Date: July 2012

C++ Language Mapping

Version 1.3

Normative reference: http://www.omg.org/spec/CPP/1.3

OMG Formal document number: formal/2012-07-02

OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, at this URL: *http://www.omg.org/report_issue.htm*.

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from this URL:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/report_issue.htm*.

1 Scope

1.1 General

The CORBA Language Mapping specifications contain language mapping information for several languages. Each language is described in a separate stand-alone volume. This particular specification 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, struct, union, fixed, sequence, array, typedefs, any, valuetype, abstract interface, exception)
- Operations and attributes
- Arguments

1.2 Alignment with CORBA

This language mapping is aligned with CORBA, v3.1.

2 Conformance/Compliance

2.1 General

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.

2.2 C++ Implementation Requirements

This mapping 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, but it does provide work-arounds for C++ compilers that do not support **`namespace`**.

2.3 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.

2.4 Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren't required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in this specification.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to CORBA, v3.1, Part 2: Conformance and Compliance.

As described in the *OMA Guide*, the OMG's Core Object Model consists of a core and components. Likewise, the body of *CORBA* specifications is divided into core and component-like specifications. The *CORBA* specifications are divided into these volumes:

1. The *CORBA/IIOP Specification (Common Object Request Broker Architecture)*, v3.1 that includes the following parts and clauses:
    - **Part I - CORBA Interfaces**
        - The Object Model
        - CORBA Overview
        - OMG IDL Syntax and Semantics
        - ORB Interface
        - Value Type Semantics
        - Abstract Interface Semantics
        - Dynamic Invocation Interface
        - Dynamic Management of Any Values
        - The Interface Repository
        - The Portable Object Adapter
        - Portable Interceptors
        - CORBA Messaging
    - **Part II - CORBA Interoperability**
        - Interoperability Overview
        - ORB Interoperability Architecture
        - Building Inter-ORB Bridges
        - General Inter-ORB Protocol
        - Secure Interoperability
        - Unreliable Multicast Inter-ORB Protocol
2. The Language Mapping Specifications, which are organized into the following stand-alone volumes:
    - Ada Mapping to OMG IDL
    - C Mapping to OMG IDL
    - C++ Mapping to OMG IDL
    - COBOL Mapping to OMG IDL

- IDL Script Mapping
- IDL to Java Mapping
- Java Mapping to OMG IDL
- Lisp Mapping to OMG IDL
- MOF to OMG IDL
- PL/1
- Python Mapping to OMG IDL
- Smalltalk Mapping to OMG IDL
- XML Valuetype Language Mapping

3 Normative References

3.1 General

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions or, any of these publications do not apply.

- OMG CORBA 3.1 specification (formal/2008-01-04): http://www.omg.org/spec/CORBA/3.1
- ISO IEC 14882-2011 (September 1, 2011), Standard for Programming Language C++

4 Additional Information

4.1 Acknowledgements

The following companies submitted the specification that was approved by the Object Management Group to become the C++ Language Mapping specification:

- Digital Equipment Corporation
- Expersoft Corporation
- Hewlett-Packard Company
- IBM Corporation
- IONA Technologies, Ltd.
- Novell USG
- SunSoft, Inc.

5 C++ Language Mapping Specification

5.1 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 7).
- 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++. For consistency, this rule also applies to identifiers that are derived from IDL identifiers. For example, an IDL interface "try" generates the names "_cxx_try_var," "_cxx_try_ptr," and "_tc__cxx_try."

The complete list of C++ keywords can be found in "C++ Keywords" on page 156.

5.2 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 15) and string (see "Mapping for String Types" on page 17).

5.3 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. See the *Common Object Request Broker Architecture*, *Annex A* for standard OMG IDL tags.

5.4 Mapping for Modules

As shown in "Scoped Names" on page 5, 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 "Alternative Mappings For C++ Dialects" on page 154.

5.5 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, or
- 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);
```

5.5.1 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`**. To facilitate template-based programming, typedefs for the **`A_ptr`** and **`A_var`** types are also provided in the interface class (see “Interface Mapping Example” on page 11). The typedef for **`A_ptr`** is named **`A::_ptr_type`** and the typedef for **`A_var`** is named **`A::_var_type`**.

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;
```

5.5.2 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** is not supported; instead, widening between **_var** types for object references requires a call to **_duplicate** (described in “Object Reference Operations” on page 9).[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**. The same rules apply for object reference types that are nested in a complex type, such as a structure or sequence.

```
// 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
```

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.

```
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
```

5.5.3 Object Reference Operations

Conceptually, the **Object** class in the **CORBA** module is the base interface type for all CORBA objects; therefore, any object reference can 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 that 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.

```
// 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.

5.5.4 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 narrowed 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`**, then:

- **`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.

5.5.5 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 7, object references may not be compared using **`operator==`**; therefore, **`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.

5.5.6 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 that 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**. To facilitate template-based programming, a typedef for the **A_out** type is also provided in the interface class (see “Interface Mapping Example” on page 11). The typedef for **A_out** is named **A::_out_type**. 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(const A_out& a) : ptr_(a.ptr_) {}
      A_out& operator=(const 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.

5.5.7 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_var;
class A_out;
class A : public virtual Object
{
   public:
      typedef A_ptr _ptr_type;
      typedef A_var _var_type;
      typedef A_out _out_type;

      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 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
      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 10.

5.6 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];
};
```

5.6.1 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

```
static const WChar *const ws = L”Hello World”;
```

in C++.

5.6.2 Fixed Point Constants

Because C++ does not have a native fixed point type, IDL fixed point literals are mapped to C++ strings without the trailing ‘d’ or ‘D’ in order to guarantee that there is no loss of precision. For example

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.

```
// IDL
const fixed F = 123.456D;

// C++
const Fixed F = "123.456";
```

5.7 Mapping for Basic Data Types

The basic data types have the mappings shown in Table 5.1[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 5.1 - 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_out

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 *Common Object Request Broker Architecture (CORBA), OMG IDL Syntax and Semantics* clause, *Basic Types* sub clause.

Types **boolean, char**, and **octet** may all map to the same underlying C++ type. This means that these types may not be distinguishable for the purposes of overloading.

Type **wchar** maps to **`wchar_t`** in standard C++ environments or, for nonstandard C++ environments, may also map to one of the integer types. This means that **wchar** may not be distinguishable from integer types for purposes of overloading.

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 can be manipulated via built-in operators. An alternate mapping to C++ classes that provides appropriate creation, conversion, and manipulation operators will be provided in a future version of this specification.

All other mappings for basic types are distinguishable for the purposes of overloading. That is, one can safely write overloaded C++ functions for **`Short`**, **`UShort`**, **`Long`**, **`ULong`**, **`LongLong, ULongLong, Float`**, **`Double`**, and **`LongDouble`**.

The **`_out`** types for the basic types are used to type **out** parameters within operation signatures, as described in "Argument Passing Considerations" on page 91. For the basic types, each **`_out`** type is a **`typedef`** to a reference to the corresponding C++ type. For example, the **`Short_out`** is 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.

IDL type **boolean** may be mapped to C++ signed, unsigned, or plain **`char`**. This mapping is legal for both classic and ANSI C++ environments. In addition, in an ANSI C++ environment, IDL **boolean** can be mapped to C++ **`bool`**. Mappings to C++ types other than a character type or **`bool`** are illegal.

5.8 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.

```
// 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 is defined in the same scope as follows:

5. Examples and descriptions in this specification still use `TRUE` and `FALSE` for purposes of clarity.

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

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

5.9 Mapping for String Types

The OMG IDL string type, whether bounded or unbounded, is mapped to `char*`. String data is NUL-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 137. Calling the `out` or `_retn` functions of a `String_var` has the side effect of setting its internal pointer back to null. An application may also explicitly assign a null pointer to the `String_var`.

C++ does not have a built-in type that would provide a "close match" for IDL-bounded strings. As a result, the programmer is responsible for enforcing the bound of bounded strings at run time. Implementations of the mapping are under no obligation to prevent assignment of a string value to a bounded string type if the string value exceeds the bound. Implementations may choose to (at run time) detect attempts to pass a string value that exceeds the bound as a parameter across an interface. If an implementation chooses to detect this error, it must raise a BAD_PARAM system exception to signal the error.

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 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";
```

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(const String_out& s) : ptr_(s.ptr_) {}
      String_out& operator=(const 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

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.

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.

A compliant mapping implementation shall provide overloaded **`operator<<`** (insertion) and **`operator>>`** (extraction) operators for using **`String_var`** and **`String_out`** directly with C++ iostreams. The **`operator>>`** extraction operator has the same semantics as the underlying standard C++ **`operator>>`** for extracting strings from an input stream (extracting until whitespace or end of file). Space to store the extracted characters are allocated by calling **`string_alloc`**, and the previous contents of the **`String_var`** are released by calling **`string_free`**.

5.10 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.

A compliant mapping implementation provides overloaded `operator<<` (insertion) and `operator>>` (extraction) operators for using `WString_var` and `WString_out` directly with C++ iostreams. The `operator>>` extraction operator has the same semantics as the underlying standard C++ `operator>>` for extracting wide strings from an input stream (extracting until whitespace or end of file). Space to store the extracted characters are allocated by calling `wstring_alloc`, and the previous contents of the `WString_var` are released by calling `wstring_free`.

5.11 Mapping for Structured Types

The mapping for **struct, union**, and **sequence** 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 and wide string members to the empty string (`""` and `L""`, respectively). 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 and wide string members. The assignment operator first releases all object reference members and frees all string members and wide 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 and wide 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 or wide string
- A bounded or unbounded sequence
- An object reference or reference to a transmissible pseudo-object
- A **valuetype**
- 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;
```

To facilitate template-based programming, all **`struct`**, **`union`**, and **`sequence`** classes contain nested public typedefs for their associated **`T_var`** and **`T_out`** types. For example, for an IDL **sequence** named **Seq**, the mapped **`sequence`** class **`Seq`** contains a **`_var_type`** and **`_out_type`** typedef as follows:

```
// C++
class Seq_var;
class Seq_out;
class Seq
{
   public:
      typedef Seq_var _var_type;
      typedef Seq_out _out_type;// ...
};
```

5.11.1 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. It is legal to initialize a `T_var` with a null pointer.

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. Assigning a null pointer to a `T_var` is legal and results in deallocation of the data pointed to by the `T_var`.

The overloaded `operator->` returns the `T*` held by the `T_var`, but retains ownership of it. Compliant applications may not de-reference the return value of 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 Table 5.2. 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.

Table 5.2 - Parameter Passing Modes

To pass a `T_var` as an:	an application can call the ...
in parameter	`in()` member function of the `T_var`
inout parameter	`inout()` member function
out parameter	`out()` member function

To obtain a return value from the `T_var`, an application can call the `_retn()` function.[7]

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.

For each `T_var` type, the return types of each of these functions match the types shown in version 2.3 of *The Common Object Request Broker: Architecture and Specifications*, *Mapping: OLE Automation and CORBA* chapter, *Mapping of Automation Types to OMG IDL Types* table 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 of the `T*` owned by a `T_var` when passed as an **out** parameter, as described in "Argument Passing Considerations" on page 91. 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 5.3), 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
```

After `_retn()` is invoked on a `T_var` instance, its internal `T*` pointer is null, so invoking either of its overloaded `operator->` functions without first assigning a valid non-null `T*` to the `T_var` will attempt to de-reference the null pointer, which is illegal in C++.

For reasons of consistency, the **`T_var`** types are also produced for fixed-length structured types. 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. **`T_var`** types for fixed-length structured types have the following general form:

```
// C++
class T_var {
   public:
      T_var() : m_ptr(0) {}
      T_var(T *t) : m_ptr(t) {}
      T_var(const T& t) : m_ptr(new T(t)) {}
      T_var(const T_var &t) : m_ptr(0) {
         if (t.m_ptr != 0)
            m_ptr = new T(*t.m_ptr);
      }
      ~T_var() { delete m_ptr; }
      T_var &operator=(T *t) {
         if (t != m_ptr) {
            delete m_ptr;
            m_ptr = t;
         }
         return *this;
      }
      T_var &operator=(const T& t) {
         if (&t != m_ptr) {
            T* old_m_ptr = m_ptr;
            m_ptr = new T(t);
            delete old_m_ptr;
         }
         return *this;
      }
      T_var &operator=(const T_var &t) {
         if (this != &t) {
            T* old_m_ptr = m_ptr;
            if (t.m_ptr != 0)
               m_ptr = new T(*t.m_ptr);
            else
               m_ptr = 0;
            delete old_m_ptr;
         }
         return *this;
      }
      T* operator->() { return m_ptr; }
      const T* operator->() const { return m_ptr; }
      const T& in() const { return *m_ptr; }
      T& inout() { return *m_ptr; }
      T& out() {
         if (m_ptr == 0)
            m_ptr = new T;
         return *m_ptr;
      }
      T _retn() { return *m_ptr; }

   private:
      T* m_ptr;
};
```

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);
```

5.11.2 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 that 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(const T_out& p) : ptr_(p.ptr_) {}

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

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

```
      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
```

For fixed-length underlying types, no memory management issues arise; however, a compliant mapping must provide the following type definition in the scope of **`T`**

```
typedef T &T_out;
```

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.

5.12 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. The C++ structure members appear in the same order as the corresponding IDL structure members. 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, wide string, or object reference member, the name of the C++ member's type is not specified by the mapping; therefore, a compliant program cannot create an object of that type. The behavior of the type is the same as the normal mapping (**`char*`** for string, **`WChar*`** for wide string, and **`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 Table 5.2. A compliant mapping implementation also provides overloaded **`operator<<`** (insertion) and **`operator>>`** (extraction) operators for using string members and wide string members directly with C++ iostreams.

For anonymous sequence members (required for recursive structures), a type name is required for the member. This name is generated by prepending an underscore to the member name, and appending "_seq".

For example

```
// IDL
struct node {
        long value;
        sequence<node, 2> operand;
};
```

This results in the following C++ code

```
// C++
struct node {
      typedef ... _operand_seq;
      Long value;
      _operand_seq operand;
};
```

In the C++ code shown above, the "..." in the **`_operand_seq`** typedef refers to an implementation-specific sequence type. The name of this type is not standardized.

Assignment between a string, wide string, or object reference member and a corresponding **`T_var`** type (**`String_var`**, **`WString_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*`** or **`const WChar*`** 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};
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 empty
Variable_var str2 = new Variable;// str2->name is
                                 // initially empty

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 empty strings. 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 an empty string, so it must 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.

Compliant programs use **`new`** to dynamically allocate structs and **`delete`** to free them.

5.13 Mapping for Fixed Types

The C++ mapping for **fixed** is defined by the following class.

```
// C++
class Fixed
{
   public:
      // Constructors
      Fixed(int val = 0);
      Fixed(unsigned val);
      Fixed(Long val);
      Fixed(ULong val);
      Fixed(LongLong val);
      Fixed(ULongLong val);
      Fixed(Double val);
      Fixed(LongDouble val);
      Fixed(const Fixed& val);
      Fixed(const char*);
      ~Fixed();

      // Conversions
      operator LongLong() const;
      operator LongDouble() const;
      Fixed round(UShort scale) const;
      Fixed truncate(UShort scale) const;
      char *to_string() const;
      // Operators
      Fixed& operator=(const Fixed& val);
      Fixed& operator+=(const Fixed& val);
      Fixed& operator-=(const Fixed& val);
      Fixed& operator*=(const Fixed& val);
      Fixed& operator/=(const Fixed& val);

      Fixed& operator++();
      Fixed operator++(int);
      Fixed& operator--();
      Fixed operator--(int);
      Fixed operator+() const;
      Fixed operator-() const;
      Boolean operator!() const;

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

istream& operator>>(istream& is, Fixed& val);
ostream& operator<<(ostream& os, const Fixed& val);

Fixed operator + (const Fixed& val1, const Fixed& val2);
Fixed operator - (const Fixed& val1, const Fixed& val2);
Fixed operator * (const Fixed& val1, const Fixed& val2);
Fixed operator / (const Fixed& val1, const Fixed& val2);

Boolean operator > (const Fixed& val1, const Fixed& val2);
Boolean operator < (const Fixed& val1, const Fixed& val2);
Boolean operator >= (const Fixed& val1, const Fixed& val2);
Boolean operator <= (const Fixed& val1, const Fixed& val2);
Boolean operator == (const Fixed& val1, const Fixed& val2);
Boolean operator != (const Fixed& val1, const Fixed& val2);
```

The **`Fixed`** class is used directly by the C++ mapping for IDL fixed-point constant values and for all intermediate results of arithmetic operations on fixed-point values. For fixed-point parameters of IDL operations or members of IDL structured datatypes, the implementation may use the **`Fixed`** type directly, or alternatively, may use a different type, with an effectively constant digits and scale, that provides the same C++ interface and can be implicitly converted from/to the **`Fixed`** class. The name(s) of this alternative class is not defined by this mapping. Since fixed-point types used as parameters of IDL operations must be named via an IDL **typedef** declaration, the mapping must use the **typedef** to define the type of the operation parameter to make sure that server-side operation signatures are portable. Below is an example of the mapping.

// IDL
typedef fixed<5,2> F;

interface A

```
{
        void op(in F arg);
};

