An OMG® C++11 Language Mapping Publication

C++11 Language Mapping

Version 1.5

OMG Document Number: ptc/2020-11-03

Release Date: November 2020

Standard document URL: https://www.omg.org/spec/CPP11/

USE OF SPECIFICATION – TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING

BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 109 Highland Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: http://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

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, under Documents, Report a Bug/Issue (http://issues.omg.org/issues/create-new-issue).

Table of Contents

Preface

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 the OMG website at:

http://www.omg.org/spec

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
109 Highland Avenue
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

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification via the report form at:

http://issues.omg.org/issues/create-new-issue

This page intentionally left blank.

1 Scope

The IDL 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++11 programming language.

2 Conformance

The C++11 mapping tries to avoid limiting the implementation freedoms of ORB developers. For each OMG IDL construct, the C++11 mapping explains the syntax and semantics of using the construct from C++11. A client or server program conforms to this mapping (is C++11 compliant) if it uses the constructs as described in the C++11 mapping clauses.

2.1 C++ Implementation Requirements

This mapping assumes that the target C++11 environment supports all the features described in C++11 as specified by the ISO/IEC 14882:2011 C++.

2.2 No Implementation Descriptions

This mapping does not contain implementation descriptions. It avoids details that would constrain implementations. Some examples show possible implementations, but these are not required implementations.

3 Normative References

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 of, any of these publications do not apply.

- OMG CORBA 3.3 Part 1 Interfaces specification (formal/2012-11-12): http://www.omg.org/spec/CORBA/3.3/

4 Symbols and Abbreviated Terms

List of symbols/abbreviations:

- ORB - Object Request Broker
- CORBA - Common Object Request Broker Architecture

5 Additional Information

5.1 Changes to Adopted OMG Specifications

None in this specification.

5.2 Acknowledgments

The following companies submitted this specification to the Object Management Group:

- Remedy IT

In addition, the following companies participated in the revision task forces:

- Object Computing, Inc. (OCI)
- ADLINK Technology, Ltd.
- Twin Oaks Computing
- Real-Time Innovations
- Northrup Grumman
- THALES
- MIT/Lincoln Laboratory
- Kongsberg Defence & Aerospace
- Micro Focus

6 C++11 Language Mapping Specification

6.1 IDL Type Traits

For any IDL defined type an IDL type trait shall be provided to facilitate template meta programming. The generic type trait for IDL type **`T`** shall be available as **`IDL::traits<T>`**. Dependent on the IDL type of T a set of members will be declared as part of the type trait. For any IDL type the following member types shall be defined.

Table 6.1: Default Member Types

Member	Definition
`value_type`	The template parameter (T)
`in_type`	Type to be used as in C++ type
`out_type`	Type to be used as out C++ type
`inout_type`	Type to be used as inout C++ type

For example, given an IDL type **`A`** the trait **`IDL::traits<A>::in_type`** will deliver the C++ type that must be used for an in argument of type **`A`**.

6.2 Anonymous IDL Types

Anonymous IDL types are deprecated by the IDL specification and are not supported in the IDL to C++11 language mapping specification. The IDL compiler that implements this mapping must throw an error when it detects an anonymous type.

6.3 Scoped Names

Scoped names in OMG IDL are specified by C++ scopes. These mappings allow the corresponding mechanisms in OMG IDL and C++11 to be used to build scoped names. For instance:

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

is mapped into:

```
// C++
namespace M
{
```

```
        class E {
        public:
                void L (int32_t _L);
                int32_t L () const;
                int32_t& L();
                // Other methods not shown
        };
};
```

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++ protected name 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. The complete list of C++ protected names can be found in Table 6.14.

6.4 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 6.6).

6.5 Mapping for Modules

As shown in 6.3, 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
};
```

6.6 Mapping for Basic Data Types

The basic data types have the mappings shown in Table 6.2.

Table 6.2: Basic Data Type Mappings

OMG IDL	**C++**	**Default value**
short	int16_t	0
long	int32_t	0
long long	int64_t	0
unsigned short	uint16_t	0
unsigned long	uint32_t	0
unsigned long long	uint64_t	0
float	float	0.0
double	double	0.0
long double	long double	0.0
char	char	0
wchar	wchar_t	0
boolean	bool	FALSE
octet	uint8_t	0

Each OMG IDL basic type is mapped to the listed C++ type as defined by C++11 or by the fixed-size integral types of the header <**`cstdint`**>.

6.7 Mapping for Interfaces

An interface is mapped to a C++ class that gives access to the types, constants, operations, and exceptions defined in the interface. This example shows the behavior of the mapping of an interface:

```
// IDL
interface A
{
        struct S { short field; };
        void op (in S data);
};
```

```
// C++
// Conformant uses
A::S s; // declare a struct variable
s.field(3); // field access
IDL::traits<A>::ref_type p (...); // ...somehow obtain an objref…
p->op (s);
```

6.7.1 Reference Types

Several OMG IDL types are mapped to so called reference types. A reference points to a valid object or a nil object. The reference types are available as final C++ templates directly or through their related type traits. Strong reference types are available as **`ref_type`** trait and have the semantics of a **`std::shared_ptr`**. It is illegal for compliant application code to create specializations of the reference types.

Related to the strong reference type a weak reference trait **`weak_ref_type`** has to be delivered which must behave as a **`std::weak_ptr`**. From a strong reference a weak reference can be obtained using the **`weak_reference()`** operation. This weak reference must be convertible to a regular reference using the **`lock()`** operation.

Conversions of references to **`void*`**, assigning pointers to references, retrieving pointers from references, arithmetic operations, and relational operations, including test for equality, are all non-compliant. It is allowed to compare a reference with the C++11 keyword **`nullptr`** or to use it through a **`bool`** conversion operator. Any other comparison should lead to a compile error.

A reference can only be created from a **`nullptr`**, another reference, or using the **`CORBA::make_reference<>`** template which must deliver **`std::make_shared`** semantics. Any other creation of a reference type is not allowed and should lead to a compile error. Declaring a reference and initializing it with its default constructor will result in a nil reference.

The strong and weak reference types are grouped together into a reference type trait. The strong and weak reference types also provide this trait as their **`traits_type`** traits.

For all reference type a namespace level **`swap`** must be provided to exchange the values of two references in an efficient manner.

6.7.2 Object Reference Types

The use of an interface type in OMG IDL denotes an object reference that has the semantics as described in 6.7.1. For an interface **A**, the reference type trait **`IDL::traits<A>`** is available. Its strong reference type is known as the **`IDL::traits<A>::ref_type`** trait (aka **`A::_ref_type`** trait). The object reference type trait itself is also known as **`IDL::traits<A>::traits_type`** (aka **`A::_traits_type`**).

An operation can be performed on an object reference by using an arrow (“->”) on the reference. 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 weak object reference has to be available as **`IDL::traits<A>::weak_ref_type`** (aka **`A::_weak_ref_type`**).

```
// C++
IDL::traits<A>::ref_type a;
IDL::traits<A>::ref_type p (...); // ...somehow obtain an objref…
a = p;

IDL::traits<A>::weak_ref_type w = a.weak_reference();
```

```
if (p == nullptr) // legal comparison
if (p != nullptr ) // legal comparison
if (p) // legal usage, true if p != nullptr
if (!p)// legal usage, true if p == nullptr
if (p == 0) // illegal comparison, should result in a compile error
if (p != 0) // illegal comparison, should result in a compile error
if (a == p) // illegal comparison, should result in a compile error
if (a != p) // illegal comparison, should result in a compile error
delete a; // illegal delete, should result in a compile error
```

6.7.3 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** to **A**
- **B** to **Object**

```
// C++
IDL::traits<B>::ref_type bp = …
IDL::traits<A>::ref_type ap = bp;    // implicit widening
IDL::traits<Object>::ref_type objp = bp;   // implicit widening
objp = ap;    // implicit widening
```

6.7.4 Object Reference Operations

Conceptually, the **Object** class in the **CORBA** module is the base interface type for all objects; therefore, any object reference can be widened to the type **IDL::traits<Object>::ref_type**.

6.7.5 Nil Object Reference

The mapping defines that a nil object reference is defined by nullptr. For any nil object reference A, the following call is guaranteed to return **true**:

