these alternatives.

20.34.1 Mapping of PortableServer::Servant

The **PortableServer** module for the Portable Object Adapter (POA) defines the native **Servant** type. The C++ mapping for **Servant** is as follows:

```
// C++
namespace PortableServer
{
class ServantBase
{
  public:
virtual ~ServantBase();

ServantBase& operator=(const ServantBase&);

virtual POA_ptr _default_POA();

  protected:
ServantBase();
ServantBase(const ServantBase&);
// ...all other constructors...
};
typedef ServantBase* Servant;
}
```

The **ServantBase** destructor is public and virtual to ensure that skeleton classes derived from it can be properly destroyed. The default constructor, along with other implementation-specific constructors, must be protected so that instances of **ServantBase** cannot be created except as sub-objects of instances of derived classes. A default constructor (a constructor that either takes no arguments or takes only arguments with default values) must be provided so that derived servants can be

constructed portably. Both copy construction and a public default assignment operator must be supported so that application-specific servants can be copied if necessary. Note that copying a servant that is already registered with the object adapter, either by assignment or by construction, does not mean that the target of the assignment or copy is also registered with the object adapter. Similarly, assigning to a **`ServantBase`** or a class derived from it that is already registered with the object adapter does not in any way change its registration.

The only operation supplied by the **`ServantBase`** class is the **`_default_POA()`** function. The default implementation of this function, provided by **`ServantBase`**, returns an object reference to the root POA of the default ORB in this process — the same as the return value of an invocation of **`ORB::resolve_initial_references("RootPOA")`** on the default ORB. Classes derived from **`ServantBase`** can override this definition to return the POA of their choice, if desired.

20.34.2 Skeleton Operations

All skeleton classes provide a **`_this()`** member function. This member function has three purposes:

1. Within the context of a request invocation on the target object represented by the servant, it allows the servant to obtain the object reference for the target CORBA object it is incarnating for that request. This is true even if the servant incarnates multiple CORBA objects. In this context, **`_this()`** can be called regardless of the policies the dispatching POA was created with.

2. Outside the context of a request invocation on the target object represented by the servant, it allows a servant to be implicitly activated if its POA allows implicit activation. This requires the activating POA to have been created with the **IMPLICIT_ACTIVATION** policy. If the POA was not created with the **IMPLICIT_ACTIVATION** policy, the **PortableServer::WrongPolicy** exception is thrown.

3. Outside the context of a request invocation on the target object represented by the servant, it will return the object reference for a servant that has already been activated, as long as the servant is not incarnating multiple CORBA objects. This requires the POA with which the servant is activated to have been created with the UNIQUE_ID and RETAIN policies. If the POA was created with the **MULTIPLE_ID** or **NON_RETAIN** policies, the **PortableServer::WrongPolicy** exception is thrown.

For example, using interface **A**

```
// IDL
interface A
{
    short op1();
    void op2(in long val);
};
```

The return value of **_this()** is a typed object reference for the interface type corresponding to the skeleton class. For example, the **_this()** function for the skeleton for interface **A** would be defined as follows:

```
// C++
class POA_A : public virtual ServantBase
{
  public:
A_ptr _this();
...
};
```

The **_this()** function follows the normal C++ mapping rules for returned object references, so the caller assumes ownership of the returned object reference and must eventually call **CORBA::release()** on it.

The **_this()** function can be virtual if the C++ environment supports covariant return types, otherwise the function must be non-virtual so the return type can be correctly specified without compiler errors. Applications use **_this()** the same way regardless of which of these implementation approaches is taken.

Assuming **A_impl** is a class derived from **POA_A** that implements the **A** interface, and assuming that the servant's POA was created with the appropriate policies, a servant of type **A_impl** can be created and implicitly activated as follows:

```
// C++
A_impl my_a;
A_var a = my_a._this();
```

20.34.3 Inheritance-Based Interface Implementation

Implementation classes can be derived from a generated base class based on the OMG IDL interface definition. The generated base classes are known as *skeleton classes*, and the derived classes are known as *implementation classes*. Each operation of the interface has a corresponding virtual member function declared in the skeleton class. The signature of the member function is identical to that of the generated client stub class. The implementation class provides implementations for these member functions. The object adapter typically invokes the methods via calls to the virtual functions of the skeleton class.

Assume that IDL interface **A** is defined as follows:

```
// IDL
interface A
{
    short op1();
    void op2(in long val);
};
```

For IDL interface **A** as shown above, the IDL compiler generates an interface class **A**. This class contains the C++ definitions for the typedefs, constants, exceptions, attributes, and operations in the OMG IDL interface. It has a form similar to the following:

```
// C++
class A : public virtual CORBA::Object
{
  public:
virtual Short op1() = 0;
virtual void op2(Long val) = 0;
...
};
```

Some ORB implementations might not use public virtual inheritance from **`CORBA::Object`**, and might not make the operations pure virtual, but the signatures of the operations will be the same.