// C++
typedef Implementation_Defined_Class F;

class A
{
   public:
   ...
   void op(const F& arg);
   ...
};
```

The **`Fixed`** class has a number of constructors to guarantee that a fixed value can be constructed from any of the IDL standard integer and floating point types. The **`Fixed(char*)`** constructor converts a string representation of a fixed-point literal, with an optional leading sign (+ or -) and an optional trailing 'd' or 'D,' into a real fixed-point value. The **`Fixed`** class also provides conversion operators back to the **`LongLong`** and **`LongDouble`** types. For conversion to integral types, digits to the right of the decimal point are truncated. If the magnitude of the fixed-point value does not fit in the target conversion type, then the DATA_CONVERSION system exception is thrown.

The **`round`** and **`truncate`** functions convert a fixed value to a new value with the specified scale. If the new scale requires the value to lose precision on the right, the **`round`** function will round away from zero values that are halfway or more to the next absolute value for the new fixed precision. The **`truncate`** function always truncates the value towards zero. If the value currently has fewer digits on the right than the new scale, **`round`** and **`truncate`** return the argument unmodified.

For example

```
// C++
Fixed f1 = "0.1";
Fixed f2 = "0.05";
Fixed f3 = "-0.005;
```

In this example, **`f1.round(0)`** and **`f1.truncate(0)`** both return 0, **`f2.round(1)`** returns 0.1, **`f2.truncate(1)`** returns 0.0, **`f3.round(2)`** returns -0.01 and **`f3.truncate(2)`** returns 0.00.

`to_string()` converts a fixed value to a string. Leading zeros are dropped, but trailing fractional zeros are preserved. (For example, a **fixed<4,2>** with the value 1.1 is converted "1.10".) The caller of **`Fixed::to_string()`** must deallocate the return value by calling **`CORBA::string_free()`** or assigning the return value to a **`String_var`**.

The **`fixed_digits`** and **`fixed_scale`** functions return the smallest digits and scale value that can hold the complete fixed-point value. If the implementation uses alternative classes for operation parameters and structured type members, then **`fixed_digits`** and **`fixed_scale`** return the constant digits and scale values defined by the source IDL fixed-point type.

Arithmetic operations on the **`Fixed`** class must calculate the result exactly, using an effective double precision (62 digit) temporary value. The results are then truncated at run time to fit in a maximum of 31 digits using the method defined in version 2.3 of the *Common Object Request Broker Architecture (CORBA)*, *OMG IDL Syntax and Semantics* clause, *Semantics* sub clause to determine the new digits and scale. If the result of any arithmetic operation produces more than

31 digits to the left of the decimal point, the DATA_CONVERSION exception will be thrown. If a fixed-point value, used as an actual operation parameter or assigned to a member of an IDL structured datatype, exceeds the maximum absolute value implied by the digits and scale, the DATA_CONVERSION exception will be thrown.

The stream insertion and extraction operators << and >> convert a fixed-point value to/from a stream. The exact definition of these operators may vary depending on the level of standardization of the C++ environment. These operators insert and extract fixed-point values into the stream using the same format as for C++ floating point types. In particular, the trailing 'd' or 'D' from the IDL fixed-point literal representation is not inserted or extracted from the stream. These operators use all format controls appropriate to floating point defined by the stream classes except that they never use the scientific format.

5.13.1 Fixed T_var and T_out Types

Because fixed-point types are always passed by reference as operation parameters and returned by value, there is no need for a **`_var`** type for a fixed-point type. For each IDL fixed-point **typedef** a corresponding **`_out`** type is defined as a reference to the fixed-point type.

```
// IDL
typedef fixed<5,2> F;

// C++
typedef Implementation_Defined_Name F;
typedef F& F_out;
```

5.14 Mapping for Union Types

Unions map to C++ classes with access functions for the union members and discriminant. Some member functions only provide read access to a member. Such functions are called "accessor functions" or "accessors" for short. For example

```
// C++
Long x() const;
```

Here, **`x()`** is an accessor that returns the value of the member **`x`** of a union (of type **`Long`** in this example).

Other member functions only provide write access to a union member. Such functions are called "modifier functions" or "modifiers" for short. For example

```
// C++
void x(Long val);
```

Here, **`x()`** is a modifier that sets the value of the member **`x`** of a union (of type **`Long`** in this example).

Still other union member functions provide read-write access to a union member by returning a reference to that member. Such functions are called "reference functions" or "referents" for short. For example

```
// C++
S& w();
```

Here, **`w()`** is a referent to the member **`w`** (of type **`S`**) of a union.

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.) Assigning, copying, and the destruction of default-constructed unions are safe. Assignment from or copying a default-constructed union results in the target of the assignment or copy being initialized the same as a default-constructed union.

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.

The union discriminant accessor and modifier functions have the name **`_d`** to both be brief and to avoid name conflicts with the union members. The **`_d`** discriminator modifier can only be used to set the discriminant to a value within the same union member. In addition to the **`_d`** accessor and modifier, a union with an implicit default member provides a **`_default()`** modifier 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. Assigning, copying, and the destruction of a union immediately after calling **`_default()`** are safe. Assignment from or copying of such a union results in the target of the assignment or copy having the same safe state as it would if its **`_default()`** function were invoked.

Setting the union value through a modifier function automatically sets the discriminant and may release the storage associated with the previous value. Attempting to get a value through an accessor that does not match the current discriminant results in undefined behavior. If a modifier 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. Calling a referent for a member that does not match the current discriminant results in undefined behavior.

The following example helps illustrate the mapping for union types.

```
// IDL
typedef octet Bytes[64];
struct S { long len; };
interface A;
valuetype Val;
union U switch (long) {
        case 1: long x;
        case 2: Bytes y;
        case 3: string z;
        case 4:
        case 5: S w;
        case 6: Val v;
        default: A obj;
};

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

```
class Val ... ;
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,
      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 v(Val*);           // _remove_ref old valuetype,
                              // _add_ref argument
      Val* v() const;         // no _add_ref of return value

      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). Referents can be used for read-write access, but are only provided for the following types: **struct, union, sequence, any**, and **fixed**.

The reference returned from a reference function continues to denote that member only for as long as the member is active. If the active member of the union is subsequently changed, the reference becomes invalid, and attempts to read or write the member via the reference result in undefined behavior.

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 43). 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[9] 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.[10] The union will also provide modifier functions that take the unnamed string struct member, array member, and sequence member types as a parameter, with the same semantics as the **`String_var`** modifier.

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.

For anonymous sequence union members (required for recursive unions), a type name is required. This name is generated by prepending an underscore to the member name, and appending "_seq."

For example

```
// IDL
union node switch (long) {
        case 0: long value;
        case 1: sequence<node, 2> operand;
};
```

This results in the following C++

9. 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.
10. 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++
class node {
   public:
      typedef ... _operand_seq;
      ...
      // Member functions dealing with the operand
      // member use _operand_seq for its type.
      ...
};
```

In the C++ code shown above, the "..." in the `_operand_seq` typedef refers to an implementation-specific sequence type. The name of this type is not standardized.

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.

```
// 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
s = u.w();      // error, member w not active
```

As shown here, neither the `_d` modifier function nor the `w` referent can 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()` modifier 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.

Compliant programs use `new` to dynamically allocate unions and `delete` to free them.

5.15 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 length of a sequence never changes without an explicit call to the length() member function.

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 typedef naming an anonymous sequence type, a compliant mapping implementation provides a separate C++ sequence type. To facilitate template-based programming, a nested public typedef **`_size_type`** is delivered as the type representing the length and maximum of the sequence.

For example

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

// C++
class LongSeq           // unbounded sequence
{
   public:
      typedef ULong _size_type;
      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:
      typedef ULong _size_type;
      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. Default constructors for bounded and unbounded sequences need not allocate buffers immediately.

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 buffer is determined by the **`release`** parameter—**`FALSE`** means the caller owns the storage for the buffer and its elements, while **`TRUE`** means that the sequence assumes ownership of the storage for the buffer and its elements. If **`release`** is **`TRUE`**, the buffer is assumed to 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 42.

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*), and destroys the underlying sequence buffer.

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 for all elements and the buffer. If **`release`** is **`FALSE`** under these circumstances, old storage will not be freed for either the elements or for the buffer 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 number of sequence elements that can be stored in the current sequence buffer. 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 **`length()`** functions can be used to access and modify the length of the sequence. Increasing the length of a sequence adds new elements at the tail. The newly-added elements behave as if they are default-constructed when the sequence length is increased. However, a sequence implementation may delay actual default construction until a newly-added element is first accessed. For sequences of strings and wide strings, default element construction requires initialization of each element to the empty string or wide string. For sequences of object references, default element construction requires initialization of each element to a suitably-typed nil reference. For sequences of valuetypes, default element construction requires initialization of each element to a null pointer. The elements of sequences of other complex types, such as structs and sequences, are initialized by their default constructors. Union sequences elements do not have any application-visible initialization; in particular, a default-constructed union element is not safe for marshaling or access. Sequence elements of a basic type, such as **`ULong`**, have undefined default values.

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, wide strings, and object references, **`operator[]`** for a sequence must return a type with the same semantics as the types used for string, wide string, and object reference members of structs and arrays, so that assignment to the string, wide 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. A compliant mapping implementation also provides overloaded **`operator<<`** (insertion) and **`operator>>`** (extraction) operators for using string sequence elements and wide string sequence elements directly with C++ iostreams.

The **`release()`** accessor function returns the state of the sequence release flag.

The overloaded **`get_buffer()`** accessor and reference functions allow direct access to the buffer underlying a sequence. This can be very useful when sending large blocks of data as sequences, such as sending image data as a sequence of octet, and the per-element access provided by the overloaded subscript operators is not sufficient.

The non-const **`get_buffer()`** reference function allows read-write access to the underlying buffer. If its **`orphan`** argument is **`FALSE`** (the default), the sequence returns a pointer to its buffer, allocating one if it has not yet done so. The size of the buffer can be determined using the **`maximum()`** accessor. For bounded sequences, the size of the returned buffer is equal to the sequence bound. The number of elements in the buffer can be determined from the sequence **`length()`** accessor. The sequence maintains ownership of the underlying buffer. Elements in the returned buffer may be directly replaced by the caller. For sequences of strings, wide strings, and object references, the caller must use the sequence **`release()`** accessor to determine whether elements should be freed (using **`string_free`**, **`wstring_free`**, or **`CORBA::release`** for string, wide strings, and object references, respectively) before being directly assigned to. Because the sequence maintains a notion of the length and size of the buffer, the caller of **`get_buffer()`** shall not lengthen or shorten the sequence by directly adding elements to the buffer or directly removing elements from the buffer. Changing the length of the sequence shall be performed only by invoking the sequence **`length()`** modifier function.

Alternatively, if the **`orphan`** argument to **`get_buffer()`** is **`TRUE`**, the sequence yields ownership of the buffer to the caller. If **`orphan`** is **`TRUE`** and the sequence does not own its buffer (i.e., its **`release`** flag is **`FALSE`**), the return value is a null pointer. If the buffer is taken from the sequence using this form of **`get_buffer()`**, the sequence reverts to the same state it would have if constructed using its default constructor. The caller becomes responsible for eventually freeing each element of the returned buffer (for strings, wide string, and object references), and then freeing the returned buffer itself using **`freebuf`**.

The const **`get_buffer()`** accessor function allows read-only access to the sequence buffer. The sequence returns its buffer, allocating one if one has not yet been allocated. No direct modification of the returned buffer by the caller is permitted.

For the non-const **`get_buffer()`** reference function with an **`orphan`** argument of **`FALSE`**, and for the const **`get_buffer()`** accessor function, the return value remains valid until another non-const member function of the sequence is invoked, or until the sequence is destroyed, whichever occurs first.

The **replace()** function allows the buffer underlying a sequence to be replaced. The parameters to **replace()** are identical in type, order, and purpose to those for the **T *data** constructor for the sequence.

Access to the underlying sequences buffers seems to imply that a sequence implementation must use contiguous memory to hold the elements, but this need not be the case. A compliant sequence implementation could keep its elements in several separate memory buffers and relocate them to a single buffer only if the application called the **get_buffer()** accessors. In fact, for applications that never invoke these accessors, such an implementation would very likely be better suited to handling large sequences than one using a large single contiguous buffer.

For the **T *data** sequence constructor and for the buffer parameter of the **replace()** function, the type of **T** for strings, wide strings, and object references is **char***, **CORBA::WChar***, and **T_ptr**, respectively. In other words, string buffers are passed as **char****, wide string buffers as **CORBA::WChar****, and object reference buffers as **T_ptr***. The return type of the non-const **get_buffer()** reference function for sequences of strings is **char****, **CORBA::WChar**** for sequences of wide strings, and **T_ptr*** for sequences of object references. The return type of the const **get_buffer()** accessor function for sequences of strings is **const char* const***, **const CORBA::WChar* const*** for sequences of wide strings, and **const T_ptr*** for sequences of object reference.

5.15.1 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:
      typedef ULong _size_type;
      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;

      Boolean release() const;

      void replace(ULong max, ULong length, T *data,
               Boolean release = FALSE);
      T* get_buffer(Boolean orphan = FALSE);
      const T* get_buffer() const;
};

class V2                                //bounded sequence
{
   public:
      typedef ULong _size_type;
      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;

      Boolean release() const;

      void replace(ULong length, T *data,
                  Boolean release = FALSE);

      T* get_buffer(Boolean orphan = FALSE);
      const T* get_buffer() const;
};
```

5.15.2 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();
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 then call **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 previous versions of this specification provided no means for the callee to determine the setting of the sequence **release** flag, and thus the callee always had to assume that **release** was 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.

For a sequence passed to an operation as an **in** parameter, the operation must not assign to the sequence if its release flag is **FALSE** and the sequence has variable-length elements.

For a sequence passed to a client as an **out** parameter or return value, the client must not assign to the sequence if its release flag is **FALSE** and the sequence has variable-length elements.

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.

5.15.3 Additional Memory Management Functions

Compliant programs use **new** to dynamically allocate sequences and **delete** to free them.

Sequences also provide additional memory management functions for their buffers. For an unbounded 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 and to the **replace()** member function. 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 and wide strings, which are initialized to pointers to empty string, and object references, which are initialized to suitably-typed nil object references. A null pointer is returned if for some reason **allocbuf** cannot allocate the requested vector.

For bounded sequences, the following static member functions are provided in the sequence class public interface.

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

For bounded sequences, the first (zero parameter) version of **`allocbuf`** allocates a buffer of **`maximum()`** elements. A null pointer is returned if the function cannot allocate the requested vector.

Note that the version of **`allocbuf`** that accepts an element count is deprecated for bounded sequences and will be removed in a future version of the mapping. Calls to the deprecated version with an argument value other than the sequence maximum have implementation-dependent behavior.

Vectors allocated by **`allocbuf`** must 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 and wide string elements, which are freed using **`string_free()`** and **`wstring_free()`**, respectively, and object reference elements, which are freed using **`CORBA::release()`**. The **`freebuf`** function will ignore null pointers passed to it. Neither **`allocbuf`** nor **`freebuf`** may throw CORBA exceptions.

A call to **`allocbuf`** with a zero-value argument causes **`allocbuf`** to allocate a zero-length buffer and return a pointer to it. Like any buffer returned from **`allocbuf`**, this buffer must be freed using the corresponding **`freebuf`** function.

5.15.4 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.[11] This subscript operator should have the same return type as that of the corresponding operator on the underlying sequence type.

5.16 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, wide string, or an object reference, then the mapping uses the same type as for structure members. That is, the default constructor for string elements and wide string elements initializes them to the empty string (**`""`** and **`L""`**, respectively), and assignment to an array element that is a string, wide string, or object reference 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);
```

11. 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 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 Table 5.3, **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 47. 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 51.

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_copy(T_slice* to, const T_slice* from);
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_copy** function copies the contents of the *from* array to the *to* array. If either argument is a null pointer, **T_copy** does not attempt a copy and results in no action being performed. 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. The **T_alloc**, **T_dup**, and **T_free** 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**, **T_copy**, and **T_free** functions may not throw CORBA exceptions.

5.17 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...
class 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_copy(another_array_slice* to,
      const another_array_slice* from)
{
      array_copy(to, from);
}

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

5.18 Mapping for the Any Type

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

1. Handling C++ types in a type-safe manner.
2. 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.

5.18.1 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 “Mapping for Basic Data Types” on page 15, the **boolean**, **octet**, **char**, and **wchar** 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 53.
- 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 53.
- 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 43.

5.18.2 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**)
- Pointers to **valuetype**s (**T***)

For values of type **T** that are too large to be passed by value efficiently, such as structs, unions, sequences, **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 non-copying 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 was called to create the **Any**, the **Any** is responsible for de-allocating 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 values being inserted are unbounded. "Distinguishing boolean, octet, char, wchar, bounded string, and bounded wstring" on page 53 describes how bounded strings and bounded wide strings may be correctly inserted into an **`Any`**. Note that insertion of wide strings in this manner depends on standard C++, in which **`wchar_t`** is a distinct type. Code that must be portable across standard and older C++ compilers must use the **`Any::from_wstring`** helper. Noncopying insertion of both bounded and unbounded strings can be achieved using the **`Any::from_string`** helper type. Similarly, noncopying insertion of bounded and unbounded wide 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 53.

Note that the following code has undefined behavior in nonstandard C++ environments:

```
// C++
Any a = ...;
WChar wc;
a >>= wc;   // undefined behavior
```

This code may erroneously extract an integer type in environments where **`wchar_t`** is not a distinct type.

Because **valuetype**s may be represented legally using null pointers, a conforming application may insert a null **valuetype** pointer into an **`Any`**.