```
// C++
bool true_result = (A == nullptr);
```

Any attempt to invoke an operation through a nil object reference should result in an INV_OBJREF exception.

6.7.6 Narrowing Object References

The object traits for type T define the method **IDL::traits<T>::narrow** to narrow an object reference. These methods return a new object reference given an existing reference. The narrow methods return a nil object reference if the given reference is nil. The parameter to the narrow methods accepts a reference of an object of any interface type (**IDL::traits<Object>::ref_type**). If the actual (runtime) type of the parameter object can be narrowed to the requested interface's type, then the operation will return a valid object reference; otherwise, the operation 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** variable called **ap**, then:

- **IDL::traits<A>::narrow(ap)** returns a valid object reference
- **IDL::traits<B>::narrow(ap)** returns a valid object reference
- **IDL::traits<C>::narrow(ap)** returns a valid object reference
- **IDL::traits<D>::narrow(ap)** returns a nil object reference

Narrowing to A, B, and C all succeed because the object supports all those interfaces. The **IDL::traits<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 **IDL::traits<B>::narrow** or **IDL::traits<C>::narrow**, a new object reference will be returned. A call to **IDL::traits<D>::narrow** will return a nil reference.

If successful, the narrow methods creates a new object reference and does not change the given object reference. The narrow operations can throw system exceptions.

6.7.7 Mapping for Operations and Attributes

An operation maps to a non-const virtual C++ function with the same name as the operation. Each read-write attribute maps to a pair of overloaded C++ virtual 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 with a return type of **void**; that is, there is no way to know by looking at the C++ signature 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++
IDL::traits<A>::ref_type a (...); // retrieve the reference from somewhere
a->f();
a->g();
int32_t const n = a->x();
a->x(n + 1);
```

6.7.8 Argument Passing Considerations

The mapping of parameter passing modes is focused at simplicity and ease of use. For all primitive types, enums, and reference types, an **`in`** argument **`A`** of type **`P`**, that argument is passed as **`P`**. For all other types, an in argument **`A`** of type **`P`** is passed as **`const P&`**. For an **`inout`** and **`out`** argument it is passed as **`P&`**. If we return a type of P, it is returned as **`P`**.

The following examples demonstrate the compliant behavior:

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

```
// C++
IDL::traits<A>::ref_type ARef = ... // Retrieve object reference
S s;
ARef->f(s);
// use s
ARef->f(s); // first result will be overwritten
```

```
// IDL
interface B {
        void a(out string s);
        void b(in string s);
        void c(inout string s);
};
```

```
// C++
IDL::traits<B>::ref_type BRef (...); // Retrieve object reference
std::string s;
for (int8_t i = 0; i < 10; i++)
{
       BRef->a(s);
       BRef->b(s);
       BRef->c(s);
}
```

6.7.9 Type trait

For an interface the following additional member types shall be available as part of its type trait.

Table 6.3: Interface Member Types

Member	Definition
`is_local`	`std::false_type` or `std::true_type` type indicating whether this interface is declared as local
`is_abstract`	`std::false_type` or `std::true_type` type indicating whether this interface is declared as abstract
`ref_type`	Strong reference type
`weak_ref_type`	Weak reference type

6.8 Mapping for Constants

OMG IDL constants are mapped directly to a C++11 constant definition.

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

interface A
{
        const float pi = 3.14159;
};
```

```
// C++
const std::string name{"testing"};

class A
{
public:
      static constexpr float pi{3.14159};
};
```

The mappings for wide character and wide string constants are 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

```
const std::wstring ws{L”Hello World”};
```

in C++11.

6.9 Mapping for Enums

An OMG IDL **enum** maps directly to the corresponding C++11 type definition. When an enum is used in a structured type, its default value is the first enum value specified.

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


// C++
enum class Color : uint32_t { red, green, blue };
```

6.10 Mapping for String Types

The OMG IDL unbounded string type is mapped to **`std::string`**. A bounded string type is mapped to a distinct type to differentiate from an unbounded string. This distinct type must deliver **`std::string`** semantics and support transparent conversion from bounded to unbounded and vice versa including support for move semantics.

Implementations of the bounded string are under no obligation to perform a bounds check on the type itself. 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 must (at run time) detect attempts to pass a string value that exceeds the bound as a parameter across an interface. It must raise a BAD_PARAM system exception to signal the error.

For an unbounded string the following additional member types shall be available as part of its type trait.

Table 6.4: Unbounded String Member Types

Member	Definition
`element_traits`	**`IDL::traits<>`** for the element of the string
`is_bounded`	**`std::false_type`** type indicating that this type is not bounded

For a bounded string the following additional member types shall be available as part of its type trait.

Table 6.5: Unbounded String Member Types

Member	Definition
`element_traits`	**`IDL::traits<>`** for the element of the string
`is_bounded`	**`std::true_type`** type indicating that this type is bounded
`bound`	**`std::integral_constant`** type of value type **`uint32_t`** indicating the bound of the string

6.11 Mapping for Wide String Types

The OMG IDL unbounded wide string type is mapped to **`std::wstring`**. A bounded wide string type is mapped to a distinct type to differentiate from an unbounded wide string. This distinct type must deliver **`std::wstring`** semantics and support transparent conversion from bounded to unbounded and vice versa including support for move semantics.

Implementations of the bounded wide string are under no obligation to perform a bounds check on the type itself. As a

result, the programmer is responsible for enforcing the bound of bounded wstrings at run time. Implementations of the mapping are under no obligation to prevent assignment of a string value to a bounded wstring type if the wstring value exceeds the bound. Implementations must (at run time) detect attempts to pass a wstring value that exceeds the bound as a parameter across an interface. It must raise a BAD_PARAM system exception to signal the error.

For an unbounded wide string the following additional member types shall be available as part of its type trait.

Table 6.6: Unbounded Wide String Member Types

Member	Definition
`element_traits`	**`IDL::traits<>`** for the element of the wide string
`is_bounded`	**`std::false_type`** type indicating that this type is not bounded

For a bounded wide string the following additional member types shall be available as part of its type trait.

Table 6.7: Unbounded String Member Types

Member	Definition
`element_traits`	**`IDL::traits<>`** for the element of the wide string
`is_bounded`	**`std::true_type`** type indicating that this type is bounded
`bound`	**`std::integral_constant`** type of value type **`uint32_t`** indicating the bound of the wide string

6.12 Mapping for Sequence Types

An unbounded sequence is mapped to a C++ **`std::vector`** or to a type that delivers **`std::vector`**'s semantics and supports transparent conversion to and from **`std::vector`**. A bounded sequence is mapped to a distinct type to differentiate from an unbounded sequence. This distinct type must deliver **`std::vector`** semantics and support transparent conversion from bounded to unbounded and vice versa including support for move semantics. As a result, the programmer is responsible for enforcing the bound of bounded sequences at run time. Implementations of the mapping are under no obligation to prevent assignment of a sequence to a bounded sequence type if the sequence exceeds the bound. Implementations must at run time detect attempts to pass a sequence that exceeds the bound as a parameter across an interface. When an implementation detects this error, it must raise a BAD_PARAM system exception to signal the error.

Additionally the C++ **`std::vector`** can have a size that is larger than a maximum size of an IDL sequence that is limited in length to the maximum of ULong. When this happens the implementation must raise a BAD_PARAM system exception to signal the error.