On the server side, a skeleton class is generated. This class is partially opaque to the programmer, though it will contain a member function corresponding to each operation in the interface. For the POA, the name of the skeleton class is formed by prepending the string "POA_" to the fully-scoped name of the corresponding interface, and the class is either directly or indirectly derived from the servant base class **`PortableServer::ServantBase`**. The **`PortableServer::ServantBase`** class must be a virtual base class of the skeleton to allow portable implementations to multiply inherit from both skeleton classes and implementation classes for other base interfaces without error or ambiguity.

The skeleton class for interface **A** shown above would appear as follows:

```
// C++
class POA_A : public virtual PortableServer::ServantBase
{
  public:
// ...server-side implementation-specific detail
// goes here...
virtual Short op1() throw(SystemException) = 0;
virtual void op2(Long val) throw(SystemException) = 0;
...
};
```

If interface **A** were defined within a module rather than at global scope, *e.g.*, **Mod::A**, the name of its skeleton class would be **`POA_Mod::A`**. This helps to separate server-side skeleton declarations and definitions from C++ code generated for the client.

To implement this interface using inheritance, a programmer must derive from this skeleton class and implement each of the operations in the OMG IDL interface. An implementation class declaration for interface **A** would take the form:

```
// C++
class A_impl : public POA_A
{
  public:
Short op1() throw(CORBA::SystemException);
void op2(Long val) throw(CORBA::SystemException);
...
};
```

Note that the presence of the **_this()** function implies that C++ servants must only be derived directly from a single skeleton class. Direct derivation from multiple skeleton classes could result in ambiguity errors due to multiple definitions of **_this()**. This should not be a limitation, since CORBA objects have only a single most-derived interface. Servants that are intended to support multiple interface types can utilize the delegation-based interface implementation approach, described below in "Delegation-Based Interface Implementation", or can be registered as DSI-based servants, as described in "Mapping of Dynamic Skeleton Interface to C++" on page 20-99.

20.34.4 Delegation-Based Interface Implementation

Inheritance is not always the best solution for implementing servants. Using inheritance from the OMG IDL–generated classes forces a C++ inheritance hierarchy into the application. Sometimes, the overhead of such inheritance is too high, or it may be impossible to compile correctly due to defects in the C++ compiler. For example, implementing objects using existing legacy code might be impossible if inheritance from some global class were required, due to the invasive nature of the inheritance.

In some cases delegation can be used to solve this problem. Rather than inheriting from a skeleton class, the implementation can be coded as required for the application, and a wrapper object will delegate upcalls to that implementation. This section describes how this can be achieved in a type-safe manner using C++ templates.

For the examples in this section, the OMG IDL interface from "Inheritance-Based Interface Implementation" on page 20-91 will again be used:

```
// IDL
interface A
{
    short op1();
    void op2(in long val);
};
```

In addition to generating a skeleton class, the IDL compiler generates a delegating class called a *tie*. This class is partially opaque to the application programmer, though like the skeleton, it provides a method corresponding to each OMG IDL operation. The name of the generated tie class is the same as the generated skeleton class with the addition that the string "_tie" is appended to the end of the name. For example:

```
// C++
template<class T>
class POA_A_tie : public POA_A
{
  public:
...
};
```

An instance of this template class performs the task of delegation. When the template is instantiated with a class type that provides the operations of **A**, then the **POA_A_tie** class will delegate all operations to an instance of that implementation class. A reference or pointer to the actual implementation object is passed to the appropriate tie constructor when an instance of the tie class is created. When a request is invoked on it, the tie servant will just delegate the request by calling the corresponding method in the implementation object.

```
// C++
template <class T>
class POA_A_tie : public POA_A
{
  public:
POA_A_tie(T& t)
: _ptr(&t), _poa(POA::_nil()), _rel(0) {}
POA_A_tie(T& t, POA_ptr poa)
: _ptr(&t),
_poa(POA::_duplicate(poa)), _rel(0) {}
POA_A_tie(T* tp, Boolean release = 1)
: _ptr(tp), _poa(POA::_nil()), _rel(release) {}
POA_A_tie(T* tp, POA_ptr poa,
Boolean release = 1)
: _ptr(tp), _poa(POA::_duplicate(poa)),
_rel(release) {}
~POA_A_tie()
{
CORBA::release(_poa);
if (_rel) delete _ptr;
}
```

```
// tie-specific functions
T* _tied_object() { return _ptr; }
void _tied_object(T& obj)
{
if (_rel) delete _ptr;
_ptr = &obj;
_rel = 0;
}
void _tied_object(T* obj, Boolean release = 1)
{
if (_rel) delete _ptr;
_ptr = obj;
_rel = release;
}
Boolean _is_owner() { return _rel; }
void _is_owner(Boolean b) { _rel = b; }

// IDL operations
Short op1() throw(SystemException)
{
return _ptr->op1();
}
void op2(Long val) throw(SystemException)
{
_ptr->op2(val);
}

// override ServantBase operations
POA_ptr _default_POA()
{
if (!CORBA::is_nil(_poa)) {
return _poa;
} else {
// return root POA
}
}

  private:
T* _ptr;
POA_ptr _poa;
Boolean _rel;

// copy and assignment not allowed
POA_A_tie(const POA_A_tie&);
void operator=(const POA_A_tie&);
};
```

It is important to note that the tie example shown above contains sample implementations for all of the required functions. A conforming implementation is free to implement these operations as it sees fit, as long as they conform to the semantics in the paragraphs described below. A conforming implementation is also allowed to

include additional implementation-specific functions if it wishes.