Type-safe insertion of arrays uses the **`Array_forany`** types described in "Mapping For Array Types" on page 43. 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[12] use the appropriate **`Array_forany`** type when inserting an array into an **`any`**.

12. A mapping implementor may use the new C++ keyword "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 noncopying version of **operator<<=** for **valuetype**s takes the address of the **T*** pointer type.

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

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

The noncopying **valuetype** insertion consumes the **valuetype** pointed to by the pointer that **T**** points to. After insertion, the caller may not access the **valuetype** instance pointed to by the pointer that **T*** points to. The caller maintains ownership of the storage for the pointed-to **T*** itself.

In general, the copying versions of **operator<<=** are 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**.

5.18.3 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, **Any**, **valuetype**s, and exceptions) this function may be prototyped as follows:

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

The non-constant version of the operator will be deprecated in a future version of the mapping and should not be used. The first form of this function is used only for the following types:

- **Short**, **UShort**, **Long**, **ULong**, **LongLong**, **ULongLong**, **Float**, **Double**, **LongDouble**
- Enumerations
- Unbounded strings and wide strings (**const char*** and **const WChar*** passed by reference (i.e., **const char*&** and **const WChar*&**)[13]
- 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**.

13. Note that extraction of wide strings in this manner depends on standard C++, in which **wchar_t** is a distinct type. Code that must be portable across standard and older C++ compilers must use the **to_wstring** helper type.

For non-primitive types, such as struct, union, sequence, exception, and **Any**, extraction is done by pointer to **const** (**valuetypes** are extracted by pointer to non-**const** because **valuetype** operations do not support **const**). 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 ...
const 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. An attempt to extract to a **T_var** type is non-conforming and must cause a compile-time error.

If the extraction is not successful, the value of the caller's pointer is set equal to the null pointer, and **operator>>=** returns **FALSE**. Note that because **valuetype**s may legally be represented as null pointers, however, a pointer to **T** extracted from an **Any**, where **T** is a **valuetype**, may be null even when extraction is successful if the **Any** holds a null **valuetype** pointer.

Correct extraction of array types relies on the **Array_forany** types described in "Mapping For Array Types" on page 43.

```
// 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`**.

5.18.4 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*`**, both bounded and unbounded wide strings map to **`WChar*`**, and all fixed-point types map to the **`Fixed`** class, 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 n = FALSE) :
            val(s), bound(b), nocopy(n) {}
         from_string(const char* s, ULong b) :
            val (const_cast<char*>(s)), bound(b),
            nocopy (0) {}
         char *val;
         ULong bound;
         Boolean nocopy;
      };
      struct from_wstring {
         from_wstring(WChar* s, ULong b,
                  Boolean n = FALSE) :
            val(s), bound(b), nocopy(n) {}
         from_wstring(const WChar* s, ULong b) :
            val(const_cast<WChar*>(s)), bound(b),
            nocopy(0) {}
         WChar *val;
         ULong bound;
         Boolean nocopy;
   };
   struct from_fixed {
         from_fixed(const Fixed& f, UShort d, UShort s)
                  : val(f), digits(d), scale(s) {}
         const Fixed& val;
         UShort digits;
         UShort scale;
   };

   void operator<<=(from_boolean);
   void operator<<=(from_char);
   void operator<<=(from_wchar);
   void operator<<=(from_octet);
   void operator<<=(from_string);
   void operator<<=(from_wstring);
   void operator<<=(from_fixed);
   // 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(const char *&s, ULong b)
         : val(s), bound(b) {}
         const char *&val;
         ULong bound;

         // the following constructor is deprecated
         to_string(char *&s, ULong b) : val(s), bound(b) {}
   };
   struct to_wstring {
         to_wstring(const WChar *&s, ULong b)
            : val(s), bound(b) {}
         const WChar *&val;
         ULong bound;

         // the following constructor is deprecated
         to_wstring(WChar *&s, ULong b)
            :val(s), bound(b) {}
   };
   struct to_fixed {
         to_fixed(Fixed& f, UShort d, UShort s)
            : val(f), digits(d), scale(s) {}
         Fixed& val;
         UShort digits;
         UShort scale;
   };

      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;
      Boolean operator>>=(to_fixed) 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...
}

const 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. Note that the non-constant versions of the **to_string** and **to_wstring** constructors will be removed in a future version of the mapping and should not be used.

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.

In standard C++ environments, the mapping implementation must declare the constructors of the **from_** and **to_** helper classes as **explicit**. This prevents undesirable conversions via temporaries.

5.18.5 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_out 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. Unlike for regular object reference extraction, the lifetime of an object reference extracted using **to_object** is independent of that of the **Any** that it is extracted from, and so the responsibility for invoking **release** on it becomes that of the caller.

5.18.6 Widening to Abstract Interface

The **CORBA::Any::to_abstract_base** type allows the contents of an **Any** to be extracted as an **AbstractBase** if the entity stored in the **Any** is an object reference type or a **valuetype** directly or indirectly derived from the **AbstractBase** base class. The **to_abstract_base** type is shown below.

```
// C++
class Any {
   public:
      ...
      struct to_abstract_base {
         to_abstract_base(AbstractBase_ptr& base)
            : ref(base) {}
         AbstractBase_ptr& ref;
      };
      Boolean operator>>=(to_abstract_base val) const;
      ...
};
```

The caller is responsible for releasing the returned **AbstractBase_ptr**. See “Abstract Interface Base” on page 83 for a description of **AbstractBase**.

5.18.7 Widening to ValueBase

The **CORBA::Any::to_value** type allows the contents of an **Any** to be extracted as a **ValueBase*** if the entity stored in the **Any** is a **valuetype**. The **to_value** type is shown below.

```
// C++
class Any {
   public:
      ...
      struct to_value {
         to_value(ValueBase*& base) : ref(base) {}
         ValueBase*& ref;
      };
      Boolean operator>>=(to_value val) const;
      ...
};
```

The caller is responsible for calling **_remove_ref** on the returned **ValueBase** pointer. See “ValueBase and Reference Counting” on page 64 for a description of **ValueBase**.

5.18.8 TypeCode Replacement

The **type** accessor 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.

```
TypeCode_ptr type() const;
```

Because C++ **typedef**s are only aliases and do not define distinct types, inserting a type with a **tk_alias TypeCode** into an **Any** while preserving that **TypeCode** is not possible.

For example

```
// IDL
typedef long LongType;

// C++
Any any;
LongType val = 1234;
any <<= val;
TypeCode_var tc = any.type();
assert(tc->kind() == tk_alias);     // assertion failure!
assert(tc->kind() == tk_long);      // assertion OK
```

In this code, the **LongType** is an alias for **CORBA::Long**. Therefore, when the value is inserted, standard C++ overloading mechanisms cause the insertion operator for **CORBA::Long** to be invoked. In fact, because **LongType** is an alias for **CORBA::Long**, an overloaded **operator<<=** for **LongType** cannot be generated anyway.

In cases where the **TypeCode** in the **Any** must be preserved as a **tk_alias TypeCode**, the application can use the **type** modifier function on the **Any** to replace its **TypeCode** with an equivalent one.

```
void type(TypeCode_ptr);
```

Revising the previous example

```
// C++
Any any;
LongType val = 1234;
any <<= val;
any.type(_tc_LongType);             // replace TypeCode
TypeCode_var tc = any.type();
assert(tc->kind() == tk_alias);     // assertion OK
```

The **type** modifier function invokes the **TypeCode::equivalent** operation on the **TypeCode** in the target **Any**, passing the **TypeCode** it received as an argument. If **TypeCode::equivalent** returns true, the **type** modifier function replaces the original **TypeCode** in the **Any** with its argument **TypeCode**. If the two **TypeCode**s are not equivalent, the **type** modifier function raises the BAD_TYPECODE exception.

5.18.9 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.

Compliant programs use **new** to dynamically allocate anys and **delete** to free them.

5.18.10 The Any Class

The full definition of the **Any** class can be found in "Any Class" on page 140.

5.18.11 The Any_var and Any_out Classes

Because **Any**s are returned via pointer as **out** and return parameters (see Table 5.3), 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, 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 144. An **Any_out** class is also available that is similar in form to the **T_out** class described in "T_out Types" on page 25.

5.19 Mapping for Valuetypes

The IDL **valuetype** has features that make its C++ mapping unlike that of any other IDL type. Specifically, from an application perspective all other IDL types comprise either pure state or pure interface, but a **valuetype** may include both. Because of this, the C++ mapping for the **valuetype** is necessarily more restrictive in terms of implementation than other parts of the C++ mapping.

An IDL **valuetype** is mapped to a C++ class with the same name as the IDL **valuetype**. This class is an abstract base class (ABC), with pure virtual accessor and modifier functions corresponding to the state members of the **valuetype**, and pure virtual functions corresponding to the operations of the **valuetype**.

A C++ class whose name is formed by prepending the string "OBV_" to the fully-scoped name of the **valuetype** provides default implementations for the accessors and modifiers of the ABC base class. The application developer then overrides the pure virtual functions corresponding to **valuetype** operations in a concrete class derived directly or indirectly from the **OBV_** base class.

Applications are responsible for the creation of **valuetype** instances, and after creation, they deal with those instances only via C++ pointers. Unlike object references, which map to C++ **_ptr** types that may be implemented either as actual C++ pointers or as C++ pointer-like objects, "handles" to C++ **valuetype** instances are actual C++ pointers. This helps to distinguish them from object references.

Because **valuetype** supports the sharing of instances within other constructed types (such as graphs), the lifetimes of C++ **valuetype** instances are managed via reference counting. Unlike the semantics of object reference counting, where neither **duplicate** nor **release** actually affect the object implementation, reference counting operations for C++ **valuetype** instances are directly implemented by those instances. Reference counting mix-in classes are provided by ORB implementations for use by **valuetype** implementors (see "Reference Counting Mix-in Classes" on page 66).

As for most other types in the C++ mapping, each **valuetype** also has an associated C++ **`_var`** type that automates its reference counting. To facilitate template-based programming, typedefs for the **`_ptr`**, **`_out`,** and **`_var`** types are provided in the valuetype class. The typedef for **`_ptr`** is named **`_ptr_type`**, the typedef for **`_out`** is named **`_out_type`**, and the typedef for **`_var`** is named **`_var_type`**. All **init** initializers declared for a **valuetype** are mapped to pure virtual functions on a separate abstract C++ factory class. The class is named by appending "_init" to the name of the **valuetype (**e.g., type **`A`** has a factory class named **`A_init)`**.

5.19.1 Valuetype Data Members

The C++ mapping for **valuetype** data members follows the same rules as the C++ mapping for unions, except that the accessors and modifiers are pure virtual. Public state members are mapped to public pure virtual accessor and modifier functions of the C++ **valuetype** base class, and private state members are mapped to protected pure virtual accessor and modifier functions (so that derived concrete classes may access them). Portable applications that use **`OBV_`** classes, including derived value type classes, shall not access the actual data members of **`OBV_`** classes, and ORB implementations are free to make such members private. The only requirement on the actual data members in a concrete or partially-concrete class such as an **`OBV_`** class is that they be self-managing so that derived classes can correctly implement copying without needing direct access to them.

Like C++ unions, the accessor and modifier functions for **valuetype** state members do not follow the regular C++ parameter passing rules. This is because they allow local program access to the state stored inside the **valuetype** instance. Modifier functions perform the equivalent of a deep-copy of their parameters, and accessors that return a reference or pointer to a state member can be used for read-write access.

For example

```
// IDL
typedef octet Bytes[64];
struct S { ... };
interface A { ... };

valuetype Val {
        public Val t;
        private long v;
        public Bytes w;
        public string x;
        private S y;
        private A z;
};

// C++
typedef Octet Bytes[64];
typedef Octet Bytes_slice;
...
struct S { ... };

typedef ... A_ptr;
typedef ... Val_ptr;
class Val_out;
class Val_var;
class Val : public virtual ValueBase {
   public:
      ...
        typedef Val_ptr _ptr_type;
        typedef Val_var _var_type;
        typedef Val_out _out_type;

        virtual Val* t() const = 0;
        virtual void t(Val*) = 0;

        virtual const Bytes_slice* w() const = 0;
        virtual Bytes_slice* w() = 0;
        virtual void w(const Bytes) = 0;

        virtual const char* x() const = 0;
        virtual void x(char*) = 0;
        virtual void x(const char*) = 0;
        virtual void x(const String_var&) = 0;

   protected:
        virtual Long v() const = 0;
        virtual void v(Long) = 0;

        virtual const S& y() const = 0;
        virtual S& y() = 0;
        virtual void y(const S&) = 0;

        virtual A_ptr z() const = 0;
        virtual void z(A_ptr) = 0;
        ...
};
```

The following rules apply to the accessor and modifier functions shown in the above example:

- The `t` accessor function does not increment the reference count of the returned **valuetype**. This implies that the caller of `t` does not adopt the return value.
- The `t` modifier function increments the reference count of its argument, then decrements the reference count of the `t` member it is replacing before returning.
- The `x(char*)` modifier function frees the old string member and adopts its argument.
- The `x(const char*)` modifier function frees the old string member and copies its argument.
- The `x(const String_var&)` modifier function frees the old string member and copies its argument.
- By returning a reference to a const `S`, the first `y` accessor function provides read-only access to the `y` member.
- By returning a reference to an `S`, the second `y` accessor function provides read-write access to the `y` member.
- The `y` modifier function deep-copies its `S` argument.
- The `z` accessor function does not invoke `_duplicate` on the object reference it returns. This implies that the caller of `z` is not responsible for invoking `release` on the return value.

- The **z** modifier function releases its old object reference corresponding to the **z** member, then duplicates its argument before returning.

These rules correspond directly to the parameter passing rules for union accessors and modifiers as explained in “Mapping for Union Types” on page 32.

State members of anonymous array and sequence types require the same supporting C++ typedefs as required for union members of anonymous array and sequence types; see “Mapping for Union Types” on page 32 for more details.

5.19.2 Constructors, Assignment Operators, and Destructors

A C++ **valuetype** class defines a protected default constructor and a protected virtual destructor. The default constructor is protected to allow only derived class instances to invoke it, while the destructor is protected to prevent applications from invoking delete on pointers to value instances instead of using reference counting operations. The destructor is virtual to provide for proper destruction of derived value class instances when their reference counts drop to zero.

For the same reasons, a C++ **OBV_** class defines a protected default constructor, a protected constructor that takes an initializer for each **valuetype** data member, and a protected destructor. The parameters of the constructor that takes an initializer for each member appear in the same order as the data members appear, top to bottom, in the IDL **valuetype** definition, regardless of whether they are public or private. If the valuetype inherits from a concrete valuetype, then parameters for the data members of the inherited valuetype appear first. All parameters for the member initializer constructor follow the C++ mapping parameter passing rules for **in** arguments of their respective types. For **valuetype**s that have no operations other than factory initializers, the same constructors and destructors are generated, but with public access so that they can be called directly by application code.

Portable applications shall not invoke a **valuetype** class copy constructor or default assignment operator. Due to the required value reference counting, the default assignment operator for a **valuetype** class shall be private and preferably unimplemented to completely disallow assignment of **valuetype** instances.

5.19.3 Valuetype Operations

Operations declared on a **valuetype** are mapped to public pure virtual member functions in the corresponding **valuetype** C++ class. (Note that state member accessor and modifier functions are *not* considered to be operations—they have different parameter passing rules than operations and so they are always referred to as accessor and modifier functions.) None of the pure virtual member functions corresponding to operations shall be declared **const** because unlike C++, IDL provides no way to distinguish between operations that change the state of an object and those that merely access that state. This choice, similar to the choice made for the C++ mapping for operations declared in IDL **interface** types, has an impact on parameter passing rules, as described in “Argument Passing Considerations” on page 91. The alternative, declaring all pure virtual member functions as **const**, is less desirable because it would not allow member functions inherited from **interface** classes to be invoked on **const** value instances, since all such member functions are already mapped as non-**const**.

The C++ signatures and memory management rules for **valuetype** operations (but not state member accessor and modifier functions) are identical to those described in “Argument Passing Considerations” on page 91 for client-side **interface** operations.

A static **_downcast** function is provided by each **valuetype** class to provide a portable way for applications to cast down the C++ inheritance hierarchy. This is especially required after an invocation of the **_copy_value** function (see “ValueBase and Reference Counting” on page 64). If a null pointer is passed to **_downcast**, it returns a null pointer.

Otherwise, if the **valuetype** instance pointed to by the argument is an instance of the **valuetype** class being downcast to, a pointer to the downcast-to class type is returned. If the **valuetype** instance pointed to by the argument is *not* an instance of the **valuetype** class being downcast to, a null pointer is returned. The `_downcast` function does not increment the reference count of the valuetype.

5.19.4 Valuetype Example

For example, consider the following IDL **valuetype**:

```
// IDL
valuetype Example {
        short op1();
        long op2(in Example x);
        private short val1;
        public long val2;

        private string val3;
        private float val4;
        private Example val5;
};
```

The C++ mapping for this **valuetype** is

```
// C++
class Example : public virtual ValueBase {
   public:
      virtual Short op1() = 0;
      virtual Long op2(Example*) = 0;

      virtual Long val2() const = 0;
      virtual void val2(Long) = 0;

      static Example* _downcast(ValueBase*);

   protected:
      Example();
      virtual ~Example();

      virtual Short val1() const = 0;
      virtual void val1(Short) = 0;

      virtual const char* val3() const = 0;
      virtual void val3(char*) = 0;
      virtual void val3(const char*) = 0;
      virtual void val3(const String_var&) = 0;

      virtual Float val4() const = 0;
      virtual void val4(Float) = 0;

      virtual Example* val5() const = 0;
      virtual void val5(Example*) = 0;

   private:
      // private and unimplemented
      void operator=(const Example&);
};

class OBV_Example : public virtual Example {
   public:
      virtual Long val2() const;
      virtual void val2(Long);

   protected:
      OBV_Example();
      OBV_Example(Short init_val1, Long init_val2,
               const char* init_val3, Float init_val4,
               Example* init_val5);
      virtual ~OBV_Example();

      virtual Short val1() const;
      virtual void val1(Short);

      virtual const char* val3() const;
      virtual void val3(char*);
      virtual void val3(const char*);
      virtual void val3(const String_var&);

      virtual Float val4() const;
      virtual void val4(Float);

      virtual Example* val5() const;
      virtual void val5(Example*);

      // ...
};
```

5.19.5 ValueBase and Reference Counting

The C++ mapping for the **ValueBase** IDL type serves as an abstract base class for all C++ `valuetype` classes. **ValueBase** provides several pure virtual reference counting functions inherited by all **valuetype** classes.