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
typedef sequence<V1> V3;         // sequence of sequences
```

```
// C++
using V1 = std::vector<T>;
using V2 = IDL::bounded_vector<T, 2>;
using V3 = std::vector<V1>;
```

For an unbounded sequence the following additional member types shall be available as part of its type trait.

Table 6.8: Unbounded Sequence Member Types

Member	Definition
element_traits	**IDL::traits<>** for the element of the sequence
is_bounded	**std::false_type** type indicating that this type is not bounded

For a bounded sequence the following additional member types shall be available as part of its type trait.

Table 6.9: Bounded Sequence Member Types

Member	Definition
element_traits	**IDL::traits<>** for the element of the sequence
is_bounded	**std::true_type** type indicating that this type is bounded
bound	**std::integral_constant** type of value type **uint32_t** indicating the bound of the sequence

6.13 Mapping for Array Types

Arrays are mapped to the corresponding C++ **std::array<>** definition, which allows the definition of statically-initialized data using the array.

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

```
// C++
using F = std::array<float, 10>;
using V = std::array<std::string, 10>;
using M = std::array<std::array<std::array<std::string, 3>, 2>, 1>;
F f1; F f2;
V v1; V v2;
```

```
M m1; M m2;
IDL::traits<Foo>::ref_type ref (..); // obtain an object reference
ref->op(f2, v2, m2);
f1[0] = f2[1];
v1[1] = v2[1];
m1[0][1][2] = m2[0][1][2];
```

For an array the following additional member types shall be available as part of its type trait.

Table 6.10: Array Member Types

Member	Definition
element_traits	**IDL::traits<>** for the element of the array
dimensions	**std::integral_constant** type of value_type **uint32_t** indicating the number of dimensions of the array

6.14 Mapping for Structured Types

The mapping for structured types is a C++ class with a default constructor, a copy constructor, a move constructor, an assignment copy operator, an assignment move operator, and a destructor. The default constructor initializes object reference members to appropriately-typed nil object references, basic datatypes to their default value as listed in Table 6.2, and enums to their first value. All other members are initialized using their default constructors. The copy constructor performs a deep-copy from the existing structure to create a new structure. The move constructor moves all arguments to their members. The copy assignment operator performs a deep-copy to create a new structure with strong exception safety, the move assignment operator performs a move of all members to the existing structure with strong exception safety. The destructor releases all members.

For each member a set of accessors must be provided. If we have a member **A** of type **T** in case **T** is a:

- short, long, long long, unsigned short, unsigned long, unsigned long long, float, double, long double, char, wchar, boolean, octet
- Enumeration
- Object reference (**IDL::traits<T>::ref_type**)
- Valuetype reference (**IDL::traits<T>::ref_type**)
- Typecode reference (**IDL::traits<CORBA::TypeCode>::ref_type**)
- Abstract base reference (**IDL::traits<T>::ref_type**)

The following set of accessors must be provided:

```
void A (T);
T A () const;
T& A ();
```

In all other cases the following set of accessors has to be provided for **T**.

```
void A (const T&);
void A (T&&);
```

```
      const T& A () const;
      T& A ();
```

First an accessor to set the value which accepts a const T&. Secondly an accessor that performs a move by accepting T&&. Thirdly an accessor that returns the member as const T& and as last an accessor that provides write access by returning the member as T&.

Mapping details of the various structured types are specified in the following subsections. Those sections may modify the general mappings of this section.

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

```
// C++
IDL::traits<Foo>::ref_type f_ref (...); // obtain a reference
S b;
f_ref->f(b);
S a = b; // deep-copy
cout << "names " << a.name() << ", " << b.name() << endl;
```

6.14.1 Mapping for Struct Types

An OMG IDL struct maps to a C++ class, with each OMG IDL struct member mapped to a set of corresponding member methods. In addition to the methods as described in 6.14, the class has an explicit constructor accepting values for each struct member in the order they are specified in IDL.

The following examples illustrate usage of a struct, using the following OMG IDL definition:

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

```
// C++
class Variable
{
public:
      Variable ();
      ~Variable ();
      Variable (const Variable&);
      Variable (Variable&&);
```

```
        Variable& operator= (const Variable& x);
        Variable& operator= (Variable&& x);
        explicit Variable (std::string name);
        void name (const std::string& _name);
        void name (std::string&& _name);
        const std::string& name () const;
        std::string& name ();
};

void swap (Variable& m1, Variable& m2);
```

A namespace level **swap** must be provided to exchange the values of two structs in an efficient manner.

6.14.2 Mapping for Union Types

A union maps to a C++ class with accessor functions for the union members and discriminant. For each member a set of accessors must be provided as described in 6.14.

The default union constructor initializes the union. If there is a default case specified, the union is initialized to this default case. In case the union has an implicit default member it is initialized to that case. In all other cases it is initialized to the first discriminant value specified in IDL. Assigning, copying, moving, 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.The copy constructor and copy assignment operator perform a deep-copy of their parameters. The move constructor and move assignment operator perform a move of their parameters. 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, moving, 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 with strong exception safety. Attempting to get a value through an accessor that does not match the current discriminant results in a BAD_PARAM exception. Union members associated with the "default" label and those associated with multiple labels may have multiple legal discriminator values. For these members, the modifiers (those that return void) have a second parameter which has the discriminator's type. This parameter has a default argument. For members associated with multiple labels, the value of the default argument is the value of the case label that appears first in IDL. For members associated with the default label, the value of the default argument is an implementation-defined value that selects the member. Calling a referent for a member that does not match the current discriminant results in a BAD_PARAM exception. The following example helps illustrate the mapping for union types:

```
// IDL
struct S { long len; };
interface A;
union U switch (long) {
        case 1: long x;
```

```
        case 2: string z;
        case 3:
        case 4: S w;
        default: A obj;
};


// C++
class S { ... };
class U
{
public:
      U();
      U(const U&);
      U(U&&)
      ~U();
      U &operator=(const U&);
      U &operator=(U&&);

      void _d(int32_t);
      int32_t _d() const;

      void x(int32_t);
      int32_t x() const;
      int32_t& x();

      void z(const std::string&);
      void z(std::string&&);
      const std::string& () z const;
      std::string& z();

      void w(const S&, int32_t = 3);
      void w(S&&, int32_t = 3);
      const S& w() const;
      S& w();

      void obj(IDL::traits<A>::ref_type, int32_t = impl_defined_constant);
      IDL::traits<A>::ref_type obj() const;
      IDL::traits<A>::ref_type& obj();
};


void swap (U& m1, U& m2);
```

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 or move of their parameters. Referents can be used for read-write access.

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.

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(3);                             // OK, member w selected
u._d(4);                             // OK, member w selected
u._d(1);                 // error, different member selected, results in BAD_PARAM
A a = ...;
u.obj(a);                // member obj selected
u._d(7);                 // OK, member obj selected
u._d(1);                 // error, different member selected, results in BAD_PARAM
s = u.w();               // error, member w not active, results in BAD_PARAM
u.w(s, 4);               // OK, member w selected and discriminator is 4
u.obj(a, 23);            // OK, member obj selected and discriminator is 23
u.obj(a, 2); // error, 2 is not a valid discriminator for obj, results in BAD_PARAM
u.x(0, 1);   // compile-time error, x only has 1 possible discriminator value
```

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
bool 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.

A namespace level **swap** must be provided to exchange the values of two unions in an efficient manner.

6.15 Mapping for Fixed Types

The C++11 mapping for fixed is defined by the following C++ class in the **IDL** namespace:

```
// C++

class Fixed final
{
```

```
public:
      // Constructors (described below)
      explicit Fixed(const std::string&);
      Fixed(const Fixed&);
      Fixed(Fixed&& val);
      ~Fixed();


      // Conversions
      explicit operator int64_t () const;
      explicit operator long double() const;
      Fixed round(uint16_t) const;
      Fixed truncate(uint16_t) const;
      std::string to_string() const;

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

      Fixed operator+() const;
      Fixed operator-() const;

      explicit operator bool() const;

      // Other member functions
      uint16_t fixed_digits() const;
      uint16_t fixed_scale() const;
};


std::string to_string(const Fixed&);
void swap(Fixed&, Fixed&);
std::istream& operator>>(std::istream& is, Fixed& val);
std::ostream& operator<<(std::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);
bool operator>(const Fixed& val1, const Fixed& val2);
bool operator<(const Fixed& val1, const Fixed& val2);
bool operator>=(const Fixed& val1, const Fixed& val2);
bool operator<=(const Fixed& val1, const Fixed& val2);
bool operator==(const Fixed& val1, const Fixed& val2);
bool operator!=(const Fixed& val1, const Fixed& val2);
```