The **T&** constructors cause the tie servant to delegate all calls to the C++ object bound to reference **t**. Ownership for the object referred to by **t** does not become the responsibility of the tie servant.

The **T*** constructors cause the tie servant to delegate all calls to the C++ object pointed to by **tp**. The **release** parameter dictates whether the tie takes on ownership of the C++ object pointed to by **tp**; if **release** is **TRUE**, the tie adopts the C++ object, otherwise it does not. If the tie adopts the C++ object being delegated to, it will **delete** it when its own destructor is invoked, as shown above in the **~POA_A_tie()** destructor.

The **_tied_object()** accessor function allows callers to access the C++ object being delegated to. If the tie was constructed to take ownership of the C++ object (**release** was **TRUE** in the **T*** constructor), the caller of **_tied_object()** should never **delete** the return value.

The first **_tied_object()** modifier function calls **delete** on the current tied object if the tie's release flag is **TRUE**, and then points to the new tie object passed in. The tie's release flag is set to **FALSE**. The second **_tied_object()** modifier function does the same, except that the final state of the tie's release flag is determined by the value of the **release** argument.

The **_is_owner()** accessor function returns **TRUE** if the tie owns the C++ object it is delegating to, or **FALSE** if it does not. The **_is_owner()** modifier function allows the state of the tie's release flag to be changed. This is useful for ensuring that memory leaks do not occur when transferring ownership of tied objects from one tie to another, or when changing the tied object a tie delegates to.

For delegation-based implementations it is important to note that the servant is the tie object, not the C++ object being delegated to by the tie object. This means that the tie servant is used as the argument to those POA operations that require a **Servant** argument. This also means that any operations that the POA calls on the servant, such as **ServantBase::_default_POA()**, are provided by the tie servant, as shown by the example above. The value returned by **_default_POA()** is supplied to the tie constructor.

It is also important to note that by default, a delegation-based implementation (the "tied" C++ instance) has no access to the **_this()** function, which is available only on the tie. One way for this access to be provided is by informing the delegation object of its associated tie object. This way, the tie holds a pointer to the delegation object, and vice-versa. However, this approach only works if the tie and the delegation object have a one-to-one relationship. For a delegation object tied into multiple tie objects, the object reference by which it was invoked can be obtained within the context of a request invocation by calling **PortableServer::Current::get_object_id()**, passing its return value to **PortableServer::POA::id_to_reference()**, and then narrowing the returned object reference appropriately.

In the tie class shown above, all the operations are shown as being inline. In practice, it is likely that they will be defined out of line, especially for those functions that override inherited virtual functions. Either approach is allowed by conforming implementations.

The use of templates for tie classes allows the application developer to provide specializations for some or all of the template's member functions for a given instantiation of the template. This allows the application to control how the tied object is invoked. For example, the **`POA_A_tie<T>::op2()`** operation is normally defined as follows:

```
// C++
template<class T>
void
POA_A_tie<T>::op2(Long val) throw(SystemException)
{
_ptr->op2(val);
}
```

This implementation assumes that the tied object supports an **`op2()`** operation with the same signature and the ability to throw CORBA system exceptions. However, if the application wants to use legacy classes for tied object types, it is unlikely they will support these capabilities. In that case, the application can provide its own specialization. For example, if the application already has a class named **`Foo`** that supports a **`log_value()`** function, the tie class **`op2()`** function can be made to call it if the following specialization is provided:

```
// C++
void
POA_A_tie<Foo>::op2(Long val) throw(SystemException)
{
_tied_object()->log_value(val);
}
```

Portable specializations like the one shown above should not access tie class data members directly, since the names of those data members are not standardized.

20.35 Implementing Operations

The signature of an implementation member function is the mapped signature of the OMG IDL operation. Unlike the client side, the server-side mapping requires that the function header include the appropriate exception (**`throw`**) specification. This requirement allows the compiler to detect when an invalid exception is raised, which is necessary in the case of a local C++-to-C++ library call (otherwise the call would have to go through a wrapper that checked for a valid exception). For example:

```
// IDL
interface A
{
    exception B {};
    void f() raises(B);
};

// C++
class MyA : public virtual POA_A
{
  public:
void f() throw(A::B, CORBA::SystemException);
...
};
```

Since all operations and attributes may throw CORBA system exceptions, **`CORBA::SystemException`** must appear in all exception specifications, even when an operation has no **`raises`** clause.