```
// C++
namespace CORBA {
   class ValueBase {
      public:
         virtual void _add_ref() = 0;
         virtual void _remove_ref() = 0;
         virtual ValueBase* _copy_value() = 0;
         virtual ULong _refcount_value() = 0;
```

```
            static ValueBase* _downcast(ValueBase*);

        protected:
            ValueBase();
            ValueBase(const ValueBase&);
            virtual ~ValueBase();

        private:
            void operator=(const ValueBase&);
        };
}
```

Operation	Description
`_add_ref`	Used to increment the reference count of a **valuetype** instance.
`_remove_ref`	Used to decrement the reference count of a **valuetype** instance and delete the instance when the reference count drops to zero. Note that the use of `delete` to destroy instances requires that all **valuetype** instances be allocated using `new`.
`_copy_value`	Used to make a deep copy of the **valuetype** instance. The copy has no connections with the original instance and has a lifetime independent of that of the original. Since C++ supports covariant return types, derived classes can override the `_copy_value` function to return a pointer to the derived class rather than `ValueBase*`, but since covariant return types are still not commonly supported by commercial C++ compilers, the return value of `_copy_value` can also be `ValueBase*`, even for derived classes. A compliant ORB implementation may use either approach. For now, portable applications will not rely on covariant return types and will instead use downcasting[a] to regain the most derived type of a copied **valuetype.**
`_refcount_value`	Returns the value of the reference count for the **valuetype** instance on which it is invoked.

a. The C++ dynamic_cast<> operator may also be used to cast down the value hierarchy, but it too is still not available in all C++ compilers and thus its use is still not portable at this time.

The names of these operations begin with underscore to keep them from clashing with user-defined operations in derived **valuetype** classes.

ValueBase also provides a protected default constructor, a protected copy constructor, and a protected virtual destructor. The copy constructor is protected to disallow copy construction of derived **valuetype** instances except from within derived class functions, and the destructor is protected to prevent direct deletion of instances of classes derived from **ValueBase**.

With respect to reference counting, **ValueBase** is intended to introduce only the reference counting interface. Depending upon the inheritance hierarchy of a **valuetype** class, its instances may require different reference counting mechanisms. For example, the reference counting mechanisms needed for a **valuetype** class that supports an **interface** are likely to be different from those needed for a regular concrete **valuetype** class, since the former has object adapter issues to consider. Therefore, **ValueBase** normally serves as a virtual base class multiply inherited into a **valuetype** class. One inheritance path is through the IDL inheritance hierarchy for the **valuetype**, since all **valuetype**s inherit from **ValueBase**, which provides the reference counting interface. The other inheritance path is through the reference counting implementation mix-in base class (see “Reference Counting Mix-in Classes” on page 66), which itself also inherits from **ValueBase**.

5.19.5.1 CORBA Module Additions

The C++ mapping also adds two additional reference counting functions to the **CORBA** namespace, as shown below:

```
// C++
namespace CORBA {
      void add_ref(ValueBase* vb)
      {
         if (vb != 0) vb->_add_ref();
      }

      void remove_ref(ValueBase* vb)
      {
         if (vb != 0) vb->_remove_ref();
      }

      // ...
}
```

These functions are provided for consistency with object reference counting functions. They are similar in that unlike the **_add_ref** and **_remove_ref** member functions, they can be called with null **valuetype** pointers. The **CORBA::add_ref** function increments the reference count of the **valuetype** instance pointed to by the function argument if non-null, or does nothing if the argument is a null pointer. The **CORBA::remove_ref** function behaves the same except it decrements the reference count. (The implementations shown above are intended to specify the required semantics of the functions, not to imply that conforming implementations must inline the functions.)

5.19.6 Reference Counting Mix-in Classes

The C++ mapping provides two standard reference counting implementation mix-in base classes:

1. **CORBA::DefaultValueRefCountBase**, which can serve as a base class for any application-provided concrete **valuetype** class whose corresponding IDL value type *does not* derive from any IDL **interface**s. For these types of **valuetype** classes, applications are also free to use their own reference-counting implementation mix-ins as long as they fulfill the **ValueBase** reference counting interface.

2. **PortableServer::ValueRefCountBase**, which *must* serve as a base class for any application-provided concrete **valuetype** class whose corresponding IDL **valuetype** *does* derive from one or more IDL **interface**s, and whose instances will be registered with the POA as servants. If IDL interface inheritance is present, but instances of the application-provided concrete **valuetype** class will not be registered with the POA, the **CORBA::DefaultValueRefCountBase** or an application-specific reference counting implementation mix-in may be used as a base class instead.

Each of these classes shall be fully concrete and shall completely fulfill the **ValueBase** reference counting interface, except that since they provide implementation, not interface, they shall not provide support for downcasting. In addition, each of these classes shall provide a protected default constructor that sets the reference count of the instance to one, a protected virtual destructor, and a protected copy constructor that sets the reference count of the newly-constructed instance to one. Just as with the **ValueBase** base class, the default assignment operator should be private and preferably unimplemented to completely disallow assignment.

Note that it is the application-supplied concrete **valuetype** classes that must derive from these mix-in classes, not the **valuetype** classes generated by the IDL compiler. This is to avoid the need to inherit these mix-ins as virtual bases, or to avoid inheriting multiple copies of the mix-ins (and thus multiple reference counts) if virtual bases are not employed. Also, only the final implementor of a **valuetype** knows whether it will ever be used as a POA servant or not, and thus the implementor must specify the desired reference counting mix-in.

5.19.7 Value Boxes

A value box class essentially provides a reference-counted version of its underlying type. Unlike normal **valuetype** classes, C++ classes for value boxes can be concrete since value boxes do not support methods, inheritance, or interfaces. Value box classes differ depending upon their underlying types.

To fulfill the **ValueBase** interface, all value box classes are derived from `CORBA::DefaultValueRefCountBase`.

5.19.7.1 Parameter Passing for Underlying Boxed Type

All value box classes provide `_boxed_in`, `_boxed_inout`, and `_boxed_out` member functions that allow the underlying boxed value to be passed to functions taking parameters of the underlying boxed type. The signatures of these functions depend on the parameter passing modes of the underlying boxed type. The return values of the `_boxed_inout` and `_boxed_out` functions shall be such that the boxed value is referenced directly, allowing it to be replaced or set to a new value. For example, invoking `_boxed_out` on a boxed string allows the actual string owned by the value box to be replaced.

```
// IDL
valuetype StringValue string;
interface X {
	void op(out string s);
};

// C++
StringValue* sval = new StringValue("string val");
X_var x = ...
x->op(sval->_boxed_out());        // boxed string is replaced
                                  // by op() invocation
```

Assume the implementation of `op` is as follows:

```
// C++
void MyXImpl::op(String_out s)
{
	s = string_dup("new string val");
}
```

The return value of the `_boxed_out` function shall be such that the string value boxed in the instance pointed to by `sval` is set to `"new string val"` after `op` returns, with the instance pointed to by `sval` maintaining ownership of the string.

5.19.7.2 Basic Types, Enums, and Object References

For all the signed and unsigned integer types except for the **fixed** type, and for **boolean**, **octet**, **char**, **wchar**, **float**, **double**, **long double**, and enumerated types, and for typedefs of all of these, value box classes provide:

- A public default constructor. Note that except for the object reference case, the value of the underlying boxed value will be indeterminate after this constructor runs (i.e., the default constructor does *not* initialize the boxed value to a given value). This is because the built-in constructors for each of the basic types and enumerations do not initialize instances of their types to particular values, either. For boxed object references, this constructor sets the underlying boxed object reference to nil.
- A public constructor that takes one argument of the underlying type. This argument is used to initialize the value of the underlying boxed type.
- A public assignment operator that takes one argument of the underlying type. This argument is used to replace the value of the underlying boxed type.
- Public accessor and modifier functions for the boxed value. The accessor and modifier functions are always named **`_value`**. For boxed object references, the return value of the accessor is not a duplicate.
- Explicit conversion functions that allow the boxed value to be passed where its underlying type is called for. These functions are named **`_boxed_in`**, **`_boxed_inout`**, and **`_boxed_out`**, and their return types match the **in**, **inout**, and **out** parameter passing modes, respectively, of the underlying boxed type. Implicit conversions to the underlying type are not provided because values are normally handled by pointer.
- A public copy constructor.
- A public static **`_downcast`** function.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

Value box classes for object references maintain a private managed copy of the object reference. The constructor, assignment operator, and **`_value`** modifier methods for these classes call **`_duplicate`** on the object reference argument; the destructor calls **`CORBA::release`** on the boxed reference.

An example value box class for an enumerated type is shown below:

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

// C++
class ColorValue : public DefaultValueRefCountBase {
   public:
      ColorValue();
      ColorValue(Color val);
      ColorValue(const ColorValue& val);

      ColorValue& operator=(Color val);

      Color _value() const;   // accessor
      void _value(Color val); // modifier

      // explicit conversion functions for
      // underlying boxed type
      //
      Color _boxed_in() const;
      Color& _boxed_inout();
      Color& _boxed_out();

      static ColorValue* _downcast(ValueBase* base);

   protected:
      ~ColorValue();

   private:
      void operator=(const ColorValue& val);
};
```

5.19.7.3 Struct Types

Value box classes for struct types map to classes that provide accessor and modifier functions for each struct member. Specifically, the classes provide:

- A public default constructor. The underlying boxed struct type is initialized as it would be by its own default constructor.
- A public constructor that takes a single argument of type `const T&`, where `T` is the underlying boxed struct type.
- A public assignment operator that takes a single argument of type `const T&`, where `T` is the underlying boxed struct type.
- Public accessor and modifier functions, all named `_value`, for the underlying boxed struct type. Two accessors are provided: one a const member function returning `const T&`, and the other a non-const member function returning a `T&`. The modifier function takes a single argument of type `const T&`.
- The `_boxed_in`, `_boxed_inout`, and `_boxed_out` functions that allow access to the boxed value to pass it in signatures expecting the underlying boxed struct type. The return values of these functions correspond to the **in**, **inout**, and **out** parameter passing modes for the underlying boxed struct type, respectively.
- For each struct member, a set of accessor and modifier functions. These functions have the same signatures as accessor and modifier functions for union members.
- A public copy constructor.
- A public static `_downcast` function.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

As with other value box classes, no implicit conversions to the underlying boxed type are provided since values are normally handled by pointer.

For example

```
// IDL
struct S {
      string str;
      long len;
};
valuetype BoxedS S;
```

```
// C++
class BoxedS : public DefaultValueRefCountBase {
   public:
      BoxedS();
      BoxedS(const S& val);
      BoxedS(const BoxedS& val);

      BoxedS& operator=(const S& val);

      const S& _value() const;
      S& _value();
      void _value(const S& val);

      const S& _boxed_in() const;
      S& _boxed_inout();
      S*& _boxed_out();

      static BoxedS* _downcast(ValueBase* base);

      const char* str() const;
      void str(char* val);
      void str(const char* val);
      void str(const String_var& val);

      Long len() const;
      void len(Long val);

   protected:
      ~BoxedS();

   private:
      void operator=(const BoxedS& val);
};
```

5.19.7.4 String and WString Types

In order to allow boxed strings to be treated as normal strings where appropriate, value box classes for strings provide largely the same interface as the `String_var` class. The only differences from the interface of the `String_var` class are:

- The value box class interface does not provide the `in`, `inout`, `out`, and `_retn` functions that `String_var` provides. Rather, the value box class provides replacements for these functions called `_boxed_in`, `_boxed_inout`, and `_boxed_out`. They have mostly the same semantics and signatures as their `String_var` counterparts, but their names have been changed to make it clear that they provide access to the underlying string, not to the value box itself.
- There are no overloaded operators for implicit conversion to the underlying string type because values are normally handled by pointer.

In addition to most of the `String_var` interface, value box classes for strings provide:

- Public accessor and modifier functions for the boxed string value. These functions are all named **`_value`**. The single accessor function takes no arguments and returns a **`const char*`**. There are three modifier functions, each taking a single argument. One takes a **`char*`** argument that is adopted by the value box class, one takes a **`const char*`** argument that is copied, and one takes a **`const String_var&`** from which the underlying string value is copied.
- A public default constructor that initializes the underlying string to an empty string.
- Three public constructors that take string arguments. One takes a **`char*`** argument that is adopted, one takes a **`const char*`** that is copied, and one takes a **`const String_var&`** from which the underlying string value is copied. If the **`String_var`** holds no string, the boxed string value is initialized to the empty string.
- Three public assignment operators: one that takes a parameter of type **`char*`** which is adopted, one that takes a parameter of type **`const char*`** that is copied, and one that takes a parameter of type **`const String_var&`** from which the underlying string value is copied. Each returns a reference to the object being assigned to. If the **`String_var`** holds no string, the boxed string value is set equal to the empty string.
- A public copy constructor.
- A public static **`_downcast`** function.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

An example of a value box class for a string is shown below.

// IDL
valuetype StringValue string;

```
// C++
class StringValue : public DefaultValueRefCountBase {
   public:
      // constructors
      //
      StringValue();
      StringValue(const StringValue& val);
      StringValue(char* str);
      StringValue(const char* str);
      StringValue(const String_var& var);

      // assignment operators
      //
      StringValue& operator=(char* str);
      StringValue& operator=(const char* str);
      StringValue& operator=(const String_var& var);

      // accessor
      //
      const char* _value() const;

      // modifiers
      //
      void _value(char* str);
      void _value(const char* str);
      void _value(const String_var& var);

      // explicit argument passing conversions for
      // the underlying string
      //
      const char* _boxed_in() const;
      char*& _boxed_inout();
      char*& _boxed_out();

      // ...other String_var functions such as overloaded
      // subscript operators, etc....

      static StringValue* _downcast(ValueBase* base);

   protected:
      ~StringValue();

   private:
      void operator=(const StringValue& val);
};
```

Note that even though value box classes for strings provide overloaded subscript operators, the fact that values are normally handled by pointer means that they must be dereferenced before the subscript operators can be used.

5.19.7.5 Union, Sequence, Fixed, and Any Types

Value boxes for these types map to classes that have exactly the same public interfaces as the underlying boxed types, except that each has:

- In addition to the constructors provided by the class for the underlying boxed type, a public constructor that takes a single argument of type `const T&`, where `T` is the underlying boxed type.
- An assignment operator that takes a single argument of type `const T&`, where `T` is the underlying boxed type.
- Accessor and modifier functions for the underlying boxed value. All such functions are named `_value`. There are two accessor functions, one a const member function returning a `const T&`, and the other a non-const member function returning `T&`. The modifier function takes a single argument of type `const T&`.
- The `_boxed_in`, `_boxed_inout`, and `_boxed_out` functions that allow access to the boxed value to pass it in signatures expecting the underlying boxed value type. The return values of these functions correspond to the **in**, **inout**, and **out** parameter passing modes for the underlying boxed type, respectively.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

As with other value box classes, no implicit conversions to the underlying boxed type are provided since values are normally handled by pointer.

Note that the value box class for sequence types provides overloaded subscript operators (`operator[]`) just as a sequence class does. However, since values are normally handled by pointer, the value instance must be dereferenced before the overloaded subscript operator can be applied to it.