In addition to the constructors listed above, the **`Fixed`** class is default constructible and has explicit constructors that can

be called with a single value of any of the C++ fundamental integer types, double, or long double. 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, which may be a template instantiation, is not defined by this mapping. Here is an example of the mapping:

```
// IDL
typedef fixed<5,2> F;
interface A
{
        void op(in F arg);
};
```

```
// C++
using F = IDL::Fixed_Or_Implementation_Defined_Type;
class A
{
public:
       …
       virtual 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(std::string&)** 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 **int64_t** and **long double** 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 **operator bool** will return **false** when the fixed value is zero, in all other cases it will return **true**.

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++
F f1 ("0.1");
F f2 (".05");
F 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 **fixed_digits** and **fixed_scale** functions return the smallest digits and scale value that can hold the complete fixed-point value.

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 3.2 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. 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.

For a fixed type the following additional member types shall be available as part of its type trait.

Table 6.11: Fixed Member Types

Member	Definition
digits	**std::integral_constant** type of value_type **uint16_t** indicating the number of digits of the fixed type
scale	**std::integral_constant** type of value_type **uint16_t** indicating the scale of the fixed type

6.16 Mapping for Typedefs

A typedef creates an alias for a type. The example below illustrates the mapping.

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

```
// C++
using T = int32_t;
// ...definitions for A1…
using A2 = A1;

// ...definitions for S1…
using S1 = std::vector<int32_t>;
using S2 = S1;
```

6.17 Mapping for the Any Type

The IDL **any** type is mapped to the CORBA::Any class. A C++ mapping for the OMG IDL type **any** must fulfill two different requirements:

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

- Handling values whose types are not known at 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.

6.17.1 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 have to be provided. 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** (see 6.22) 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.

6.17.2 Insertion into an any

To allow a value to be set in an **any** in a type-safe fashion, an 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, long, long long, unsigned short, unsigned long, unsigned long long, float, double, long double, char, wchar, boolean, octet
- Enumeration
- Object reference (**IDL::traits<T>::ref_type**)
- Valuetype reference (**IDL::traits<T>::ref_type**)
- Typecode reference (**IDL::traits<CORBA::TypeCode>::ref_type**)
- Abstract base reference (**IDL::traits<T>::ref_type**)

For values of type T that are too large to be passed by value efficiently, such as array, string, wstring, struct, union, sequence, Any, and exception, the following functions are provided.

```
// C++
void operator<<=(CORBA::Any&, const T&);  // copying insert
void operator<<=(CORBA::Any&, T&&);       // move insert
```

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

```
// C++
int32_t value = 42;
Any a;
a <<= value;
```

In this case, the version of **operator<<=** overloaded for type **int32_t** must be able to set both the value and the **TypeCode** properly for the **any** variable.

6.17.3 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++
bool operator>>=(const CORBA::Any&, T&);
```

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

```
// C++
int32_t value;
CORBA::Any a;

a <<= int32_t(42);
if (a >>= value) {
// ... use the value …
}
```

In this case, the version of **operator>>=** for type **int32_t** must be able to determine whether the Any truly does contain a value of type **int32_t** 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 **int32_t**, the value of the caller’s reference variable is not changed, and **operator>>=** returns **false**.

For example, consider the following IDL struct:

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

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

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

If the extraction is successful, the caller variable will contain the value that was stored by the **any**, and **operator>>=** will return **true**. If the extraction is not successful, the **operator>>=** returns **false**.

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

Any object reference should be extractable from an **Any** as a base object reference. Any abstract reference should be extractable from an **Any** as a base object reference or as a base valuetype reference. any valuetype reference should be extractable from an **Any** as a base valuetype reference.

6.17.4 TypeCode Replacement

The **type** accessor function returns a **TypeCode** reference to the **TypeCode** associated with the **Any**.

```
IDL::traits<CORBA::TypeCode>::ref_type type() const;
```

Because C++ **typedefs** 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;
IDL::traits<CORBA::TypeCode>::ref_type 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 **int32_t.** Therefore, when the value is inserted, standard C++ overloading mechanisms cause the insertion operator for **int32_t** to be invoked. In fact, because **LongType** is an alias for **int32_t,** 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(IDL::traits<CORBA::TypeCode>::ref_type);
```

Revising the previous example:

```
// C++
Any any;
LongType val = 1234;
any <<= val;
any.type(_tc_LongType);                   // replace TypeCode
IDL::traits<CORBA::TypeCode>::ref_type 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.

6.18 Mapping for Valuetypes

The IDL **valuetype** has features that make its C++11 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.

An IDL **valuetype** is mapped to the C++ trait **`IDL::traits<>::base_type`**. This trait relates to 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**.

The C++ **`IDL::traits<>::obv_type`** trait is provided for referring to the OBV class that 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 **`IDL::traits<>::obv_type`** trait.

In C++11 **valuetypes** map to so called valuetype references that behave as reference type as described in 6.7.1. The reference type trait **`IDL::traits<>`** is available for each valuetype. The strong reference is delivered as **`IDL::traits<>::ref_type`** trait and the weak reference as **`IDL::traits<>::weak_ref_type`** trait.

All **init** initializers declared for a **valuetype** are mapped to pure virtual functions on a separate abstract C++ factory class. This class is available through the **`IDL::traits<>::factory_type`** trait.

6.18.1 Valuetype Data Members

The C++ mapping for **valuetype** data members follows the same rules as the C++ mapping for structured types as described in 6.14, 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). The actual data members of the OBV classes will be declared private.

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++
using Bytes = std::array<uint8_t, 64>;
class S {
public:
  ...
};
class Val : public virtual ValueBase {
public:

       …
       virtual IDL::traits <Val>::ref_type t() const = 0;
```

```
      virtual IDL::traits <Val>::ref_type& t() = 0;
      virtual void t(IDL::traits <Val>::ref_type) = 0;

      virtual const Bytes& w() const = 0;
      virtual Bytes& w() = 0;
      virtual void w(const Bytes&) = 0;
      virtual void w(Bytes&&) = 0;

      virtual const std::string& x() const = 0;
      virtual std::string& x() = 0;
      virtual void x(const std::string&) = 0;
      virtual void x(std::string&&) = 0;

protected:
      virtual int32_t v() const = 0;
      virtual int32_t& v() = 0;
      virtual void v(int32_t) = 0;

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



      virtual IDL::traits<A>::ref_type z() const = 0;
      virtual IDL::traits<A>::ref_type& z() = 0;

      virtual void z(IDL::traits<A>::ref_type) = 0;
      …
};
```

These rules for the accessors correspond directly to the parameter passing rules for structured types as explained in 6.14.

6.18.2 Constructors, Assignment Operators, and Destructors

A C++ **valuetype** class defines a protected default constructor, protected copy constructor, protected move 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 deleting value instances directly instead of using the reference type. The destructor is virtual to provide for proper destruction of derived value class instances.

For the same reasons, the generated OBV classes define a protected default constructor, protected copy constructor, protected move constructor, a protected explicit constructor that takes an initializer for each **valuetype** data member, and a protected destructor. The protected default constructor initializes object reference members to appropriately-typed nil object references, basic datatypes to their default value as listed in Table 6.2, and enums to their first value. All other members are initialized using their default constructors. The parameters of the explicit 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.

6.18.3 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 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.

The C++ signatures and memory management rules for **valuetype** operations are identical to those described in 6.7.8 for client-side **interface** operations.

As part of the valuetype traits **IDL::traits<>:: narrow** is provided. This method provides a portable way for applications to cast down the C++ inheritance hierarchy. If a nil reference is passed to one of these operations, it returns a nil reference. Otherwise, if the **valuetype** instance referenced to by the argument is an instance of the **valuetype** class being narrowed to, a reference to the narrowed-to class type is returned. If the **valuetype** instance pointed to by the argument is not an instance of the **valuetype** class being narrowed to, a nil reference is returned.

6.18.4 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 Example val5;
};
```