Within a member function, the “this” pointer refers to the implementation object’s data as defined by the class. In addition to accessing the data, a member function may implicitly call another member function defined by the same class. For example:

```
// IDL
interface A
{
    void f();
    void g();
};


// C++
class MyA : public virtual POA_A
{
  public:
void f() throw(SystemException);
void g() throw(SystemException);
  private:
long x_;
};

void
MyA::f() throw(SystemException)
{
this->x_ = 3;
this->g();
}
```

However, when a servant member function is invoked in this manner, it is being called simply as a C++ member function, not as the implementation of an operation on a CORBA object. In such a context, any information available via the **POA_Current** object refers to the CORBA request invocation that performed the C++ member function invocation, not to the member function invocation itself.

20.35.1 Skeleton Derivation From Object

In several existing ORB implementations, each skeleton class derives from the corresponding interface class. For example, for interface **Mod::A**, the skeleton class **POA_Mod::A** is derived from class **Mod::A**. These systems therefore allow an object reference for a servant to be implicitly obtained via normal C++ derived-to-base conversion rules:

```
// C++
MyImplOfA my_a;// declare impl of A
A_ptr a = &my_a;// obtain its object reference
// by C++ derived-to-base
// conversion
```

Such code can be supported by a conforming ORB implementation, but it is not required, and is thus not portable. The equivalent portable code invokes **_this()** on the implementation object in order to implicitly register it if it has not yet been registered, and to get its object reference:

```
// C++
MyImplOfA my_a;// declare impl of A
A_ptr a = my_a._this();// obtain its object
   // reference
```

20.36 Mapping of Dynamic Skeleton Interface to C++

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

This section contains the following information:

- Mapping of the Dynamic Skeleton Interface's **ServerRequest** to C++
- Mapping of the Portable Object Adapter's Dynamic Implementation Routine to C++

20.36.1 Mapping of ServerRequest to C++

The **ServerRequest** pseudo object maps to a C++ class in the **CORBA** namespace which supports the following operations and signatures:

```
// C++
class ServerRequest
{
  public:
const char* operation() const;
void arguments(NVList_ptr& parameters);
Context_ptr ctx();
void set_result(const Any& value);
void set_exception(const Any& value);
};
```

Note that, as with the rest of the C++ mapping, ORB implementations are free to make such operations virtual and modify the inheritance as needed.

All of these operations follow the normal memory management rules for data passed into skeletons by the ORB. That is, the DIR is not allowed to modify or change the string returned by **`operation()`**, **`in`** parameters in the **`NVList`** returned from **`arguments()`**, or the **`Context`** returned by **`ctx()`**. Similarly, data allocated by the DIR and handed to the ORB (the **`NVList`** parameters, the result value, and exception values) are freed by the ORB rather than by the DIR.

20.36.2 Handling Operation Parameters and Results

The **`ServerRequest`** provides parameter values when the DIR invokes the **`arguments()`** operation. The **`NVList`** provided by the DIR to the ORB includes the **`TypeCodes`** and direction **`Flags`** (inside **`NamedValues`**) for all parameters, including **`out`** ones for the operation. This allows the ORB to verify that the correct parameter types have been provided before filling their values in, but does not require it to do so. It also relieves the ORB of all responsibility to consult an Interface Repository, promoting high performance implementations.

The **`NVList`** provided to the ORB then becomes owned by the ORB. It will not be deallocated until after the DIR returns. This allows the DIR to pass the **`out`** values, including the return side of **`inout`** values, to the ORB by modifying the **`NVList`** after **`arguments()`** has been called. Therefore, if the DIR stores the **`NVList_ptr`** into an **`NVList_var`**, it should pass it to the **`arguments()`** function by invoking the **`_retn()`** function on it, in order to force it to release ownership of its internal **`NVList_ptr`** to the ORB.

20.36.3 Mapping of PortableServer Dynamic Implementation Routine

In C++, DSI servants inherit from the standard **`DynamicImplementation`** class. This class inherits from the **`ServantBase`** class and is also defined in the **`PortableServer`** namespace. The Dynamic Skeleton Interface (DSI) is implemented through servants that are members of classes that inherit from dynamic skeleton classes.

```
// C++
namespace PortableServer
{
class DynamicImplementation : public virtual ServantBase
{
  public:
CORBA::Object_ptr _this();
virtual void invoke(
CORBA::ServerRequest_ptr request
) = 0;
virtual CORBA::RepositoryId
_primary_interface(
const ObjectId& oid,
POA_ptr poa
) = 0;
};
}
```

The **_this()** function returns a **CORBA::Object_ptr** for the target object. Unlike **_this()** for static skeletons, its return type is not interface-specific because a DSI servant may very well incarnate multiple CORBA objects of different types. If **DynamicImplementation::_this()** is invoked outside of the context of a request invocation on a target object being served by the DSI servant, it raises the **PortableServer::WrongPolicy** exception.