Value box instances for the **any** type can be passed to the overloaded operators for insertion and extraction by invoking the appropriate explicit conversion function.

```
// C++
AnyValueBox* val = ...
val->_boxed_inout() <<= something;
if (val->_boxed_in() >>= something_else) ...
```

Below is an example value box along with its corresponding C++ class.

```
// IDL
typedef sequence<long> LongSeq;
valuetype LongSeqValue LongSeq;
```

```
// C++
class LongSeqValue : public DefaultValueRefCountBase {
   public:
      LongSeqValue();
      LongSeqValue(ULong max);
      LongSeqValue(ULong max,
               ULong length,
               Long* buf,
               Boolean release = 0);
      LongSeqValue(const LongSeq& init);
      LongSeqValue(const LongSeqValue& val);

      LongSeqValue& operator=(const LongSeq& val);

      const LongSeq& _value() const;
      LongSeq& _value();
      void _value(const LongSeq&);

      const LongSeq& _boxed_in() const;
      LongSeq& _boxed_inout();
      LongSeq*& _boxed_out();

      static LongSeqValue* _downcast(ValueBase*);

      ULong maximum() const;
      ULong length() const;
      void length(ULong len);

      Long& operator[](ULong index);
      Long operator[](ULong index) const;

   protected:
      ~LongSeqValue();

   private:
      void operator=(const LongSeqValue&);
};
```

5.19.7.6 Array Types

In order to allow boxed arrays to be treated as normal arrays where appropriate, value box classes for arrays provide largely the same interface as the corresponding array **`_var`** class. The only differences from the interface of the **`_var`** class are:

- The value box class interface does not provide the **`in`**, **`inout`**, **`out`**, and **`_retn`** functions that **`_var`** provides. Rather, the value box class provides replacements for these functions called **`_boxed_in`**, **`_boxed_inout`**, and **`_boxed_out`**. They have mostly the same semantics and signatures as their **`_var`** counterparts, but their names have been changed to make it clear that they provide access to the underlying array, not to the value box itself.
- There are no overloaded operators for implicit conversion to the underlying array type because values are normally handled by pointer.

In addition to most of the **`_var`** interface, value box classes for arrays provide:

- Public accessor and modifier functions for the boxed array value. These functions are named **`_value`**. The single accessor function takes no arguments and returns a pointer to array slice. The modifier function takes a single argument of type const array.
- A public default constructor.
- A public constructor that takes a const array argument.
- A public assignment operator that takes a const array argument.
- A public copy constructor.
- A public static **`_downcast`** function.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

An example of a value box class for an array is shown below.

```
// IDL
typedef long LongArray[3][4];
valuetype ArrayValue LongArray;

// C++
typedef Long LongArray[3][4];
typedef Long LongArray_slice[4];
class ArrayValue : public DefaultValueRefCountBase {
   public:
      ArrayValue();
      ArrayValue(const ArrayValue& val);
      ArrayValue(const LongArray val);

      ArrayValue& operator=(const LongArray val);

      const LongArray_slice* _value() const;
      LongArray_slice* _value();

      void _value(const LongArray val);
```

```
        // explicit argument passing conversions for
        // the underlying array
        //
        const LongArray_slice* _boxed_in() const;
        LongArray_slice* _boxed_inout();
        LongArray_slice* _boxed_out();

        // ...overloaded subscript operators...

        static ArrayValue* _downcast(ValueBase* base);

    protected:
        ~ArrayValue();

    private:
        void operator=(const ArrayValue& val);
};
```

Note that even though value box classes for arrays provide overloaded subscript operators, the fact that values are normally handled by pointer means that they must be de-referenced before the subscript operators can be used.

5.19.8 Abstract Valuetypes

Abstract IDL **valuetype**s follow the same C++ mapping rules as concrete IDL **valuetype**s, except that because they have no data members, the IDL compiler does not generate **OBV_** classes for them.

5.19.9 Valuetype Inheritance

For an IDL **valuetype** derived from other **valuetype**s or that supports **interface** types, several C++ inheritance scenarios are possible:

- *Concrete value base classes* are inherited as public virtual bases to allow for “ladder style” implementation inheritance.
- *Abstract value base classes* are inherited as public virtual base classes, since they may be multiply inherited in IDL.
- *Interface classes* supported by the IDL **valuetype** are not inherited. Instead, the operations on the interface (and base interfaces, if any) are mapped to pure virtual functions in the generated C++ base value class. In addition to this abstract base value class and the **OBV_** class, the IDL compiler generates a POA skeleton for this value type; the name of this skeleton is formed by prepending the string “POA_” to the fully-scoped name of the **valuetype**. The base value class and the POA skeleton of the interface type are public virtual base classes of this skeleton. No tie skeleton class is generated for the valuetype because the tie for the supported class can be used instead.

The reason that **interface** classes are not inherited is that **valuetype** instances, like POA servants, are themselves *not* object references. Providing this inheritance would allow for error-prone code that implicitly widened pointers to **valuetype** instances to C++ object references for the supported interfaces, but without first obtaining valid object references for those **valuetype** instances from the POA. When such an application attempted to use an invalid object reference obtained in this manner, it would encounter errors that could be difficult to track back to the implicit widening of the pointer to **valuetype** to object reference. The C++ language provides no hooks into the implicit pointer-to-class widening mechanism by which an application might guard against this type of error.

Avoiding the derivation of **valuetype** classes from **interface** classes also separates the lifetimes of **valuetype** instances from the lifetimes of object reference instances. It would be surprising to an application if a valid object reference that had not yet been released unexpectedly became invalid because another part of the program had decremented the **valuetype** part of the object reference instance to zero. This scenario could be solved by the provision of an appropriate reference counting mix-in class. However, given that such an approach breaks local/remote transparency by having object reference release operations affect the servant, and given the associated problems described in the preceding paragraphs, deriving **valuetype** classes from **interface** classes is best avoided.

An example of the mapping for a **valuetype** that supports an **interface** is shown below.

```
// IDL
interface A {
        void op();
};

valuetype B supports A {
        public short data;
};
```

```
// C++
class B : public virtual ValueBase {
   public:
      virtual void op();

      virtual Short data() const = 0;
      virtual void data(Short) = 0;

      // ...
};

class POA_B : public virtual POA_A, public virtual B {
   public:
      virtual void op();
      // ...
};
```

5.19.10 Valuetype Factories

Because concrete **valuetype** classes are provided by the application developer, the creation of values is problematic under certain circumstances. These circumstances include:

- *Unmarshaling*. The ORB cannot know *a priori* about all potential concrete value classes supplied by the application, and so the ORB unmarshaling mechanisms do not possess the capability to directly create instances of those classes.
- *Component Libraries*. Portions of an application, such as parts of a framework, may be limited to only manipulating **valuetype** instances while leaving creation of those instances to other parts of the application.

5.19.10.1 ValueFactoryBase Class

Just as they provide concrete C++ **valuetype** classes, applications must also provide factories for those concrete classes. The base of all value factory classes is the C++ `CORBA::ValueFactoryBase` class:

```
// C++
namespace CORBA {
      class ValueFactoryBase;
      typedef ValueFactoryBase* ValueFactory;

      class ValueFactoryBase
      {
      public:
         virtual ~ValueFactoryBase();

         virtual void _add_ref();
         virtual void _remove_ref();

         static ValueFactory _downcast(ValueFactory vf);

      protected:
         ValueFactoryBase();

      private:
         virtual ValueBase* create_for_unmarshal() = 0;
      };
      // ...
}
```

The C++ mapping for the IDL **CORBA::ValueFactory** native type is a pointer to a **`ValueFactoryBase`** class, as shown above. Applications derive concrete factory classes from **`ValueFactoryBase`**, and register instances of those factory classes with the ORB via the **`ORB::register_value_factory`** function. If a factory is registered for a given value type and no previous factory was registered for that type, the **`register_value_factory`** function returns a null pointer.

When unmarshaling value instances, the ORB needs to be able to call up to the application to ask it to create those instances. Value instances are normally created via their type-specific value factories (see "Valuetype Factories" on page 76) so as to preserve any invariants they might have for their state. However, creation for unmarshaling is different because the ORB has no knowledge of application-specific factories, and in fact in most cases may not even have the necessary arguments to provide to the type-specific factories.

To allow the ORB to create value instances required during unmarshaling, the **`ValueFactoryBase`** class provides the **`create_for_unmarshal`** pure virtual function. The function is private so that only the ORB, through implementation-specific means (e.g., via a friend class), can invoke it. Applications are not expected to invoke the **`create_for_unmarshal`** function. Derived classes shall override the **`create_for_unmarshal`** function and shall implement it such that it creates a new value instance and returns a pointer to it. The caller assumes ownership of the returned instance and shall ensure that **`_remove_ref`** is eventually invoked on it. Since the **`create_for_unmarshal`** function returns a pointer to **`ValueBase`**, the caller may use the downcasting functions supplied by value types to downcast the pointer back to a pointer to a derived value type.

Once the ORB has created a value instance via the **`create_for_unmarshal`** function, it can use the value data member modifier functions to set the state of the new value instance from the unmarshaled data. How the ORB accesses the protected value data member modifiers of the value is implementation-specific and does not affect application portability.

The **ValueFactoryBase** uses reference counting to prevent itself from being destroyed while still in use by the application. A **ValueFactoryBase** initially has a reference count of one. Invoking **_add_ref** on a **ValueFactoryBase** increases its reference count by one. Invoking **_remove_ref** on a **ValueFactoryBase** decrements its reference count by one, and if the resulting reference count equals zero, **_remove_ref** invokes **delete** on its **this** pointer in order to destroy the factory. For ORBs that operate in multi-threaded environments, the implementations of **ValueFactoryBase::_add_ref** and **ValueFactoryBase::_remove_ref** are thread-safe.

When a **valuetype** factory is registered with the ORB, the ORB invokes **_add_ref** once on the factory before returning from **register_value_factory**. When the ORB is done using that factory, the reference count is decremented once. This can occur in any of the following circumstances:

- If the factory is explicitly unregistered via **unregister_value_factory**, the ORB invokes **_remove_ref** once on the factory.
- If the factory is implicitly unregistered due to **ORB::shutdown**, the ORB is responsible for invoking **_remove_ref** once on each registered factory.
- If the factory is replaced with a new invocation of **register_value_factory**, the previously registered factory is returned to the caller who assumes ownership of one reference to that factory. When the caller is done with the factory, it invokes **_remove_ref** once on that factory.

The caller of **lookup_value_factory** assumes ownership of one reference to the factory. When the caller is done with the factory, it invokes **_remove_ref** once on that factory.

The **_downcast** function on the factory allows the return type of the **ORB::lookup_value_factory** function to be downcast to a pointer to a type-specific factory (see "Valuetype Factories" on page 76). It is important to note that the return value of the factory **_downcast** does *not* become the memory management responsibility of the caller, and thus **_remove_ref** is not called on it.

5.19.10.2 ValueFactoryBase_var Class

For the convenience of automatically managing **valuetype** factory reference counts, the **CORBA** namespace provides the **ValueFactoryBase_var** class. This class behaves similarly to the **PortableServer::ServantBase_var** class for servant memory management (see "ServantBase_var Class" on page 122).

```
// C++
namespace CORBA
{
      class ValueFactoryBase_var
      {
      public:
         ValueFactoryBase_var() :_ptr(0) {}
         ValueFactoryBase_var(ValueFactoryBase* p)
            : _ptr(p) {}
         ValueFactoryBase_var(const ValueFactoryBase_var& b)
            : _ptr(b._ptr)
         {
            if (_ptr != 0) _ptr->_add_ref();
         }
         ~ValueFactoryBase_var()
         {
            if (_ptr != 0) _ptr->_remove_ref();
         }
         ValueFactoryBase_var&
         operator=(ValueFactoryBase* p)
         {
            if (_ptr != 0) _ptr->_remove_ref();
         _ptr = p;
            return *this;
         }
         ValueFactoryBase_var&
         operator=(const ValueFactoryBase_var& b)
         {
               if (_ptr != b._ptr) {
               if (_ptr != 0) _ptr->_remove_ref();
               if ((_ptr = b._ptr) != 0)
                  _ptr->_add_ref();
            }
            return *this;
         }
         ValueFactoryBase* operator->() const {return _ptr;}
         ValueFactoryBase* in() const { return _ptr; }
         ValueFactoryBase*& inout() { return _ptr; }
         ValueFactoryBase*& out()
         {
            if (_ptr != 0) _ptr->_remove_ref();
            _ptr = 0;
            return _ptr;
         }
         ValueFactoryBase* _retn()
         {
            ValueFactoryBase* retval = _ptr;
            _ptr = 0;
            return retval;
         }

      private:
         ValueFactoryBase* _ptr;
      };
      // ...
}
```

The implementation shown above for the **`ValueFactoryBase_var`** is intended only as an example that conveys required semantics. Variations of this implementation are conforming as long as they provide the same semantics as the implementation shown here.

5.19.10.3 Type-Specific Value Factories

All **valuetype**s that have initializer operations declared for them also have type-specific C++ value factory classes generated for them. For a **valuetype A**, the name of the factory class, which is generated at the same scope as the value class, shall be **`A_init`**. Each initializer operation maps to a pure virtual function in the factory class, and each of these initializers defined in IDL is mapped to an initializer function of the same name. Base **valuetype** initializers are not inherited, and so do not appear in the factory class. The initializer parameters are mapped using normal C++ parameter passing rules for **in** parameters. The return type of each `initializer` function is a pointer to the created **valuetype**.

For example, consider the following **valuetype**:

```
// IDL
valuetype V {
        factory create_bool(boolean b);
        factory create_(char c);
        factory create_(octet o);
        factory create_(short s, string p);
        ...
};
```

The factory class for the example given above will be generated as follows:

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

      virtual V*
      create_bool(Boolean val) = 0;

      virtual V* create_char(Char val) =0;
      virtual V* create_octet(Octet val)=0;
      virtual  V* create_other(Short s, const char* p) = 0;

      static V_init* _downcast(ValueFactory vf);

   protected:
      V_init();
};
```

Each generated factory class has a public virtual destructor, a protected default constructor, and a public `_downcast` function allowing downcasting from a pointer to the base `ValueFactoryBase` class. Each also supplies a public pure virtual `create` function corresponding to each initializer. Applications derive concrete factory classes from these classes and register them with the ORB. Note that since each generated value factory derives from the base `ValueFactoryBase`, all derived concrete factory classes shall also override the private pure virtual `create_for_unmarshal` function inherited from `ValueFactoryBase`.

For **valuetype**s that have no operations or initializers, a concrete type-specific factory class is generated whose implementation of the `create_for_unmarshal` function simply constructs an instance of the `OBV_` class for the valuetype using new and the default constructor. The constructor for a concrete factory is public, not protected.

For **valuetype**s that have operations, but no initializers, there are no type-specific abstract factory classes, but applications must still supply concrete factory classes. These classes, which are derived directly from `ValueFactoryBase`, need not supply `_downcast` functions[14], and only need to override the `create_for_unmarshal` function.

5.19.10.4 Unmarshaling Issues

When the ORB unmarshals a **valuetype** for a request handled via C++ static stubs or skeletons, it tries to find a factory for the **valuetype** via the **ORB::lookup_value_factory** operation. If the factory lookup fails, the client application receives a CORBA::MARSHAL exception. Thus, applications utilizing static stubs or skeletons must ensure that a valuetype factory is registered for every **valuetype** it expects to receive via static invocation mechanisms.

Because of their dynamic nature, applications using the DII or DSI are not expected to have compile-time information for all the **valuetype**s they might receive. For these applications, **valuetype** instances are represented as **`CORBA::Any`**, and so value factories are not required to be registered with the ORB to allow such **valuetype**s to be unmarshaled. However, value factories must be registered with the ORB and available for lookup if the application attempts extraction of the **valuetypes** via the statically-typed **`Any`** extraction functions. See “Extraction from any” on page 51 for more details.

5.19.11 Custom Marshaling

The C++ mappings for the IDL **CORBA::CustomerMarshal**, **CORBA::DataOutputStream**, and **CORBA::DataInputStream** types follow normal C++ **valuetype** mapping rules.

5.19.12 Another Valuetype Example

```
// IDL
valuetype Node {
        public long data;
        public Node next;
        void print();

        Node change(in Node inval,
        inout Node ioval,
        out Node outval);
};

// C++
class Node : public virtual ValueBase
{
   public:
      virtual Long data() const = 0;
      virtual void data(Long) = 0;

      virtual Node* next() const = 0;
      virtual void next(Node*) = 0;

      virtual void print() = 0;
      virtual Node* change(Node* inval,
                  Node*& ioval,
                  Node_out outval) = 0;
      static Node* _downcast(ValueBase*);
```

14. Since the factory class hierarchy has virtual functions in it, a C++ dynamic_cast can always be used to traverse the factory inheritance hierarchy, but it is not portable since all C++ compilers do not yet support it.