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

```
// C++
class Example : public virtual ValueBase {
public:
      virtual int16_t op1() = 0;
      virtual int32_t op2(IDL::traits<Example>::ref_type) = 0;
      virtual int32_t val2() const = 0;
      virtual int32_t& val2() = 0;
      virtual void val2(int32_t) = 0;
protected:
      Example();
      Example (const Example&);
      Example (Example&&);
      virtual ~Example();
      virtual int16_t val1() const = 0;
      virtual int16_t& val1() = 0;
      virtual void val1(int16_t) = 0;
      virtual const std::string& val3() const = 0;
      virtual std::string& val3() = 0;
      virtual void val3(const std::string&) = 0;
      virtual void val3(std::string&&) = 0;
      virtual IDL::traits<Example>::ref_type val5() const = 0;
      virtual IDL::traits<Example>::ref_type& val5() = 0;
      virtual void val5(IDL::traits<Example>::ref_type) = 0;
```

```
private:
      Example& operator=(const Example&) = delete;
      Example& operator=(Example&&) = delete;
};


class OBV_Example : public virtual Example {
public:
      virtual void val2 (int32_t) override;
      virtual int32_t val2 () const override;
      virtual int32_t& val2 () override;
protected:
      OBV_Example();
      OBV_Example (const OBV_Example&);
      OBV_Example (OBV_Example&&);
      OBV_Example (int16_t, int32_t, std::string, IDL::traits
      <Example>::ref_type;
      virtual ~OBV_Example();
      virtual int16_t val1() const override;
      virtual int16_t& val1() override;
      virtual void val1(int16_t) override;
      virtual const std::string& val3() const override;
      virtual std::string& val3() override;
      virtual void val3(const std::string&) override;
      virtual void val3(std::string&&) override;
      virtual IDL::traits<Example>::ref_type val5() const override;
      virtual IDL::traits<Example>::ref_type& val5() override;
      virtual void val5(IDL::traits<Example>::ref_type) override;
      // …
};
```

6.18.5 ValueBase default methods

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

```
// C++
class ValueBase {
public:
      virtual IDL::traits<ValueBase>::ref_type _copy_value();
protected:
      ValueBase();
      ValueBase (ValueBase&&);
      ValueBase(const ValueBase&);
      virtual ~ValueBase();
private:
      ValueBase operator=(ValueBase&&) = delete;
      ValueBase operator=(const ValueBase&) = delete;
};
```

The names of these operations begin with underscore to keep them from clashing with user-defined operations in derived **valuetype** classes. The copy_value operation returns by default a nil valuetype reference. The user can override this

method to allow the copy of a valuetype reference using its base reference.

ValueBase also provides a protected default constructor, a protected copy constructor, a protected move constructor, and a protected virtual destructor. The copy and move constructors are protected to disallow 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**.

6.18.6 Valuetype trait member types

For a valuetype the following additional member types shall be available as part of its type trait.

Table 6.12: Valuetype trait member types

Member	Definition
`is_abstract`	`std::false_type` or `std::true_type` type indicating whether the valuetype is defined as abstract
`is_truncatable`	`std::false_type` or `std::true_type` type indicating whether the valuetype is defined as truncatable

6.18.7 Value Boxes

A value box class essentially provides a shared 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 `ValueBase`. Unlike **valuetype**s no Valuetype factory has to be provided by the user.

6.18.7.1 Parameter Passing for Underlying Boxed Type

All value box classes provide `_value` member functions that allow the underlying boxed value to be passed to functions taking parameters of the underlying boxed type. For example, invoking `_value` 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++
IDL::traits<StringValue>::ref_type sval =
      CORBA::make_reference <StringValue>("string val");
X x (…)
x->op(sval->_value());         // boxed string is replaced
                               // by op() invocation
```

Assume the implementation of op is as follows:

```
// C++
void op(std::string& s)
{
      s = "new string val";
}
```

The return value of the **_value** 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.

6.18.7.2 Provided signature

Value boxclasses follow the rules of structured types as explained in 6.14. Additionally to these rules valueboxes ~~also~~are final and have:

- Accessors are always named **_value**
- A protected destructor and protected constructors
- ~~Protected~~Private, deleted copy and move assignment operators

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

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

```
// C++
class ColorValue final : public ValueBase {
public:
      Color _value() const;
      Color& _value();
      void _value(Color);

protected:
      ColorValue();
      explicit ColorValue(Color);
      ColorValue(ColorValue&&);
      ColorValue(const ColorValue&);
      ColorValue& operator=(const ColorValue&) = delete;
      ColorValue& operator=(ColorValue&&) = delete;
      virtual ~ColorValue();
};
```

For a valuebox the following additional member types shall be available as part of its type trait.

Table 6.13: Valuebox member types

Member	Definition
`boxed_traits`	**`IDL::traits<>`** for the boxed type of the valuebox

6.18.8 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 the **`OBV`** traits for them.

6.18.9 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 (except for abstract interfaces because here the valuetype class has to support implicit widening; see 6.19.2).
- 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 skeleton for this value type; this skeleton is available through the **`CORBA::servant_traits<>::base_type`** trait with 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.

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() = 0;

      virtual int16_t data() const = 0;
      virtual int16_t& data() = 0;
      virtual void data(int16_t) = 0;
      // …
};
```

```
class B_impl :
      public virtual CORBA::servant_traits<A>::base_type,
      public virtual IDL::traits<B>::base_type
{
public:
      virtual void op() override;
      // …
};
```

6.18.10 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 implementation cannot know *a priori* about all potential concrete value classes supplied by the application, and so the implementation 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.

6.18.10.1 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++ **`ValueFactoryBase`** class which has a protected constructor and destructor and deleted copy and move constructors and assignment operators.

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

private:
      virtual IDL::traits<ValueBase>::ref_type create_for_unmarshal() = 0;
      ValueFactoryBase(const ValueFactoryBase&) = delete;
      ValueFactoryBase(ValueFactoryBase&&) = delete;
      ValueFactoryBase& operator=(const ValueFactoryBase&) = delete;
      ValueFactoryBase& operator=(ValueFactoryBase&&) = delete;
};
```

The C++ mapping for the IDL **CORBA::ValueFactory** native type is an object reference to the **`ValueFactoryBase`** class, as shown above. Applications derive concrete factory classes 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 nil reference.

When unmarshaling value instances, the implementation 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 6.18.10) so as to preserve

any invariants they might have for their state. However, creation for unmarshaling is different because the implementation 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 implementation 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 implementation, 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 reference to it. Since the **`create_for_unmarshal`** function returns a reference to **`ValueBase`**, the caller may use the narrow function supplied by the value type IDL trait to narrow the reference back to a reference to a derived value type.

Once the implementation 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 implementation accesses the protected value data member modifiers of the value is implementation-specific and does not affect application portability.

The function allows the return type of the **`ORB::lookup_value_factory`** function to be narrowed to a reference to a type-specific factory (see 6.18.10).

6.18.10.2 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 factory class can be retrieved using the **`IDL::traits<A>::factory_type`** trait. 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 reference to the created **valuetype**.

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

```
// IDL
valuetype V {
        factory create_bool(in_boolean b);
        factory create_char(in_char c);
        factory create_octet(in octet o);
        factory create_other(in short s, in string p);
                ...
};
```

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

```
// C++
class V_factory : public ... {
public:
      virtual IDL::traits<V>::ref_type create_bool(bool val) = 0;
      virtual IDL::traits<V>::ref_type create_char(char val) = 0;
      virtual IDL::traits<V>::ref_type create_octet(uint8_t val)= 0;
      virtual IDL::traits<V>::ref_type create_other(
            uint16_t s, const std::string& p) = 0;
```

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

private:
      V_factory(const V_factory&) = delete;
      V_factory(V_factory&&) = delete;
      V_factory& operator=(const V_factory&) = delete;
      V_factory& operator=(V_factory&&) = delete;
};
```

Each generated factory class has a protected virtual destructor, a protected default constructor, deleted copy/move constructors, and deleted copy/move assignment operators. It also supplies a public pure virtual function corresponding to each initializer. Applications derive concrete factory classes from the **`IDL::traits<>::factory_type`** trait and register them with the implementation. 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 **valuetypes** 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 **`IDL::traits<>::obv_type`** trait class for the valuetype using the **`CORBA::make_reference<>`**.

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 **`IDL::traits<>::factory_type`** only need to override the **`create_for_unmarshal`** function.