The **invoke()** method receives requests issued to any CORBA object incarnated by the DSI servant and performs the processing necessary to execute the request.

The **_primary_interface()** method receives an **ObjectId** value and a **POA_ptr** as input parameters and returns a valid **RepositoryId** representing the most-derived interface for that **oid**.

It is expected that the **invoke()** and **_primary_interface()** methods will be only invoked by the POA in the context of serving a CORBA request. Invoking this method in other circumstances may lead to unpredictable results.

20.37 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++
namespace PortableServer
{
char* ObjectId_to_string(const ObjectId&);
wchar_t* ObjectId_to_wstring(const ObjectId&);

ObjectId* string_to_ObjectId(const char*);
ObjectId* wstring_to_ObjectId(const wchar_t*);
}
```

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 throw the **CORBA::BAD_PARAM** exception.

20.38 Mapping for PortableServer::ServantManager

20.38.1 Mapping for 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++
namespace PortableServer
{
class ServantLocator {
...
typedef void* 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`**.

20.38.2 ServantManagers and AdapterActivators

Portable servants that implement the **`PortableServer::AdapterActivator`**, the **`PortableServer::ServantActivator`**, or **`PortableServer::ServantLocator`** interfaces are implemented just like any other servant. They may use either the inheritance-based approach or the tie approach.

20.39 C++ Definitions for CORBA

This section provides a complete set of C++ definitions for the **CORBA** module. The definitions appear within the C++ namespace named **CORBA**.

```
// C++
namespace CORBA { ... }
```

Any implementations shown here are merely sample implementations: they are not the required definitions for these types.

20.39.1 Primitive Types

```
typedef unsigned charBoolean;
typedef unsigned charChar;
typedef wchar_tWChar;
typedef unsigned charOctet;
typedef shortShort;
typedef unsigned shortUShort;
typedef longLong;
typedef ...LongLong;
typedef unsigned longULong;
typedef ...ULongLong;
typedef floatFloat;
typedef doubleDouble;
typedef long doubleLongDouble;
typedef Boolean&Boolean_out;
typedef Char&Char_out;
typedef WChar&WChar_out;
typedef Octet&Octet_out;
typedef Short&Short_out;
typedef UShort&UShort_out;
typedef Long&Long_out;
typedef LongLong&LongLong_out;
typedef ULong&ULong_out;
typedef ULongLong&ULongLong_out;
typedef Float&Float_out;
typedef Double&Double_out;
typedef LongDouble&LongDouble_out;
```

20.39.2 String_var and String_out Class

```
class String_var
{
  public:
String_var();
String_var(char *p);
String_var(const char *p);
String_var(const String_var &s);
~String_var();

String_var &operator=(char *p);
String_var &operator=(const char *p);
String_var &operator=(const String_var &s);

operator char*();
operator const char*() const;
const char* in() const;
char*& inout();
char*& out();
char* _retn();

char &operator[](ULong index);
char operator[](ULong index) const;
};

class String_out
{
  public:
String_out(char*& p);
String_out(String_var& p);
String_out(String_out& s);
String_out& operator=(String_out& s);
String_out& operator=(char* p);
String_out& operator=(const char* p)

operator char*&();
char*& ptr();

  private:
// assignment from String_var disallowed
void operator=(const String_var&);
};
```

20.39.3 WString_var and WString_out

The **`WString_var`** and **`WString_out`** types are identical to **`String_var`** and **`String_out`**, respectively, except that they operate on wide string and wide character types.

20.39.4 Any Class

```
class Any
{
  public:
Any();
Any(const Any&);
Any(TypeCode_ptr tc, void *value,
Boolean release = FALSE);
~Any();

Any &operator=(const Any&);

void operator<<=(Short);
void operator<<=(UShort);
void operator<<=(Long);
void operator<<=(ULong);
void operator<<=(Float);

void operator<<=(Double);
void operator<<=(const Any&);// copying
void operator<<=(Any*);// non-copying
void operator<<=(const char*);

Boolean operator>>=(Short&) const;
Boolean operator>>=(UShort&) const;
Boolean operator>>=(Long&) const;
Boolean operator>>=(ULong&) const;
Boolean operator>>=(Float&) const;
Boolean operator>>=(Double&) const;
Boolean operator>>=(Any*&) const;
Boolean operator>>=(char*&) const;

// special types needed for boolean, octet, char,
// and bounded string insertion
// these are suggested implementations only
struct from_boolean {
from_boolean(Boolean b) : val(b) {}
Boolean val;
};
struct from_octet {
from_octet(Octet o) : val(o) {}
Octet val;
};
struct from_char {
from_char(Char c) : val(c) {}
Char val;
};
struct from_wchar {
from_char(WChar c) : val(c) {}
```