```
   protected:
      Node();
      virtual ~Node();

   private:
      // private and unimplemented
      void operator=(const Node&);
};

class OBV_Node : public virtual Node
{
   public:
      virtual Long data() const;
      virtual void data(Long);

      virtual Node* next() const;
      virtual void next(Node*);

   protected:
      OBV_Node();
      OBV_Node(Long data_init, Node* next_init);
      virtual ~OBV_Node();

   private:
      // private and unimplemented
      void operator=(const OBV_Node&);
};
```

5.19.13 Valuetype Members of Structs

As described in “Mapping for Structured Types” on page 20, struct members are required to be self-managing. This results in the need for manager types for both strings and object references. Since **valuetype**s are handled by pointer, similar to the way strings and object references are handled, they too require manager types to represent them when they are used as struct members.

The **valuetype** instance manager types have semantics similar to that of the manager types for object references:

- Any assignment to a managed **valuetype** member causes that member to decrement the reference count of the **valuetype** it is managing, if any.
- A **valuetype** pointer assigned to a managed **valuetype** member is adopted by the member.
- A **valuetype** `_var` assigned to a managed **valuetype** member results in the reference count of the instance being incremented. The `_var` types and **valuetype** member manager types follow the same rules for widening assignment that those for object references do.
- If the constructed type holding the managed **valuetype** member is assigned to another constructed type (for example, an instance of a struct with a **valuetype** member is assigned to another instance of the same struct), the reference count of the managed **valuetype** instance in the struct on the right-hand side of the assignment is incremented, while the reference count of the managed instance on the left-hand side is decremented. As usual in C++, assignment to self must be guarded against to avoid any mishandling of the reference count.

- When it is destroyed, the managed **valuetype** member decrements the reference count of the managed **valuetype** instance.

The semantics of **valuetype** managers described here provide for sharing of **valuetype** instances across constructed types by default. Each C++ `valuetype` also provides an explicit copy function that can be used to avoid sharing when desired.

5.20 Mapping for Abstract Interfaces

The C++ mapping for abstract interfaces is almost identical to the mapping for regular interfaces. Rather than defining a complete C++ mapping for abstract interfaces, which would only duplicate much of the specification of the mapping for regular interfaces found in "Mapping for Interfaces" on page 7, only the ways in which the abstract interface mapping differs from the regular interface mapping are described here.

5.20.1 Abstract Interface Base

C++ classes for abstract interfaces are not derived from the `CORBA::Object` C++ class. In IDL, abstract interfaces have no common base. However, to facilitate narrowing from an abstract interface base class down to derived abstract interfaces, derived interfaces, and derived **valuetype** types, all abstract interface base classes that have no other base abstract interfaces derive directly from `CORBA::AbstractBase`. This base class provides the following:

- a protected default constructor
- a protected copy constructor
- a protected pure virtual destructor
- a public static `_duplicate` function
- a public static `_narrow` function
- a public static `_nil` function

The `AbstractBase` class is shown below:

```
// C++
class AbstractBase;
typedef ... AbstractBase_ptr;// either pointer or class

class AbstractBase {
   public:
      static AbstractBase_ptr _duplicate(AbstractBase_ptr);
      static AbstractBase_ptr _narrow(AbstractBase_ptr);
      static AbstractBase_ptr _nil();

      Object_ptr _to_object();
      ValueBase* _to_value();

   protected:
      AbstractBase();
      AbstractBase(const AbstractBase& val);
      virtual ~AbstractBase() = 0;
};
```

The **_duplicate** function operates polymorphically over both object references and **valuetype** types. If an **AbstractBase_ptr** that actually refers to an object reference is passed to the **_duplicate** function, the object reference is duplicated and a duplicate object reference is returned. Otherwise, the argument refers to a **valuetype** instance, so the **_add_ref** function is called on the **valuetype** and the argument is returned. If the argument is a nil **AbstractBase_ptr**, the return value is nil.

The implementation of **AbstractBase::_narrow** merely passes its argument to **_duplicate** and uses the value it returns as its own return value. Strictly speaking, the **_narrow** function is not needed in the **AbstractBase** interface because it is of little use to narrow an **AbstractBase** to its own type, but it is required by all conforming implementations to make writing C++ templates that deal with abstract interfaces easier (**AbstractBase** does not present a special case).

As with regular object references, the **_nil** function returns a typed **AbstractBase** nil reference.

Both the **is_nil** and **release** functions in the **CORBA** namespace are overloaded to handle abstract interface references.

```
// C++
namespace CORBA {
      Boolean is_nil(AbstractBase_ptr);
      void release(AbstractBase_ptr);
}
```

These behave the same as their object reference counterparts. Note that **release** is expected to operate polymorphically over both **valuetype** types and object reference types. If its argument is nil, it does nothing. If its argument refers to a **valuetype** instance, it invokes **_remove_ref** on that instance. Otherwise, its argument refers to an object reference, on which it invokes **CORBA::release** for object references.

If the concrete type of an abstract interface instance is a normal object reference, the **_to_object** function returns a reference to that object, otherwise it returns a nil reference. If the concrete type is a valuetype, **_to_value** returns a pointer to that valuetype, otherwise it returns a null pointer. The caller of **_to_object** or **_to_value** is responsible for properly releasing the returned reference or pointer.

5.20.2 Client Side Mapping

The client side mapping for abstract interfaces is almost identical to the mapping for object references, except:

- C++ classes for abstract interfaces derive from **CORBA::AbstractBase**, not **CORBA::Object**.
- Because abstract interface classes can serve as base classes for application-supplied concrete **valuetype** classes, they shall provide a protected default constructor, a protected copy constructor, and a protected destructor (which is virtual by virtue of inheritance from **AbstractBase**).
- The mapping for object reference classes does not specify the type of inheritance used for base object reference classes. However, because abstract interfaces can serve as base classes for application-supplied concrete **valuetype** classes, which themselves can be derived from regular interface classes, abstract interface classes shall always be inherited as public virtual base classes.
- Normal **Any** insertion and extraction operators are generated for abstract interfaces. The **Any::to_object**, **Any::to_abstract_base**, and **Any::to_value** types can be used to extract the contents of an Any as a generic object reference, abstract object reference, or valuetype respectively.

Other than that, the mapping for abstract interfaces is identical to that for regular interfaces, including the provision of **_var** types, **_out** types, manager types for struct, sequence, and array members, identical memory management, and identical C++ signatures for operations.

Both interfaces that are derived from one or more abstract interfaces, and **valuetype**s that support one or more abstract interfaces support implicit widening to the **_ptr** type for each abstract interface base class. Specifically, the **T*** for **valuetype T** and the **T_ptr** type for interface type **T** support implicit widening to the **Base_ptr** type for abstract interface type **Base**. The only exception to this rule is for **valuetype**s that only support an abstract interface indirectly via support for a regular interface type (see "Valuetype Inheritance" on page 75). In this case, it is the object reference for the **valuetype**, not the pointer to the **valuetype**, that supports widening to the abstract interface base. If a valuetype supports an abstract interface directly (or inherits that support via derivation from another valuetype) and at the same time supports a normal interface that inherits from the same abstract interface, then either the valuetype pointer or the object reference may be widened to the abstract interface.

5.21 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 6). 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. String and wide string exception members must be initialized to the empty string (**""** and **L""**, respectively) by the default constructor for the exception.
- 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() const = 0;
      virtual const char * _name() const;
      virtual const char * _rep_id() const;
   protected:
      Exception();
      Exception(const Exception &);
      Exception &operator=(const Exception &);
};
```

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. This function 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** implements **_raise()** as follows:

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

For environments that do not support exception handling, please refer to “Without Exception Handling” on page 154 for information about the `_raise()` function.

The `_name()` function returns the unqualified (unscoped) name of the exception. The `_rep_id()` function returns the repository ID of the exception. The return value from `_name()` and `_rep_id()` must not be deallocated.

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();

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

      virtual void _raise() const = 0;

      CompletionStatus completed() const;
      void completed(CompletionStatus);
protected:
      SystemException();
      SystemException(const SystemException &);
      SystemException(ULong minor, CompletionStatus status);
      SystemException &operator=(const SystemException &);
};
```

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

Each specific system exception 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 downcasting within the exception hierarchy.

```
// C++
class UserException : public Exception
{
   public:
      static UserException *_downcast(Exception *);
      static const UserException *_downcast(
                          const Exception *
                      );

      virtual void _raise() const = 0;

      // ...
};

class SystemException : public Exception
{
   public:
      static SystemException *_downcast(Exception *);
      static const SystemException *_downcast(
                        const Exception *
               );

      virtual void _raise() const = 0;

      // ...
};
```

Each exception class supports an overloaded static member function named **`_downcast`**. The parameter to the **`_downcast`** calls is a pointer to a **`const`** or non-**`const`** instance of the base class **`Exception`**. If the parameter is a null pointer, the return type of **`_downcast`** is a null pointer. If the actual (run time) type of the parameter exception can be widened to the requested exception's type, then **`_downcast`** will return a valid pointer to the parameter **`Exception`**. Otherwise, **`_downcast`** will return a null pointer. The version of **`_downcast`** overloaded to take a pointer to a **`const Exception`** returns a pointer to **`const`** in order to preserve **`const`**-correctness.

Unlike the **`_narrow`** operation on object references, the **`_downcast`** operation on exceptions is equivalent to the C++ **`dynamic_cast`** operator in that it 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 **`_downcast`** is no longer valid. The semantics for **`_downcast`** are thus the same as for **valuetype** as described in "Valuetype Operations" on page 62.

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 downcasting 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 downcasting 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 downcast within the exception hierarchy.

Previous versions of this mapping provided support for downcasting via a static member function called **`_narrow`**, which had exactly the same semantics as **`_downcast`**. Due to confusion over memory management differences between object reference **`_narrow`** functions and exception **`_narrow`** functions, the exception **`_narrow`** function is now deprecated in favor of **`_downcast`**. Portable applications shall use **`_downcast`** for exception downcasting, not **`_narrow`**. ORB implementations that provide **`_narrow`** functions for exceptions for purposes of backwards compatibility shall provide overloaded **`_narrow`** functions for both **`const`** and non-**`const Exception*`**, same as for **`_downcast`**.

5.21.1 ostream Inserters

Conforming implementations shall provide ostream inserters with the following signatures:

```
// C++
ostream& operator<<(ostream &, const CORBA::Exception &);
ostream& operator<<(ostream &, const CORBA::Exception *);
```

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.

These inserters print information about an exception on an ostream. The format and amount of detail of the printed information is implementation dependent. To guarantee that applications can control formatting of exceptions by providing custom overloaded inserters for more derived exception types, a conforming implementation must never provide overloaded inserters for **`SystemException`**, **`UserException`** or other more derived exception types.

5.21.2 UnknownUserException

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.

5.21.3 Any Insertion and Extraction for Exceptions

Conforming implementations shall generate **`Any`** insertion and extraction operators (**`operator<<=`** and **`operator>>=`**, respectively) that allow all system and user exceptions to be correctly inserted into and extracted from **`Any`**. Both copying and non-copying forms of the **`Any`** insertion operator shall be provided for all system and user exceptions.

In addition, conforming mapping implementations must support **`Any`** insertion (but not extraction) for **`CORBA::Exception`**. This is required to allow DSI-based applications to catch exceptions as **`CORBA::Exception&`** and store them into a **`ServerRequest`**.

```
// C++
try {
      // ...
} catch (Exception& exc) {
      Any any;
      any <<= exc;
      server_request->set_exception(any);
}
```

Note that this shall result in both the **`TypeCode`** and value for the actual derived exception type being stored into the **`Any`**. Both copying and non-copying forms of **`Any`** insertion for **`CORBA::Exception`** shall be provided.

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

For applications using the DII or portable interceptors, it is useful to be able to extract system exceptions generically. The mapping provides the following operator to do this:

```
// C++
Boolean operator>>=(const SystemException*& se) const;
```

The operator returns true if the **Any** on which it is invoked contains a system exception and the ORB has static type information for the actual system exception contained in the **Any**. In that case, **se** points at the base part of the actual exception after the operator returns. If the ORB does not have static type information for the system exception, the operator returns true and **se** points at an instance of CORBA::UNKNOWN. Otherwise, the operator returns false and the value of **se** is unchanged.

5.22 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.

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.

Operation and attribute signatures do not have exception specifications.

```
// 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);
```

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 85 for more details.

5.23 Implicit Arguments to Operations

If an operation in an OMG IDL specification has a context specification, then a **`Context_ptr`** input parameter (see "Context" on page 109) 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 154.

5.24 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.

If strings or wide strings are passed as **inout** parameters, the callee may modify the contents of the string or wide string in place. However, if the new string or wide string is longer than the initial string or wide string, reallocation becomes necessary. For a new string or wide string that is shorter than the original string or wide string, reallocation may also be used to conserve memory. However, shortening the string or wide string by replacing a character that is part of the initial string or wide string with the appropriate NUL character is also legal.

For **inout** object references, reallocation is necessary whenever the callee needs to change the initial value of the reference. The example below illustrates this.

For **in valuetype**s, the callee shall receive a copy of each **valuetype** argument passed to it even if the caller and callee are collocated in the same process. The callee is allowed to invoke operations and modifier functions that modify the state of the **valuetype** instance, but the state of the caller's copy of that **valuetype** instance shall not be affected by the callee's state changes. This is required to preserve location transparency for interface operations.

For **inout valuetype**s, the callee may either modify the incoming **valuetype** instance, or may replace the incoming pointer with a pointer to a different **valuetype** instance. The callee shall invoke `_remove_ref` on the **valuetype** instance passed in before replacing it with a **valuetype** instance to be passed back out. The caller shall eventually invoke `_remove_ref` on the **valuetype** instance it receives back as either an **inout**, **out**, or return value.

The example below illustrates the replacement of **inout** arguments. For the operation `f`, `s1` is an **inout** string that is modified in place and whose length is not changed by the callee, `s2` is an **inout** string that is grown by the callee, `obj` is an **inout** object reference that is changed by the callee, val1 is an **inout valuetype** that is changed in place by the callee, and val2 is an **inout valuetype** that is replaced by the callee. The example code uses local `T_var` variables to ensure automatic deallocation, but explicit calls to `CORBA::string_free` and `CORBA::release` could have been used instead.

Example

```
// IDL
valuetype V { public long state; };
interface A {
        void f(inout string s1, inout string s2, inout A obj,
                inout V val1, inout V val2);
};
```

```
// C++
void Aimpl::f(char *&s1, char *&s2, A_ptr &obj,
            V *&val1, V *&val2)
{
      // Convert s1 to uppercase in place
      while (*s1 != '\0') to upper(*s1++);

      // Return a different string value for s2
      String_var s2_tmp = s2;
      s2 = string_dup("new s2");

      // Assign new value to obj
      A_ptr newobj = ...
      A_var obj_tmp = obj;
      obj = A::_duplicate(newobj);

      // Change value of val1 in place
      if (val1 != 0) val1->state(42);

      // Replace val2 entirely
      CORBA::remove_ref(val2);
      val2 = new MyVImpl(1234);
}
```

For parameters that are passed or returned as a pointer (`T*`) or reference to pointer (`T*&`), except for **valuetype**s, 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:

- **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. Furthermore, conforming applications may also pass and return null pointers for all **valuetype** parameters and return types, and may embed null **valuetype** pointers within constructed types that are passed as parameters or return values, such as structs, unions, arrays, sequences, `Any`, and other valuetypes.

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, except for **valuetype**s, allowing the passage of null pointers to or from an operation would project C++ semantics onto OMG IDL operations.[16] A compliant implementation is allowed but not required to raise a BAD_PARAM exception if it detects such an error.

5.24.1 Operation Parameters and Signatures

Table 5.3 displays the mapping for the basic OMG IDL parameter passing modes and return type according to the type being passed or returned, while Table 5.3 displays the same information for **`T_var`** types. Table 5.4 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 5.3, *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 5.3, and callees should treat their **`T_out`** parameters as if they were actually the corresponding underlying types shown in Table 5.3.

In Table 5.3, 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.

Table 5.3 - 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

16. 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 154 for more details.

Table 5.3 - Basic Argument and Result Passing *(Continued)*

Data Type	In	Inout	Out	Return
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[a]	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*[b]
array, variable	const array	array	array slice*&[2]	array slice*[2]
any	const any&	any&	any*&	any*
fixed	const fixed&	fixed&	fixed&	fixed
valuetype[c]	valuetype*	valuetype*&	valuetype*&	valuetype*

a. Including pseudo-object references.

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

c. Including value boxes.

Table 5.4 - T_var Argument and Result Passing[a]

Data Type	In	Inout	Out	Return
object reference var[b]	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
array_var	const array_var&	array_var&	array_var&	array_var
any_var	const any_var&	any_var&	any_var&	any_var
valuetype_var[c]	const valuetype_var&	valuetype_var&	valuetype_var&	valuetype_var

a. Fixed types have no corresponding_var type and are therefore not shown in this table.

b. Including pseudo-object references.

c. Including value boxes.

Table 5.5 and Table 5.6 describe the caller’s responsibility for storage associated with **inout** and **out** parameters and for return results.

Table 5.5 - 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
valuetype	7	7	7

.

Table 5.6 - 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.
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.
7	Caller allocates storage for the valuetype instance. For inout parameters, the caller provides an initial value; if the callee wants to reassign the inout pointer value to point to a different valuetype instance, it will first call _remove_ref on the original input valuetype. To continue to use a valuetype instance passed in as an inout after the invoked operation returns, the caller must first invoke _add_ref on the valuetype instance. The caller is responsible for invoking _remove_ref on all out and return valuetype instances. The reduction of reference counts via _remove_ref for all valuetype instances embedded in other structures is performed automatically by the structures themselves.

5.25 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 sub clause 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*.

5.26 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[17] serverless object types that are not otherwise necessarily allowed in OMG IDL.

As explained in “Mapping of Pseudo Objects to C++” on page 98, 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 sub clauses or the normal mapping rules described in this clause.

5.27 Mapping Rules

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

17. 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.

1. 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.
2. As with normal interfaces, assignment operators are not supported.
3. 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.

5.28 Relation to the C PIDL Mapping

All serverless object interfaces and declarations that rely on them have direct analogs in 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 sub clauses. Further details, including definitions of types referenced but not defined below, may be found in the relevant clauses of this specification.

Some of the pseudo-interfaces shown in this clause rely on a user-defined exception supplied in the **CORBA** module by ORB implementations. This exception is called Bounds and is defined as follows:

```
// IDL
module CORBA
{
        exception Bounds {};
        // ...
};
```

Note that this exception is not the same as the CORBA::TypeCode::Bounds exception.

5.29 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.

5.29.1 Environment Interface

```
// IDL
pseudo interface Environment
{
    attribute exception exception;
    void clear();
};
```

5.29.2 Environment C++ Class

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

5.29.3 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.

5.29.4 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.

5.30 NamedValue

NamedValue is used only as an element of **NVList** and for return values in the **CORBA::Object::create_request** operation. **NamedValue** maintains an (optional) name, an **any** value, and labeling 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**.

5.30.1 NamedValue Interface

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

5.30.2 NamedValue C++ Class

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

5.30.3 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.

5.30.4 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.

5.31 NVList

NVList is a list of **NamedValue**s. A new **NVList** is constructed using the **ORB::create_list** operation (see "ORB" on page 111). New **NamedValue**s may be constructed as part of an **NVList**, in any of these 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.

5.31.1 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);
    void remove(in unsigned long index) raises(Bounds);
};
```

5.31.2 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);
      void remove(ULong);
};
```

5.31.3 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.

5.31.4 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**.

5.32 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 115.