6.18.10.3 Unmarshaling Issues

When the implementation 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 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 **`Any`**, and so value factories are not required to be registered with the implementation to allow such **valuetype**s to be unmarshaled. However, value factories must be registered with the implementation and available for lookup if the application attempts extraction of the **valuetypes** via the statically-typed Any extraction functions. See 6.17.3 for more details.

6.18.11 Custom Marshaling

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

6.19 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 6.7, only the ways in which the abstract interface mapping differs from the regular interface mapping are described here.

6.19.1 Abstract Interface Base

For abstract interfaces the **IDL::traits<>** trait must be provided. This trait delivers a strong reference type as **IDL::traits<>::ref_type** and a weak reference type as **IDL::traits<>::weak_ref_type**.

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 and move constructor
- a protected copy and move assignment operators
- a protected virtual destructor
- a **_to_object** and a **_to_value** operation

The **AbstractBase** class is shown below:

```
// C++

class AbstractBase {
public:
      virtual IDL::traits<Object>::ref_type _to_object();
      virtual IDL::traits<ValueBase>::ref_type _to_value();

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

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 reference to that valuetype, otherwise it returns a nil reference.

6.19.2 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 valuetype classes, abstract interface classes shall always be inherited as public virtual base classes.

- Normal **Any** insertion and extraction operators are generated for abstract interfaces.

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 reference for each abstract interface base class.

6.20 Mapping for Exception Types

- An OMG IDL exception is mapped to a C++ class that derives from the standard **UserException** class. All exception members must be initialized to their default value by the default constructor for the exception.
- The copy constructor, move constructor, assignment operator, move operator, and destructor automatically copy, move, or free the storage associated with the exception. For convenience, the mapping also defines an explicit constructor with one parameter for each exception member—this constructor initializes the exception members to the given values. The default constructor initialized all members to their default values as described in 6.14.

```
// C++
class Exception : public std::exception
{
public:
      virtual ~Exception();
      virtual void raise() const = 0;
      virtual const char* _name() const;
      virtual const char* _rep_id() const;
      virtual const char* what() const noexcept override;

protected:
      Exception();
      Exception(const Exception &);
      Exception(Exception &&);
      Exception &operator=(const Exception &);
      Exception &operator=(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;
}
```

The **_name()** function returns the unqualified (unscoped) name of the exception. The **_rep_id()** function returns the repository ID of the exception. Both return a pointer to a c-string with content related to the exception. This is guaranteed to be valid at least until the exception object from which it is obtained is destroyed or until a non-const member function of the exception object is called.

Each **Exception** class has to override **what()** which must return a null terminated character sequence containing a

generic description of the exception. Both the wording of such description and the character width are implementation-defined.

The **UserException** class is derived from a base **Exception** class.

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

```
// C++
enum class CompletionStatus : uint32_t {
      COMPLETED_YES,
      COMPLETED_NO,
      COMPLETED_MAYBE
};

class SystemException : public Exception
{
public:
      virtual ~SystemException();

      uint32_t minor() const;
      void minor(uint32_t);

      virtual void raise() const = 0;
      virtual const char* what() const noexcept override;

      CompletionStatus completed() const;
      void completed(CompletionStatus);
protected:
      SystemException();
      SystemException(const SystemException &);
      SystemException(SystemException &&);
      SystemException(uint32_t minor, CompletionStatus status);
      SystemException &operator=(const SystemException &);
      SystemException &operator=(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 final : public SystemException { ... };

class BAD_PARAM final : public SystemException { ... };
// etc.
```

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. Also the exceptions can be caught as **`std::exception`**.

6.20.1 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 final : public UserException
{
public:
      const Any& exception() const;
};
```

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.

6.20.2 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 moving 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 **Exception**. This is required to allow DSI-based applications to catch exceptions as **Exception&** and store them into a **ServerRequest**:

```
// C++
try
{

...

}

catch (const 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**. The following **Any** insertion for **Exception** shall be provided:

```
// C++
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++
bool operator>>=(const Any&, SystemException& se);
```

The operator returns true if the **Any** on which it is invoked contains a system exception and the implementation 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 implementation does not have static type information for the system exception, the operator returns true and **se** points to an instance of **UNKNOWN**. Otherwise, the operator returns false and the value of **se** is unchanged.

6.21 Mapping of Pseudo Objects to C++

IDL pseudo objects must be mapped to IDL local interfaces. These local interfaces must be implemented following the regular mapping for local objects.

6.22 TypeCode

A **TypeCode** represents OMG IDL type information. Typecodes are handled as reference type as described in 6.7.1. For typecodes the **IDL::traits** trait is provided. For the strong reference the mapping will provide the trait **IDL::traits<CORBA::TypeCode>::ref_type**.

No public 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** reference (**IDL::traits<CORBA::TypeCode>::ref_type**) 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**> refers 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.

6.22.1 TypeCode Interface

The **TypeCode** IDL interface is fully defined in version 3.2 of *Common Object Request Broker Architecture (CORBA), Interface Repository* clause. The *TypeCode Interface* sub clause is thus not duplicated here.

6.22.2 TypeCode C++ Class

```
// C++
class TypeCode
{
public:

      class Bounds final : public UserException { ... };
      class BadKind final : public UserException { ... };

      bool equal(IDL::traits<CORBA::TypeCode>::ref_type) const;
      bool equivalent(IDL::traits<CORBA::TypeCode>::ref_type) const;
      TCKind kind() const;

      IDL::traits<CORBA::TypeCode>::ref_type get_compact_typecode() const;

      const std::string& id() const;
      const std::string& name() const;

      uint32_t member_count() const;
      const std::string& member_name(uint32_t index) const;

      IDL::traits<CORBA::TypeCode>::ref_type member_type(uint32_t index) const;

      const Any& member_label(uint32_t index) const;
      IDL::traits<CORBA::TypeCode>::ref_type discriminator_type() const;
      int32_t default_index() const;

      uint32_t length() const;

      IDL::traits<CORBA::TypeCode>::ref_type content_type() const;

      uint16_t fixed_digits() const;
      int16_t fixed_scale() const;

      Visibility member_visibility(uint32_t index) const;
```

```
        ValueModifier type_modifier() const;
        IDL::traits<CORBA::TypeCode>::ref_type concrete_base_type() const;
};
```

6.23 ORB

An **ORB** is the programmer interface to the Object Request Broker. This pseudo interface has to be implemented as a regular local interface.

6.23.1 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 3.2 of *Common Object Request Broker Architecture (CORBA), 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 {
            using ORBid = std::string;
            static IDL::traits<ORB>::ref_type ORB_init(
                        int& argc,
                        char** argv,
                        const std::string& orb_identifier = std::string ()
            );
};
```

The C++ mapping for **`ORB_init`** deviates from the regular C++11 mapping in its handling of the **`arg_list`** parameter. This is intended to provide a meaningful C++11 definition of the initialization interface, which has a natural C++ binding matching the main of an application. The **`arg_list`** sequence 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 **`argv`** 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**&**.

6.24 Object

The rules in this sub clause apply to OMG IDL interface **Object**, the base of the OMG IDL interface hierarchy. Interface **Object** defines a normal CORBA object and is mapped as defined in this specification. In addition to these rules, all operation names in interface **Object** have leading underscores in the corresponding C++ class.

6.25 Local Object

The C++ mapping of **LocalObject** is a class derived from **Object** that is used as a base class for locality constrained object implementations. The class mapping the interface should be (indirectly) derived from **LocalObject** and must be available as **IDL::traits<>::base_type**. An object reference referring to a local object must be created using the **CORBA::make_reference<>** factory method as described in 6.7.1. Within the context of a request on a local object the **_this()** method allows the retrieval of the object reference for the target object handling that request. Here is an example of how to implement a local interface:

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

```
// C++
class LocalIF {
protected:
       virtual ~LocalIF ();
       IDL::traits<LocalIF>::ref_type_ this ();
};
class MyLocalIF : public IDL::traits<LocalIF>::base_type {
public:
       MyLocalIF ();
       virtual ~MyLocalIF();

       void an_op(int32_t an_arg) override {
             IDL::traits<LocalIB>::ref_type_b_ref = ...; // obtain reference to B
             // invoke if_op with a reference to this local object
             b_ref->if_op (this->_this ());
};
```

```
IDL::traits<LocalIF>::ref_type myref =
      CORBA::make_reference <MyLocalIF> ();
```

6.26 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.