```
WChar val;
};
struct from_string {
from_string(char* s, ULong b,
Boolean nocopy = FALSE) :
val(s), bound(b) {}
char *val;
ULong bound;
};
struct from_wstring {
from_wstring(WChar* s, ULong b,
Boolean nocopy = FALSE) :
val(s), bound(b) {}
WChar *val;
ULong bound;
};

void operator<<=(from_boolean);
void operator<<=(from_char);
void operator<<=(from_wchar);
void operator<<=(from_octet);
void operator<<=(from_string);
void operator<<=(from_wstring);

// special types needed for boolean, octet,
// char extraction
// these are suggested implementations only
struct to_boolean {
to_boolean(Boolean &b) : ref(b) {}
Boolean &ref;
};
struct to_char {
to_char(Char &c) : ref(c) {}
Char &ref;
};
struct to_wchar {
to_wchar(WChar &c) : ref(c) {}
WChar &ref;
};
struct to_octet {
to_octet(Octet &o) : ref(o) {}
Octet &ref;
};
struct to_object {
to_object(Object_ptr &obj) : ref(obj) {}
Object_ptr &ref;
};
struct to_string {
```

```
to_string(char *&s, ULong b) : val(s), bound(b) {}
char *&val;
ULong bound;
};
struct to_wstring {
to_wstring(WChar *&s, ULong b)
: val(s), bound(b) {}
WChar *&val;
ULong bound;
};

Boolean operator>>=(to_boolean) const;
Boolean operator>>=(to_char) const;
Boolean operator>>=(to_wchar) const;
Boolean operator>>=(to_octet) const;
Boolean operator>>=(to_object) const;
Boolean operator>>=(to_string) const;
Boolean operator>>=(to_wstring) const;

void replace(TypeCode_ptr, void *value,
Boolean release = FALSE);

TypeCode_ptr type() const;
const void *value() const;

  private:
// these are hidden and should not be implemented
// so as to catch erroneous attempts to insert
// or extract multiple IDL types mapped to unsigned char
void operator<<=(unsigned char);
Boolean operator>>=(unsigned char&) const;
};
```

20.39.5 Any_var Class

```
class Any_var
{
  public:
Any_var();
Any_var(Any *a);
Any_var(const Any_var &a);
~Any_var();

Any_var &operator=(Any *a);
Any_var &operator=(const Any_var &a);

Any *operator->();

const Any& in() const;
Any& inout();
```

```
Any*& out();
Any* _retn();

// other conversion operators for parameter passing
};
```

20.39.6 Exception Class

```
// C++
class Exception
{
  public:
Exception(const Exception &);
virtual ~Exception();
Exception &operator=(const Exception &);

virtual void _raise() = 0;

  protected:
Exception();
};
```

20.39.7 SystemException Class

```
// C++
enum CompletionStatus { COMPLETED_YES, COMPLETED_NO,
   COMPLETED_MAYBE };
class SystemException : public Exception
{
  public:
SystemException();
SystemException(const SystemException &);
SystemException(ULong minor, CompletionStatus status);
~SystemException();
SystemException &operator=(const SystemException &);

ULong minor() const;
void minor(ULong);

CompletionStatus completed() const;
void completed(CompletionStatus);

static SystemException* _narrow(Exception*);
};
```

20.39.8 UserException Class

```
// C++
class UserException : public Exception
{
```

```
  public:
UserException();
UserException(const UserException &);
~UserException();
UserException &operator=(const UserException &);

static UserException* _narrow(Exception*);
};
```

20.39.9 UnknownUserException Class

```
// C++
class UnknownUserException : public UserException
{
  public:
Any &exception();

static UnknownUserException* _narrow(Exception*);
virtual void raise();
};
```

20.39.10 release and is_nil

```
// C++
namespace CORBA {
void release(Object_ptr);
void release(Environment_ptr);
void release(NamedValue_ptr);
void release(NVList_ptr);
void release(Request_ptr);
void release(Context_ptr);
void release(TypeCode_ptr);
void release(POA_ptr);
void release(ORB_ptr);

Boolean is_nil(Object_ptr);
Boolean is_nil(Environment_ptr);
Boolean is_nil(NamedValue_ptr);
Boolean is_nil(NVList_ptr);
Boolean is_nil(Request_ptr);
Boolean is_nil(Context_ptr);
Boolean is_nil(TypeCode_ptr);
Boolean is_nil(POA_ptr);
Boolean is_nil(ORB_ptr);
...
}
```

20.39.11 Object Class