```
// 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() == 0) {
      *(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() == 0) {
      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 115:

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

5.32.1 Request Interface

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

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

5.32.2 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);
      void remove(ULong index);
};

class ContextList
{
   public:
      ULong count();
      void add(const char* ctxt);
      void add_consume(char* ctxt);
      const char* item(ULong index);
      void 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();
      void invoke();
      void send_oneway();
      void send_deferred();
      void get_response();
      Boolean poll_response();
};
```

5.32.3 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. If invoked before the request has completed, **get_response** blocks until the request completes; if invoked after the request has completed, **get_response** returns immediately. The **poll_response** operation returns immediately. A true return value indicates that the request has completed. A false return value indicates that the reply for the request is still outstanding.
- The **add_*_arg**, **set_return_type**, and **return_value** member functions are added as shortcuts for using the attribute-based accessors.

5.32.4 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.

5.33 Context

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

5.33.1 Context Interface

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

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

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

5.33.2 Context C++ Class

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

      void create_child(const char *, Context_out);

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

5.33.3 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**.
- The **release** operation frees child contexts.

5.33.4 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.

5.34 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 110.

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.

5.34.1 TypeCode Interface

The **TypeCode** IDL interface is fully defined in version 2.3 of *Common Object Request Broker Architecture, Interface Repository* clause, *The TypeCode Interface* sub clause and is thus not duplicated here.

5.34.2 TypeCode C++ Class

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

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

      TypeCode_ptr get_compact_typecode() 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;

      Visibility member_visibility(ULong index) const;
      ValueModifier type_modifier() const;
      TypeCode_ptr concrete_base_type() const;
};
```

5.34.3 Differences from C-PIDL

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

5.34.4 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.

5.35 ORB

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

5.35.1 ORB Interface

```
// IDL
pseudo interface ORB
{
        typedef sequence<Request> RequestSeq;
        string object_to_string(in Object obj);
        Object string_to_object(in string str);

        void create_list(in long count, out NVList new_list);
        void create_operation_list(in OperationDef oper, out NVList new_list);

        void create_named_value(out NamedValue nmval);
        void create_exception_list(out ExceptionList exclist);
        void create_context_list(out ContextList ctxtlist);

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

        void send_multiple_requests_oneway(in RequestSeq req);
        void send_multiple_requests_deferred(in RequestSeq req);
        boolean poll_next_response();
        void get_next_response(out Request req);

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

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

        typedef string ObjectId;
        typedef sequence<ObjectId> ObjectIdList;
        Object resolve_initial_references(in ObjectId id) raises(InvalidName);
        ObjectIdList list_initial_services();

        Policy create_policy(in PolicyType type, in any val)
            raises(PolicyError);
};
```

5.35.2 ORB C++ Class

```
// C++
class ORB
{
   public:
      class RequestSeq {...};
      char *object_to_string(Object_ptr);
      Object_ptr string_to_object(const char *);
      void create_list(Long, NVList_out);
      void create_operation_list(OperationDef_ptr, NVList_out);

      void create_named_value(NamedValue_out);
      void create_exception_list(ExceptionList_out);
      void create_context_list(ContextList_out);

      void get_default_context(Context_out);
      void create_environment(Environment_out);

      void send_multiple_requests_oneway(const RequestSeq&);

      void send_multiple_requests_deferred(const RequestSeq &);
      Boolean poll_next_response();
      void 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);

      typedef char* ObjectId;
      class ObjectIdList { ... };
      Object_ptr resolve_initial_references(const char* id);
      ObjectIdList* list_initial_services();

      Policy_ptr create_policy(PolicyType type, const Any& val);
};
```

5.35.3 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 154 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 104 for more details).

5.35.4 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 version 2.3 of *Common Object Request Broker Architecture*, *ORB Interface* clause, *ORB Initialization* sub clause.

```
// 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 is 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**&`.

5.36 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.

5.36.1 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);
        void create_request(
            in Context ctx,
            in Identifier operation,
            in NVList arg_list,
            in NamedValue result,
            out Request request,
            in Flags req_flags);

        void 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 get_policy(in PolicyType policy_type);
        DomainManagerList get_domain_managers();
        Object set_policy_overrides(in PolicyList policies,
                    in SetOverrideType set_or_add);
        Policy get_client_policy(in PolicyType      type);
        PolicyList get_policy_overrides(in PolicyTypeSeq      types);
        boolean validate_connection(out PolicyList      inconsistent_policies );
```

```
      string repository_id();
      Object get_component ();
      ORB get_ORB ();
};
```

5.36.2 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 “Request” on page 104 of this specification. The **is_nil** and **release** functions are provided in the `CORBA` namespace, as described in “Object Reference Operations” on page 9.

```
// 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);
      void _create_request(
         Context_ptr ctx,
         const char *operation,

         NVList_ptr arg_list,
         NamedValue_ptr result,
         Request_out request,
         Flags req_flags

      void _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_overrides(
         const PolicyList&,
         SetOverrideType);
      Policy_ptr _get_client_policy(PolicyType type);
      PolicyList* _get_policy_overrides(const PolicyTypeSeq& types);
      Boolean _validate_connection(PolicyList_out inconsistent_policies);
      char* _repository_id();
      Object_ptr _get_component();
      ORB_ptr _get_ORB();
};
```

5.37 Local Object

The C++ mapping of **LocalObject** is a class derived from `CORBA::Object` that is used as a base class for locality constrained object implementations. A locality constrained object is implemented by a class derived both from the class mapping the interface and from `CORBA::LocalObject`.

```
namespace CORBA
{
    class LocalObject : public virtual Object
    {
    public:
         virtual void _add_ref();
         virtual void _remove_ref();
         virtual ULong _refcount_value() const;

         // ...other pseudo ops not shown...

    protected:
         LocalObject();
         ~LocalObject();
    };
};
```

Member functions and any data members needed to implement the **Object** pseudo-operations and any other ORB support functions shall also be supplied but are not shown.

The IDL compiler will generate appropriate conversion operations to allow a pointer to a local object implementation to automatically be converted to the corresponding _ptr or _var type.

Local object instances implement reference counting to prevent themselves from being destroyed while the application is still using them. The constructor and copy constructor initialize the reference count member to one. The assignment operator returns `*this` and does not affect the reference count.

`_refcount_value` returns the current value of the reference count member.

`_add_ref` increments the reference count member by one.

`_remove_ref` decrements the reference count member by one; if the resulting reference count equals ezero, `_remove_ref` invokes `delete` on its this pointer in order to destroy the local object.

For ORBs that operate in multi-threaded environments, the implementations of `_refcount_value`, `_add_ref`, and `_remove_ref` shall be thread-safe.

Local objects can be allocated on the stack even though they are reference-counted: because the constructor sets the initial reference count to one, and the program makes an equal number of calls to **`_add_ref`** and **`_remove_ref`**, when the local object is popped off the stack, the destructor simply destroys a local object with a reference count of one (that is, the reference count never drops to zero).

Note that reference counting can be disabled completely by providing no-op implementations of **`_add_ref`** and **`_remove_ref`** in the derived local object implementation.

Here is an example of how to implement a local interface.

```
// IDL
local interface LocalIF {
  void an_op(in long an_arg);
};
```

```
// C++
class MyLocalIF : public LocalIF, public CORBA::LocalObject {
public:
    MyLocalIF(...);
    ~MyLocalIF();

    void an_op(CORBA::Long an_arg);
};
```

5.38 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.

5.39 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). 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.

5.39.1 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();

         virtual POA_ptr _default_POA();

         virtual InterfaceDef_ptr
         _get_interface();

         virtual Boolean
         _is_a(const char* logical_type_id);

         virtual Boolean
         _non_existent();

         virtual void _add_ref();
         virtual void _remove_ref();
         virtual ULong  _refcount_value();

      protected:
         ServantBase() : _ref_count(1) {}
         ServantBase(const ServantBase &) : _ref_count(1) {}
         ServantBase& operator=(const ServantBase&);
         // ...all other constructors...
      private:
         ULong _ref_count;
      };
      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 protected 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 default implementation of the **_default_POA** 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.

ServantBase provides default implementations of the **_get_interface**, **_is_a**, and **_non_existent** object reference operations that can be overridden by derived servants if the default behavior is not adequate. The POA invokes these just like normal skeleton operations, thus allowing overriding definitions in derived servant classes to use **_this** and the **PortableServer::Current** interface within their function bodies.

For static skeletons, the default implementation of the **_get_interface** and **_is_a** functions provided by **ServantBase** use the interface associated with the skeleton class to determine their respective return values. For dynamic skeletons (see Section 5.42, “Mapping of DSI to C++,” on page 134), these functions use the **_primary_interface** function to determine their return values.

The default implementation of **_non_existent** simply returns false.

Servant instances implement reference counting to prevent themselves from being destroyed while the application is still using them. The constructor and copy constructor initialize the **_ref_count** member to one. The assignment operator returns ***this** and does not affect the reference count. **_refcount_value** returns the current value of the **_ref_count** member. **_add_ref** increments the **_ref_count** member by one. **_remove_ref** decrements the **_ref_count** member by one; if the resulting reference count equals zero, **_remove_ref** invokes delete on its **this** pointer in order to destroy the servant. For ORBs that operate in multi-threaded environments, the implementations of **_refcount_value**, **_add_ref**, and **_remove_ref** shall be thread-safe.

```
// C++
void PortableServer::ServantBase::_add_ref()
{
   ++_ref_count;
}

void PortableServer::ServantBase::_remove_ref()
{
   if (--_ref_count == 0)
      delete this;
}
ULong PortableServer::ServantBase:: _refcount_value()
{
   return _ref_count;
}
```

Servants can be allocated on the stack even though they are reference-counted: because the constructor sets the initial reference count to one, and the ORB makes an equal number of calls to **_add_ref** and **_remove_ref**, when the servant is popped off the stack, the destructor simply destroys a servant with a reference count of one (that is, the reference count never drops to zero).

Note that reference counting can be disabled completely by providing no-op implementations of **_add_ref** and **_remove_ref** in the derived servant implementation.

5.39.2 Servant Reference Counting Mix-In

The **PortableServer** namespace provides a **RefCountServantBase** class. This class exists for backward compatibility reasons; its use is deprecated and the class will be removed in a future revision of the C++ mapping. The **RefCountServantBase** class is defined as follows:

```
// C++
namespace PortableServer
{
   struct RefCountServantBase {};
}
```

5.40 Servant Memory Management Considerations

Portable memory management of servants requires an exact specification of when and how a servant may be deleted:

- The POA ensures that a servant will not be deleted while invocations are currently outstanding on that servant by maintaining a reference to the servant until the invocations have completed. For example, the POA may increment the reference count of the servant before invoking the implementation (but after **`preinvoke`**) and decrement the reference count after the invocation (but before **`postinvoke`**).
- Beware that explicit deletion of a servant will cause memory access violations if that servant is still in use by some other part of the application. For example, if the same servant instance was obtained from **`POA::reference_to_servant`** or **`POA::id_to_servant`** (perhaps in another thread), the caller that obtained the servant instance may still be using it. Also, explicit deletion may cause problems if the same servant instance is registered in multiple POAs.

For each POA, **ServantActivator**, or **ServantLocator** operation that either passes a **`Servant`** as a parameter or returns a **`Servant`**, the following rules described caller and callee memory management responsibilities:

- **`ServantActivator::incarnate`**—returns a **`Servant`**. The POA may use this **`Servant`** until it is passed to **`etherealize`**.
- **`ServantActivator::etherealize`**—has an **in** **`Servant`** argument. The POA assumes that **`etherealize`** consumes the **`Servant`** argument, and does not access a **`Servant`** in any way after it has been passed to **`etherealize`**. A conforming implementation of **`etherealize`** may invoke **`_remove_ref`** on the **`Servant`**.
- **`ServantLocator::preinvoke`**—returns a **`Servant`**. The POA may use this **`Servant`** until it is passed to **`postinvoke`**.
- **`ServantLocator::postinvoke`**—has an **in** **`Servant`** argument. The POA assumes that **`postinvoke`** consumes the **`Servant`** argument, and does not access a **`Servant`** in any way after it has been passed to **`postinvoke`**. A conforming implementation invoke **`_remove_ref`** on the **`Servant`**.
- **`POA::get_servant`**—returns a **`Servant`**. The POA invokes **`_add_ref`** once on the **`Servant`** before returning it; the caller of **`get_servant`** is responsible for invoking **`_remove_ref`** once on the returned **`Servant`** when it is finished with it.
- **`POA::set_servant`**—has an **in** **`Servant`** argument. The implementation of **`set_servant`** will invoke **`_add_ref`** at least once on the **`Servant`** argument before returning. When the POA no longer needs the **`Servant`**, it will invoke **`_remove_ref`** on it the same number of times.
- **`POA::activate_object`**—has an **in** **`Servant`** argument. The implementation of **`activate_object`** will invoke **`_add_ref`** at least once on the **`Servant`** argument before returning. When the POA no longer needs the **`Servant`**, it will invoke **`_remove_ref`** on it the same number of times.
- **`POA::activate_object_with_id`**—has an **in** **`Servant`** argument. The implementation of **`activate_object_with_id`** will invoke **`_add_ref`** at least once on the **`Servant`** argument before returning. When the POA no longer needs the **`Servant`**, it will invoke **`_remove_ref`** on it the same number of times.

- **POA::servant_to_id**—has an **in** **Servant** argument. If this operation causes the object to be activated, **_add_ref** is invoked at least once on the **Servant** argument before returning. Otherwise, the POA does not increment or decrement the reference count of the **Servant** passed to this function.
- **POA::servant_to_reference**—has an **in** **Servant** argument. If this operation causes the object to be activated, **_add_ref** is invoked at least once on the **Servant** argument before returning. Otherwise, the POA does not increment or decrement the reference count of the **Servant** passed to this function.
- **POA::reference_to_servant**—returns a **Servant**. The POA invokes **_add_ref** once on the **Servant** before returning it; the caller of **reference_to_servant** is responsible for invoking **_remove_ref** once on the returned **Servant** when it is finished with it.
- **POA::id_to_servant**—returns a **Servant**. The POA invokes **_add_ref** once on the **Servant** before returning it; the caller of **id_to_servant** is responsible for invoking **_remove_ref** once on the returned **Servant** when it is finished with it.

The following operations do not receive or return **Servants** in their signatures, but have behavior that may require invocations of **_add_ref** or **_remove_ref**:

- **_this**—invoked on a **Servant** to obtain an object reference for an object implemented by that **Servant**. If this operation causes the object to be activated, **_add_ref** is invoked at least once on the **Servant** argument before returning. Otherwise, the POA does not increment or decrement the reference count of the **Servant** passed to this function.
- **POA::deactivate_object**—upon activation, **_add_ref** is invoked on the **Servant**. Therefore, the act of deactivation must cause **_remove_ref** to be invoked. If the POA has no **ServantActivator** associated with it, the POA implementation calls **_remove_ref** when all operation invocations have completed. If there is a **ServantActivator**, the Servant is consumed by the call to **ServantActivator::etherealize** instead.
- **POA::destroy**—upon activation of a servant or registration of a default servant, **_add_ref** is invoked on the **Servant**. Therefore, the destruction of a POA must cause **_remove_ref** to be invoked. The POA implementation invokes **_remove_ref** on any default servant. If the POA has no **ServantActivator** associated with it, the POA implementation calls **_remove_ref** on each **Servant** in the Active Object Map when all operation invocations have completed. If there is a **ServantActivator**, each **Servant** is consumed by the call to **ServantActivator::etherealize** instead.
- **POAManager::deactivate**—upon activation of a servant or registration of a default servant, **_add_ref** is invoked on the **Servant**. Therefore, the destruction of a POA must cause **_remove_ref** to be invoked. If **etherealize_objects** is true, the POA implementation invokes **_remove_ref** on any default servant. If **etherealize_objects** is true and a managed POA does not have a **ServantActivator** associated with it, the POA implementation invokes **_remove_ref** on each Servant in that POA's Active Object Map after all dispatched operations have completed. If there is a **ServantActivator**, each **Servant** is consumed by the call to **ServantActivator::etherealize** instead.

Note that in those cases where the caller becomes responsible for invoking **_remove_ref** on a **Servant** returned to it, the caller can assign the return value to a **ServantBase_var** instance for automatic reference count management.