6.26.1 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 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 an *inheritance-based* mapping for the application-supplied implementation class and the generated class or classes for the interface.

6.26.2 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 Servant
      {
      public:
            virtual IDL::traits<PortableServer::POA>::ref_type _default_POA();
            virtual IDL::traits<CORBA::InterfaceDef>::ref_type _get_interface();
            virtual bool _is_a(const std::string& logical_type_id);
            virtual bool _non_existent();
      protected:
            virtual ~Servant();
            Servant();
            Servant(const Servant &);
            Servant(Servant &&);
            Servant& operator=(const Servant &);
            Servant& operator=(Servant &&);
      };
};
```

The **Servant** destructor is protected and virtual to ensure that skeleton classes derived from it can be properly destroyed but never be deleted directly. The default constructor, along with other implementation-specific constructors, must be

protected so that instances of **Servant** 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 a copy constructor 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 **Servant** 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 **Servant** 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 **Servant** can override this definition to return the POA of their choice, if desired.

Servant 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 operations 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.

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

6.26.3 Servant references

Given an interface **Foo** the mapping will provide a **CORBA::servant_traits<Foo>** trait. The strong reference type is provided as **CORBA::servant_traits<Foo>::ref_type** trait (aka **CORBA::servant_reference<>**) that can be used to store or pass a reference to the servant of type **Foo**. Also a weak reference **CORBA::servant_traits<Foo>::weak_ref_type** trait (aka **CORBA::weak_servant_reference<>**) has to be provided. These servant reference types behave as reference types as described in “Reference Types” on page 6.

This trait together with the **CORBA::make_reference<>** factory method must be used to write exception-safe and type-safe code for heap-allocated servants (a C++11 program is not allowed to use **new/delete** to allocate servants). For example if we have an interface **Test::Hello** that is implemented by **Foo_impl**:

```
IDL::traits<Test::Hello>::ref_type
Foo::some_function()
{
      CORBA::servant_traits<Test::Hello>::ref_type foo_servant =
            CORBA::make_reference <Foo_impl> ();
      foo_servant->do_something();        // might throw…
      some_poa->activate_object_with_id(...);
      return foo_servant->_this ();
}
```

6.26.4 Servant argument passing

The POA will maintain servants as servant references with the semantics as described in 6.7.1. For each POA the **ServantActivator** and **ServantLocator** provide operations that either pass a **Servant** as a parameter or returns a **Servant** a **CORBA::servant_traits<PortableServer::Servant>::ref_type** will be passed.

6.26.5 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 acquired 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 acquired 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 A_skel : public virtual ...{
public:
       IDL::traits<A>::ref_type _this();
       …
};
```

Assuming **`A_impl`** is a class derived from **`CORBA::servant_traits<A>::base_type`** 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++
CORBA::servant_traits<A>::ref_type my_a =
       CORBA::make_reference<A_impl> ();
IDL::traits<A>::ref_type a = my_a->_this();
```

6.26.6 Inheritance-Based Interface Implementation

Implementation must 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 …
{
public:
       virtual int16_t op1();
       virtual void op2(const int32_t& val);
       …
};
```

On the server side, a skeleton class is generated. This class is opaque to the programmer, though it will contain a member function corresponding to each operation in the interface. The type of the skeleton class is defined by the **CORBA::servant_traits<T>::base_type** trait related to the corresponding interface **T**. The type the traits refers to has to be either directly or indirectly derived from the servant base class **PortableServer::Servant.** The **PortableServer::Servant** class must be a virtual base class of the type related to the trait to allow portable implementations to multiply inherit from both skeleton classes and implementation classes for other base interfaces without error or ambiguity.

The **PortableServer::Servant** must have a protected destructor preventing the user to directly delete a servant instead of using the reference semantics.

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

```
// C++
class A_skel : public virtual …
{
public:
       // ...server-side implementation-specific detail

       // goes here…
       virtual int16_t op1() = 0;
       virtual void op2(const int32_t& val) = 0;
       …
protected:
       A_skel ();
       virtual ~A_skel ();
};
```

If interface **A** were defined within a module rather than at global scope, e.g., **Mod::A**, the trait for this skeleton class would

be **CORBA::servant_traits<Mod::A>::base_type**.

To implement this interface using inheritance, a programmer must derive from this trait 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 virtual CORBA::servant_traits<A>::base_type
{
public:
      virtual int16_t op1() override;
      virtual void op2(const int32_t val) override;
      …
protected:
      virtual ~A_impl ();
};
```

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 be registered as DSI-based servants, as described in 6.27.

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 A_skel : public virtual ... { ... };
class B_skel : public virtual A_skel { ... };
class D_skel : public virtual B_skel,
                  public virtual C_skel { ... };
```

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 CORBA::servant_traits<A>::base_type { ... };
class B_impl: public virtual CORBA::servant_traits<B>::base_type,
                           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 B_skel : public virtual ...,
      protected virtual A { ... };
```

The abstract interface is inherited with protected access to prevent accidental conversion of the skeleton reference to an abstract interface reference. This also allows implementation classes and valuetypes to share an implementation of the abstract interface:

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

```
// C++
class MyA : public virtual CORBA::servant_traits<A>::base_type { ... };
class MyB : public virtual CORBA::servant_traits<B>::base_type,
                        protected virtual MyA
{ ... };
class MyV : public virtual V, public virtual MyA { ... };
```

6.26.7 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 CORBA::servant_traits<A>::base_type
{
public:
      virtual void f() override;
      …
};
```

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 CORBA::servant_traits<A>::base_type
{
public:
        virtual void f() override;
        virtual void g() override;
private:
        int32_t 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.

When the application code needs a **`CORBA::servant_reference<>`** within a member function it can retrieve a servant reference to this using **`this->_lock ()`** which returns a reference to this. This reference then can be passed to other operations that require a **`CORBA::servant_reference<>`**.

6.26.8 Delegation-Based Interface Implementation

Inheritance is not always the best solution for implementing servants. Using inheritance from the IDL-generated traits 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. For example, implementing objects using existing legacy code might be impossible if inheritance from some predefined 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 trait, 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. The examples in this sub clause use the following IDL.

```
// IDL
interface A
{
        short op1();
```

```
        void op2(in long val);
};
```

In addition to generating a skeleton traits, the IDL compiler generates a delegating template called a *tie*. This template is opaque to the application programmer, though like the skeleton, it provides a method corresponding to each IDL operation. The type of the tie template is defined by the **`CORBA::servant_traits<T>::tie_type`** trait related to the corresponding interface T.

```
// C++
template <class T>
class TIE : public …
{
public:
        …
};
```

An instantiation of this template performs the task of delegation. When the template is instantiated with a class T that provides the operations of interface A, then the **`TIE`** template will delegate all operations to an instance of that implementation class. A shared pointer to the actual implementation object is passed to the tie constructor when an instance of the **`TIE`** template is created. When a request is invoked on it, the tie servant will delegate the request by calling the corresponding method in the implementation object.

```
// C++
template <class T>
class TIE : public …

{
private:
        std::shared_ptr<T> tied_object_{};
public:
        explicit TIE(std::shared_ptr<T> t, IDL::traits<PortableServer::POA>::ref_type
poa = {})
                : tied_object_(std::move(t)), poa_(std::move(poa)){}

        virtual ~TIE() = default;

        // tie-specific functions
        std::shared_ptr<T> _tied_object()
        {
                return tied_object_;
        }

        void _tied_object(std::shared_ptr<T> t)
        {
                tied_object_ = t;
        }

        // IDL operations
        int16_t op1()
        {
                return tied_object_->op1();
        }

        void op2(int32_t val)
        {
```

```
            tied_object_->op2(val);
      }

      // override ServantBase operations
      IDL::traits<PortableServer::POA>::ref_type _default_POA() override
      {
            if (poa_) {
                  return poa_;
            } else {
                  // return root POA
            }
      }

private:
      IDL::traits<PortableServer::POA>::ref_type poa_;

      // copy and assignment not allowed
      TIE() = delete;
      TIE(const TIE&) = delete;
      TIE(TIE&&) = delete;
      TIE& operator=(const TIE&) = delete;
      TIE& operator=(TIE&&) = delete;
};
```

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.