```
// C++
class Object
{
  public:
static Object_ptr _duplicate(Object_ptr obj);
static Object_ptr _nil();
InterfaceDef_ptr _get_interface();
Boolean _is_a(const char* logical_type_id);
Boolean _non_existent();
Boolean _is_equivalent(Object_ptr other_object);
ULong _hash(ULong maximum);
Status _create_request(
Context_ptr ctx,
const char *operation,
NVList_ptr arg_list,
NamedValue_ptr result,
Request_out request,
Flags req_flags
);
Status _create_request(
Context_ptr ctx,
const char *operation,
NVList_ptr arg_list,
NamedValue_ptr result,
ExceptionList_ptr,
ContextList_ptr,
Request_out request,
Flags req_flags
);
Request_ptr _request(const char* operation);
Policy_ptr _get_policy(PolicyType policy_type);
    DomainManagerList* _get_domain_managers();
    Object_ptr _set_policy_override(
                        const PolicyList& policies,
                        SetOverrideType set_or_add
                    );
};
```

20.39.12 Environment Class

```
// C++
class Environment
{
  public:
void exception(Exception*);
Exception *exception() const;
void clear();

static Environment_ptr _duplicate(Environment_ptr ev);
static Environment_ptr _nil();
};
```

20.39.13 NamedValue Class

```
// C++
class NamedValue
{
  public:
const char *name() const;
Any *value() const;
Flags flags() const;

static NamedValue_ptr _duplicate(NamedValue_ptr nv);
static NamedValue_ptr _nil();
};
```

20.39.14 NVList Class

```
// C++
class NVList
{
  public:
ULong count() const;
NamedValue_ptr add(Flags);
NamedValue_ptr add_item(const char*, Flags);
NamedValue_ptr add_value(const char*, const Any&,
      Flags);
NamedValue_ptr add_item_consume(
char*,
Flags
);
NamedValue_ptr add_value_consume(
char*,
Any *,
Flags
);
NamedValue_ptr item(ULong);
Status remove(ULong);
```

```
static NVList_ptr _duplicate(NVList_ptr nv);
static NVList_ptr _nil();
};
```

20.39.15 ExceptionList Class

```
// C++
class ExceptionList
{
  public:
ULong count();
void add(TypeCode_ptr tc);
void add_consume(TypeCode_ptr tc);
TypeCode_ptr item(ULong index);
Status remove(ULong index);
};
```

20.39.16 ContextList Class

```
class ContextList
{
  public:
ULong count();
void add(const char* ctxt);
void add_consume(char* ctxt);
const char* item(ULong index);
Status remove(ULong index);
};
```

20.39.17 Request Class

```
// C++
class Request
{
  public:
Object_ptr target() const;
const char *operation() const;
NVList_ptr arguments();
NamedValue_ptr result();
Environment_ptr env();
ExceptionList_ptr exceptions();
ContextList_ptr contexts();

void ctx(Context_ptr);
Context_ptr ctx() const;

Any& add_in_arg();
Any& add_in_arg(const char* name);
Any& add_inout_arg();
```

```
Any& add_inout_arg(const char* name);
Any& add_out_arg();
Any& add_out_arg(const char* name);
void set_return_type(TypeCode_ptr tc);
Any& return_value();
Status invoke();
Status send_oneway();
Status send_deferred();
Status get_response();
Boolean poll_response();

static Request_ptr _duplicate(Request_ptr req);
static Request_ptr _nil();
};
```

20.39.18 Context Class

```
// C++
class Context
{
  public:
const char *context_name() const;
Context_ptr parent() const;

Status create_child(const char*, Context_out);

Status set_one_value(const char*, const Any&);
Status set_values(NVList_ptr);
Status delete_values(const char*);
Status get_values(const char*, Flags, const char*,
NVList_out);

static Context_ptr _duplicate(Context_ptr ctx);
static Context_ptr _nil();
};
```

20.39.19 TypeCode Class

```
// C++
class TypeCode
{
  public:
class Bounds { ... };
class BadKind { ... };

TCKind kind() const;
Boolean equal(TypeCode_ptr) const;

const char* id() const;
const char* name() const;
```

```
ULong member_count() const;
const char* member_name(ULong index) const;

TypeCode_ptr member_type(ULong index) const;

Any *member_label(ULong index) const;
TypeCode_ptr discriminator_type() const;
Long default_index() const;

ULong length() const;

TypeCode_ptr content_type() const;

UShort fixed_digits() const;
Short fixed_scale() const;

Long param_count() const;
Any *parameter(Long) const;

static TypeCode_ptr _duplicate(TypeCode_ptr tc);
static TypeCode_ptr _nil();
};
```

20.39.20 ORB Class

```
// C++
class ORB
{
  public:
typedef sequence<Request_ptr> RequestSeq;
char *object_to_string(Object_ptr);
Object_ptr string_to_object(const char*);
Status create_list(Long, NVList_out);
Status create_operation_list(OperationDef_ptr,
      NVList_out);
Status create_named_value(NamedValue_out);
Status create_exception_list(ExceptionList_out);
Status create_context_list(ContextList_out);

Status get_default_context(Context_out);
Status create_environment(Environment_out);

Status send_multiple_requests_oneway(
const RequestSeq&
);
Status send_multiple_requests_deferred(
const RequestSeq&
);
Boolean poll_next_response();
```

```
Status get_next_response(Request_out);