5.40.1 ServantBase_var Class

For the convenience of automatically managing servant reference counts, the **PortableServer** namespace also provides the **ServantBase_var** class. This class behaves similarly to _var classes for object references (see "Object Reference Types" on page 7). Class **ServantBase_var** is a type definition of the **Servant_var** template for type **ServantBase**:

```
// C++
namespace PortableServer
{
   class ServantBase { /* ... */ };
   typedef Servant_var<ServantBase> ServantBase_var;
}
```

The definition of the **Servant_var** template is as follows:

```
// C++
namespace PortableServer
{
   template<typename Servant>
   class Servant_var
   {
   protected:
      void swap(Servant* lhs, Servant* rhs)
      {
         Servant *tmp = lhs;
         lhs = rhs;
         rhs = tmp;
      }

   public:
      Servant_var() : _ptr(0) {}
      Servant_var(Servant* p) : _ptr(p) {}
      Servant_var(const Servant_var& b)
         : _ptr(b._ptr)
      {
         if (_ptr != 0) _ptr->_add_ref();
      }
      ~Servant_var()
      {
         if (_ptr != 0) {
            try {
               _ptr->_remove_ref();
            } catch (...) {
               // swallow exceptions
            }
         }
      }

      Servant_var& operator=(Servant* p)
      {
         if (_ptr != p) {
            Servant_var<Servant> tmp = p;
            swap(_ptr, p);
         }
         return *this;
      }
      Servant_var&
      operator=(const Servant_var& b)
      {
         if (_ptr != b._ptr) {
            Servant_var<Servant> tmp = b;
            swap(_ptr, tmp._ptr);
         }
         return *this;
      }

      Servant* operator->() const { return _ptr; }

      Servant* in() const { return _ptr; }
      Servant*& inout() { return _ptr; }
      Servant*& out()
      {
         if (_ptr != 0) {
            Servant_var<Servant> tmp;
            swap(_ptr, tmp._ptr);
         }
         return _ptr;
      }
      Servant* _retn()
      {
         Servant* retval = _ptr;
         _ptr = 0;
         return retval;
      }

   private:
      Servant* _ptr;
   };
}
```

The implementation shown above for the **`ServantBase_var`** is intended only as an example that conveys required semantics. Variations of this implementation are possible as long as they provide the same semantics as the implementation shown here.

The **`Servant_var`** template can be used to write exception-safe and type-safe code for heap-allocated servants. For example:

```
Foo* some_function(/*...*/)
{
   Servant_var<Foo_impl> foo_servant = new Foo_impl;
   foo_servant->do_something();           // might throw...
   some_poa->activate_object_with_id(...);
   return foo_servant->_this();
}
```

5.40.2 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 used to create the dispatching POA.

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. The POA used for implicit activation is gotten by invoking **`_default_POA()`** on the servant.

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 was 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. The POA is gotten by invoking **`_default_POA()`** on the servant.

For example, for interface **A** defined as follows:

```
// 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();
```

5.40.3 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 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();
      virtual void op2(Long val);
      ...
};
```

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();
      void op2(Long val);
      ...
};
```

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 DSI to C++” on page 134.

For interfaces that inherit from one or more base interfaces, the generated POA skeleton class uses virtual inheritance:

```
// IDL
interface A { ... };
interface B : A { ... };
interface C : A { ... };
interface D : B, C { ... };

// C++
class POA_A : public virtual PortableServer::ServantBase
{ ... };
class POA_B : public virtual POA_A { ... };
class POA_C : public virtual POA_A { ... };
class POA_D : public virtual POA_B, public virtual POA_C
{ ... };
```

This guarantees that the POA skeleton class inherits only one version of each operation, and also allows optional inheritance of implementations. In this example, the implementation of interface **B** reuses the implementation of interface A:

```
// C++
class A_impl: public virtual POA_A { ... };
class B_impl: public virtual POA_B, public virtual A_impl
{};
```

For interfaces that inherit from an abstract interface, the POA skeleton class is also virtually derived directly from the abstract interface class, but with protected access:

```
// IDL
abstract interface A { ... };
interface B : A { ... };

// C++
class A { ... };
class POA_B : public virtual PortableServer::ServantBase,
      protected virtual A { ... };
```

The abstract interface is inherited with protected access to prevent accidental conversion of the POA skeleton pointer to an abstract interface reference, for ORBs that implement object references as pointers. This also allows implementation classes and valuetypes to share an implementation of the abstract interface:

```
// IDL
valuetype V : supports A { ... };

// C++
class MyA : virtual A { ... };
class MyB : public virtual POA_B, protected virtual MyA
{ ... };
class MyV : public virtual V, public virtual MyA { ... };
```

5.40.4 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 sub clause describes how this can be achieved in a type-safe manner using C++ templates.

For the examples in this sub clause, the OMG IDL interface from “Inheritance-Based Interface Implementation” on page 126 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()
      {
         return _ptr->op1();
      }
      void op2(Long val)
      {
         _ptr->op2(val);
      }

      // override ServantBase operations
      POA_ptr _default_POA()
      {
         if (!CORBA::is_nil(_poa)) {
            return PortableServer::POA::_duplicate(_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)
{
      _ptr->op2(val);
}
```

This implementation assumes that the tied object supports an **op2()** operation with the same signature. 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)
{
      _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.

For C++ implementations that do not support namespaces or the definition of template classes inside other classes, tie template classes must be defined at global scope. For these environments, the names of tie template classes shall be formed by "flattening" the normal tie name, i.e., replacing all occurrences of "**::**" with "_". For example, in such an environment the name of the tie template class for interface **A::B::C** would be **POA_A_B_C_tie**.

5.41 Implementing Operations

The signature of an implementation member function is the mapped signature of the OMG IDL operation.

For example

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

```
// C++
class MyA : public virtual POA_A
{
   public:
      void f();
      ...
};
```

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();
      void g();
   private:
      long x_;
};

void
MyA::f()
{
      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.

5.41.1 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
```

5.42 Mapping of DSI to C++

The *Common Object Request Broker Architecture (CORBA)* specification, *Dynamic Skeleton Interface* clause, *DSI: Language Mapping* sub clause contains general information about mapping the Dynamic Skeleton Interface to programming languages.

This sub clause 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++

5.42.1 Mapping of ServerRequest to C++

The **ServerRequest** pseudo object maps to a C++ class in the **CORBA** namespace that 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) are freed by the ORB rather than by the DIR.

5.42.2 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 becomes deallocated 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.

5.42.3 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:
         Object_ptr _this();
         virtual void invoke(
               ServerRequest_ptr request
               ) = 0;
         virtual 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. Requests for the standard object operations (**_get_interface**, **_is_a**, and **_non_existent**) do not call **invoke()**, but call the corresponding functions defined in **ServantBase** instead.

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 invoked only by the POA in the context of serving a CORBA request. Invoking this method in other circumstances may lead to unpredictable results.

5.43 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* ObjectId_to_wstring(const ObjectId&);

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

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.

5.44 Mapping for PortableServer::ServantManager

5.44.1 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`.

5.44.2 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.

5.44.3 Server Side Mapping for Abstract Interfaces

The only circumstances under which an IDL compiler should generate C++ code for abstract interfaces for the server side are when either an interface is derived from an abstract interface, or when a **valuetype** supports an abstract interface indirectly through one or more intermediate regular interface types. Abstract interfaces by themselves cannot be directly implemented or instantiated by portable applications. Because of this, standard C++ skeleton classes for abstract interfaces are not necessary.

5.45 C++ Definitions for CORBA

This sub clause provides a partial 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. Furthermore, in some cases these types do not define the complete interfaces of their IDL counterparts; if any type is missing one or more operations, those operations are assumed to follow normal C++ mapping rules for their signatures, parameter passing rules, memory management rules, etc.

5.45.1 Primitive Types

```
typedef unsigned char       Boolean;
typedef unsigned char       Char;
typedef wchar_t             WChar;
typedef unsigned char       Octet;
typedef short               Short;
typedef unsigned short      UShort;
typedef long                Long;
typedef ...                 LongLong;
typedef unsigned long       ULong;
typedef ...                 ULongLong;
typedef float               Float;
typedef double              Double;
typedef long double         LongDouble;

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;
```

5.45.2 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 * operator();
        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(const String_out& s);
        String_out& operator=(const 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&);
};
```

5.45.3 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.

5.45.4 Fixed Class

```
class Fixed
{
    public:
        // Constructors
        Fixed(int val = 0);
```

```
      Fixed(unsigned val);
      Fixed(Long val);
      Fixed(ULong val);
      Fixed(LongLong val);
      Fixed(ULongLong val);
      Fixed(Double val);
      Fixed(LongDouble val);
      Fixed(const Fixed& val);
      Fixed(const char *);
      ~Fixed();

      // Conversions
      operator LongLong() const;
      operator LongDouble() const;
      Fixed round(UShort scale) const;
      Fixed truncate(UShort scale) const;
      char *to_string() const;

      // Operators
      Fixed& operator=(const Fixed& val);
      Fixed& operator+=(const Fixed& val);
      Fixed& operator-=(const Fixed& val);
      Fixed& operator*=(const Fixed& val);
      Fixed& operator/=(const Fixed& val);

      Fixed& operator++();
      Fixed operator++(int);
      Fixed& operator--();
      Fixed operator--(int);
      Fixed operator+() const;
      Fixed operator-() const;
      Boolean operator!() const;

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

istream& operator>>(istream& is, Fixed& val);
ostream& operator<<(ostream& os, const Fixed& val);

Fixed operator + (const Fixed& val1, const Fixed& val2);
Fixed operator - (const Fixed& val1, const Fixed& val2);
Fixed operator * (const Fixed& val1, const Fixed& val2);
Fixed operator / (const Fixed& val1, const Fixed& val2);

Boolean operator > (const Fixed& val1, const Fixed& val2);
Boolean operator < (const Fixed& val1, const Fixed& val2);
Boolean operator >= (const Fixed& val1, const Fixed& val2);
Boolean operator <= (const Fixed& val1, const Fixed& val2);
Boolean operator == (const Fixed& val1, const Fixed& val2);
Boolean operator != (const Fixed& val1, const Fixed& val2);
```

5.45.5 Any Class

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

      Any &operator=(const Any&);

      // 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 n = FALSE) :
            val(s), bound(b), nocopy(n) {}
         from_string(const char* s, ULong b) :
            val(const_cast<char*>(s)), bound(b),
            nocopy(0) {}
         char *val;
         ULong bound;
         Boolean nocopy;
      };
      struct from_wstring {
         from_wstring(WChar* s, ULong b,
                  Boolean n = FALSE) :
            val(s), bound(b), nocopy(n) {}
         from_wstring(const WChar*, ULong b) :
            val(const_cast<WChar*>(s)), bound(b),
            nocopy(0) {}
         WChar *val;
         ULong bound;
         Boolean nocopy;
      };
      struct from_fixed {
         from_fixed(const Fixed& f, UShort d, UShort s)
            : val(f), digits(d), scale(s) {}
         const Fixed& val;
         UShort digits;
         UShort scale;
      };

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

      // 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_out obj) : ref(obj) {}
   Object_ptr &ref;
};
struct to_string {
   to_string(const char *&s, ULong b)
      : val(s), bound(b) {}
   const char *&val;
   ULong bound;

   // the following constructor is deprecated
   to_string(char *&s, ULong b) : val(s), bound(b) {}
};
struct to_wstring {
   to_wstring(const WChar *&s, ULong b)
      : val(s), bound(b) {}
   const WChar *&val;
   ULong bound;

   // the following constructor is deprecated
   to_wstring(WChar *&s, ULong b)
      : val(s), bound(b) {}
};
struct to_fixed {
   to_fixed(Fixed& f, UShort d, UShort s)
      : val(f), digits(d), scale(s) {}
   Fixed& val;
   UShort digits;
   UShort scale;
};
struct to_abstract_base {
   to_abstract_base(AbstractBase_ptr& base)
   : ref(base) {}
   AbstractBase_ptr& ref;
};
struct to_value {
   to_value(ValueBase*& base) : ref(base) {}
   ValueBase*& ref;
};

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;
      Boolean operator>>=(to_fixed) const;
      Boolean operator>>=(to_abstract_base) const;
      Boolean operator>>=(to_value) const;

      TypeCode_ptr type() const;
      void type(TypeCode_ptr);

   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;
};

void operator<<=(Any&, Short);
void operator<<=(Any&, UShort);
void operator<<=(Any&, Long);
void operator<<=(Any&, ULong);
void operator<<=(Any&, Float);
void operator<<=(Any&, Double);
void operator<<=(Any&, LongLong);
void operator<<=(Any&, ULongLong);
void operator<<=(Any&, LongDouble);
void operator<<=(Any&, const Any&);    // copying
void operator<<=(Any&, Any*);          // non-copying
void operator<<=(Any&, const char*);
void operator<<=(Any&, const WChar*);

Boolean operator>>=(const Any&, Short&);
Boolean operator>>=(const Any&, UShort&);
Boolean operator>>=(const Any&, Long&);
Boolean operator>>=(const Any&, ULong&);
Boolean operator>>=(const Any&, Float&);
Boolean operator>>=(const Any&, Double&);
Boolean operator>>=(const Any&, LongLong&);
Boolean operator>>=(const Any&, ULongLong&);
Boolean operator>>=(const Any&, LongDouble&);
Boolean operator>>=(const Any&, const Any*&);
Boolean operator>>=(const Any&, const char*&);
Boolean operator>>=(const Any&, const WChar*&);
```

5.45.6 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
};
```

5.45.7 Exception Class

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

      virtual void _raise() const = 0;
      virtual const char * _name() const;
      virtual const char * _rep_id() const;

   protected:
      Exception();
};
```

5.45.8 SystemException Class

```
// C++
enum CompletionStatus { COMPLETED_YES, COMPLETED_NO,
   COMPLETED_MAYBE };
class SystemException : public Exception
{
   public:
      ~SystemException();

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

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

      virtual void _raise() const = 0;

      static SystemException* _downcast(Exception*);
      static const SystemException* _downcast(
                        const Exception*
      );
   protected:
      SystemException();
      SystemException(const SystemException &);
      SystemException(ULong minor, CompletionStatus status);
      SystemException &operator=(const SystemException &);
};
```

5.45.9 UserException Class

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

      virtual void _raise() const = 0;

      static UserException* _downcast(Exception*);
      static const UserException* _downcast(
                        const Exception*
   );
   protected:
      UserException();
      UserException(const UserException &);
      UserException &operator=(const UserException &);
};
```

5.45.10 UnknownUserException Class

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

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

5.45.11 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(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(ORB_ptr);
      ...
}
```

5.45.12 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);
      void _create_request(
            Context_ptr ctx,
            const char *operation,
            NVList_ptr arg_list,
            NamedValue_ptr result,
            Request_out request,
            Flags req_flags
         );

      void _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_overrides(
      const PolicyList& policies,
      SetOverrideType set_or_add
   );
};
```

5.45.13 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();
};
```

5.45.14 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();
};
```

5.45.15 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);
      void remove(ULong);

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

5.45.16 ExceptionList Class

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

5.45.17 ContextList Class

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

5.45.18 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();

      void invoke();
      void send_oneway();
      void send_deferred();
      void get_response();
      Boolean poll_response();

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

5.45.19 Context Class

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

      void create_child(const char*, Context_out);
      void set_one_value(const char*, const Any&);
      void set_values(NVList_ptr);

      void delete_values(const char*);
      void get_values(const char*, Flags, const char*,
                  NVList_out);

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

5.45.20 TypeCode Class

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

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

      Visibility member_visibility(ULong index) const;
      ValuetypeModifier type_modifier() const;
      TypeCode_ptr concrete_base_type() const;

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

5.45.21 ORB Class

```
// C++
class ORB
{
   public:
      typedef sequence<Request_ptr> RequestSeq;
      char *object_to_string(Object_ptr);
      Object_ptr string_to_object(const char*);
      void create_list(Long, NVList_out);
      void create_operation_list(OperationDef_ptr,
                        NVList_out);
      void create_named_value(NamedValue_out);
      void create_exception_list(ExceptionList_out);
      void create_context_list(ContextList_out);
      void get_default_context(Context_out);
      void create_environment(Environment_out);
      void send_multiple_requests_oneway(const RequestSeq&);
      void send_multiple_requests_deferred(const RequestSeq&);
      Boolean poll_next_response();
      void get_next_response(Request_out);

      // Obtaining initial object references
      typedef char* ObjectId;
      class ObjectIdList {...};
      class InvalidName : public UserException {...};
      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 destroy ();
      void run();

      Boolean get_service_information(
               ServiceType svc_type,
               ServiceInformation_out svc_info);

      typedef char* ObjectId;
      class ObjectIdList { ... };
      Object_ptr resolve_initial_references(const char* id);
      ObjectIdList* list_initial_services();

      Policy_ptr create_policy(PolicyType type, const Any& val);

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

5.45.22 ORB Initialization

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

5.45.23 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&):
};
```

5.46 Alternative Mappings For C++ Dialects

5.46.1 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.

5.46.2 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 100, the `Environment` class supports the ability to access and modify the `Exception` it holds.

As shown in “Mapping for Exception Types” on page 85, 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 111 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.

5.47 C++ Keywords

Table 5.7 lists all C++ keywords from the ISO/IEC 14822:2011 (September 1, 2011), Standard for Programming Language C++.

Table 5.7 - **C++ Keywords**

and	and_eq	alignas	alignof	asm	auto
bitand	bitor	bool	break	case	catch
char	char16_t	char32_t	class	compl	const
constexpr	const_cast	continue	decltype	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	noexcept	not	not_eq	nullptr
operator	or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed	sizeof
static	static_assert	static_cast	struct	switch	template
this	thread_local	throw	true	true	typedef
typeid	typename	union	unsigned	using	virtual
void	volatile	wchar_t	while	xor	xor_eq