// Obtaining initial object references
typedef char* ObjectId;
class ObjectIdList {...};
class InvalidName {...};
ObjectIdList *list_initial_services();
Object_ptr resolve_initial_references(
const char *identifier
);

Boolean work_pending();
void perform_work();
void shutdown(Boolean wait_for_completion);
void run();

Boolean get_service_information(
ServiceType svc_type,
ServiceInformation_out svc_info
);

static ORB_ptr _duplicate(ORB_ptr orb);
static ORB_ptr _nil();
};
```

20.39.21 ORB Initialization

```
// C++
typedef char* ORBid;
static ORB_ptr ORB_init(
int& argc,
 char** argv,
const char* orb_identifier = ""
);
```

20.39.22 General T_out Types

```
// C++
class T_out
{
  public:
T_out(T*& p) : ptr_(p) { ptr_ = 0; }
T_out(T_var& p) : ptr_(p.ptr_) {
delete ptr_;
ptr_ = 0;
}
T_out(T_out& p) : ptr_(p.ptr_) {}
T_out& operator=(T_out& p) {
ptr_ = p.ptr_;
return *this;
```

```
}
T_out& operator=(T* p) { ptr_ = p; return *this; }

operator T*&() { return ptr_; }
T*& ptr() { return ptr_; }

T* operator->() { return ptr_; }

  private:
T*& ptr_;

// assignment from T_var not allowed
void operator=(const T_var&):
};
```

20.40 Alternative Mappings For C++ Dialects

20.40.1 Without Namespaces

If the target environment does not support the **`namespace`** construct but does support nested classes, then a module should be mapped to a C++ class. If the environment does not support nested classes, then the mapping for modules should be the same as for the CORBA C mapping (concatenating identifiers using an underscore ("_") character as the separator).

Note that module constants map to file-scope constants on systems that support namespaces and class-scope constants on systems that map modules to classes.

20.40.2 Without Exception Handling

For those C++ environments that do not support real C++ exception handling, referred to here as *non-exception handling (non-EH) C++ environments*, an **`Environment`** parameter passed to each operation is used to convey exception information to the caller.

As shown in "Environment" on page 20-71, the **`Environment`** class supports the ability to access and modify the **`Exception`** it holds.

As shown in "Mapping for Exception Types" on page 20-58, both user-defined and system exceptions form an inheritance hierarchy that normally allow types to be caught either by their actual type or by a more general base type. When used in a non-EH C++ environment, the narrowing functions provided by this hierarchy allow for examination and manipulation of exceptions:

```
// IDL
interface A
{
    exception Broken { ... };
```

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

// C++
Environment ev;
A_ptr obj = ...
obj->op(ev);
if (Exception *exc = ev.exception()) {
   if (A::Broken *b = A::Broken::_narrow(exc)) {
      // deal with user exception
   } else {
      // must have been a system exception
      SystemException *se = SystemException::_narrow(exc);
      ...
   }
}
```

"ORB" on page 20-83 specifies that **`Environment`** must be created using **`ORB::create_environment`**, but this is overly constraining for implementations requiring an **`Environment`** to be passed as an argument to each method invocation. For implementations that do not support real C++ exceptions, **`Environment`** may be allocated as a static, automatic, or heap variable. For example, all of the following are legal declarations on a non-EH C++ environment:

```
// C++
Environment global_env;       // global
static Environment static_env;// file static

class MyClass
{
  public:
   ...
  private:
   static Environment class_env;    // class static
};

void func()
{
   Environment auto_env;                    // auto
   Environment *new_env = new Environment;  // heap
   ...
}
```

For ease of use, **`Environment`** parameters are passed by reference in non-EH environments:

```
// IDL
interface A
{
    exception Broken { ... };
    void op() raises(Broken);
```

```
};

// C++
class A ...
{
  public:
   void op(Environment &);
   ...
};
```

For additional ease of use in non-EH environments, **`Environment`** should support copy construction and assignment from other **`Environment`** objects. These additional features are helpful for propagating exceptions from one **`Environment`** to another under non-EH circumstances.

When an exception is "thrown" in a non-EH environment, object implementors and ORB runtimes must ensure that all **`out`** and return pointers are returned to the caller as null pointers. If non-initialized or "garbage" pointer values are returned, client application code could experience runtime errors due to the assignment of bad pointers to **`T_var`** types. When a **`T_var`** goes out of scope, it attempts to **`delete`** the **`T*`** given to it; if this pointer value is garbage, a runtime error will almost certainly occur.

Exceptions in non-EH environments need not support the virtual **`_raise()`** function, since the only useful implementation of it in such an environment would be to abort the program.

20.41 C++ Keywords

Table 20-6 lists all C++ keywords from the 2 December 1996 Working Paper of the ANSI (X3J16) C++ Language Standardization Committee.

Table 20-6 C++ Keywords

and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch
char	class	compl	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	not	not_eq	operator
or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t
while	xor	xor_eq		