The constructors cause the tie servant to delegate all calls to the C++ object bound to shared pointer T. The **`_tied_object()`** accessor function allows callers to access the C++ object being delegated 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 to 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 reference 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.

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 **`TIE<T>::op2()`** operation is normally defined as follows:

```
// C++
template <class T>
void TIE<T>::op2(int32_t val)
{
```

```
        tied_object_->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 **op2()** function can be made to call it if the following specialization is provided:

```
// C++
template <>
void CORBA::servant_traits<A>::tie_type<Foo>::op2(int32_t val)
{
        _tied_object()->log_value(val);
}
```

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

6.27 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++

6.27.1 Mapping of ServerRequest to C++

The **ServerRequest** pseudo object maps to a C++ class that follows the local interface mapping.

6.27.2 Mapping of PortableServer Dynamic Implementation Routine

In C++, DSI servants inherit from the standard **DynamicImplementation** class. This class inherits from the **Servant** 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 Servant
        {
        public:
                IDL::traits<Object>::ref_type _this();
                virtual void invoke(IDL::traits<ServerRequest>::ref_type request)=0;
                virtual RepositoryId _primary_interface(const ObjectId& oid,
                        IDL::traits<POA>::ref_type poa) = 0;
        };
};
```

The **_this()** function returns an **IDL::traits<Object>::ref_type** 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 **Servant** instead.

The **_primary_interface()** method receives an **ObjectId** value and a **POA** 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.

6.28 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++11 mapping provides several conversion functions that convert strings to **ObjectId** and vice-versa:

```
// C++
namespace PortableServer
{
      std::string ObjectId_to_string(const ObjectId&);
      std::wstring ObjectId_to_wstring(const ObjectId&);

      ObjectId string_to_ObjectId(const std::string&);
      ObjectId wstring_to_ObjectId(const std::wstring&);
};
```

If conversion of an **ObjectId** to a string would result in illegal characters in the string, the first two functions throw the BAD_PARAM exception.

6.29 Mapping for PortableServer::ServantManager

6.29.1 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 {
                  …
                  using Cookie = void*;
```

```
        };
};
```

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`.

6.29.2 ServantManagers and AdapterActivators

Portable servants that implement the `PortableServer::AdapterActivator`, the `PortableServer::ServantActivator`, or `PortableServer::ServantLocator` interfaces are implemented just like any other servant using the inheritance-based approach.

6.29.3 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.

6.30 C++11 Protected names

Table 6.14 lists all C++11 protected names including the keywords from the C++11 specification (ISO/IEC 14882:2011) dated September 2011.

Table 6.14: C++ Protected Names

and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	class
compl	const	const_cast	continue	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	FALSE	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static	static_cast
struct	switch	template	this	throw	TRUE
thread_local	int16_t	int32_t	int64_t	uint16_t	uint32_t
try	typedef	typeid	typename	union	unsigned
uint64_t	uint8_t	noexcept	char16_t	char32_t	what
using	virtual	void	volatile	wchar_t	while
xor	xor_eq	alignas	alignof	constexpr	decltype

6.31 Mapping for IDL 4 Extended Data-Types

6.31.1 Maps

An IDL unbounded map type maps to a C++ **`std::map`** or to a type that delivers **`std::map`**'s semantics and supports transparent conversion to and from **`std::map`**. The **`std::map<K, T, Compare, Allocator>`** template shall be instantiated with the **`K`** class parameter being the C++11 type corresponding to the key type and the **`T`** parameter is the C++11 type corresponding to the element type.

The arguments for the **`Compare`** and **`Allocator`** parameters are unspecified and may or may not take their default values.

A bounded map is mapped to a distinct type to differentiate from an unbounded map. This distinct type must deliver **`std::map`** semantics and support transparent conversion from bounded to unbounded and vice versa including support for move semantics. As a result, the programmer is responsible for enforcing the bound of bounded maps at run time. Implementations of the mapping are under no obligation to prevent assignment of a map to a bounded map type if the map size exceeds the bound.

Implementations must at run time detect attempts to pass a map that exceeds the bound as a parameter across an interface. When an implementation detects this error, it must raise a **BAD_PARAM** system exception to signal the error.

Additionally the C++ **`std::map`** can have a size that is larger than the maximum size of an IDL map that is limited in length to the maximum of ULong. When this happens the implementation must raise a **BAD_PARAM** system exception to signal the error.

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

// IDL

typedef map<unsigned long, T> M1; // unbounded map

typedef map<string, T, 20> M2; // bounded map

```
// C++
using M1 = std::map<uint32_t, T>;
using M2 = IDL::bounded_map<std::string, T, 20;
```

For an unbounded map the following additional member types shall be available as part of its type trait.

Table 6.15: Unbounded Map Traits Member Types

Member	Definition
`key_traits`	**`IDL::traits<>`** for the key type
`value_traits`	**`IDL::traits<>`** for the value type
`is_bounded`	**`std::false_type`** type indicating that this type is not bounded

For a bounded map the following additional member types shall be available as part of its type trait.

Table 6.16: Bounded Map Traits Member Types

Member	Definition
key_traits	**IDL::traits<>** for the key type
value_traits	**IDL::traits<>** for the value type
is_bounded	**std::true_type** type indicating that this type is bounded
bound	**std::integral_constant** type of value type **uint32_t** indicating the bound of the map

6.31.2 Integers restricted to holding 8-bits of information

IDL int8 and uint8 data types are mapped to C++'s fixed-width integer types int8_t and uint8_t, respectively. The default value for these types is 0.

6.31.3 Structures with Single Inheritance

In addition to the requirements in 6.14.1, mapped classes for IDL structures that make use of inheritance have the following features:

- Public inheritance from the mapped C++ class corresponding to the IDL struct's base type
- Any member function defined by 6.14 or 6.14.1 that acts on a per-data-member basis includes the base subobject as the implicit first data member. This also applies to the swap() function.
- The constructor which "accepts values for each struct member in the order they are specified in IDL" also accepts, as its first parameter, an object of the mapped base type (passed by const reference).

6.31.4 Bit Masks

IDL bitmask types each map to two C++ type names:

- An unscoped enum named <Bitmask>Bits with an explicitly defined underlying type. That underlying type is the smallest mapped unsigned integer type (see Table 6.2: Basic Data Type Mappings in combination with section 6.31.2) which has sufficient bits for the bit_bound of the bitmask. The <Bitmask>Bits enumerators are the values defined in the scope of the IDL bitmask, with each enumerator explicitly initialized to its corresponding integer value. The exact form of this initialization expression (for example, use of non-decimal bases or bit shifts or other operators) is implementation-defined.
- A type alias ("using") defining the <Bitmask> name itself as an alias for the underlying type of the enum described above.

An example follows:

```
// IDL
@bit_bound(8)
bitmask MyBitMask {
  @position(0) flag0,
```

```
  @position(1) flag1,
  @position(4) flag4,
  @position(6) flag6
};

// C++
enum MyBitMaskBits : std::uint8_t {
  flag0 = 1,
  flag1 = 2,
  flag4 = 16,
  flag6 = 64
};
using MyBitMask = std::uint8_t;
```

6.31.5 Bit Sets

IDL bitset types are mapped to to C++ structs that meet the C++11 requirements for aggregate initialization. The only members of these structs are bit fields, and they don't use inheritance. In cases where the IDL bitset does inherit from a base type, the mapped bitfields of the base type (recursively) appear directly in the mapped derived type.

The mapped type's bit field members directly correspond to the IDL bitfields, including the use of anonymous (nameless) bit fields. The C++ data type for each bit field is the mapped type (see Table 6.2: Basic Data Type Mappings in combination with section 6.31.2) of the IDL bitfield destination type.

An example follows:

```
// IDL
bitset BitSet1 {
  bitfield<1> bit0;
  bitfield<1>;
  bitfield<2, unsigned short> bits2_3;
};

bitset BitSet2 : BitSet1 {
  bitfield<3>;
  bitfield<1> bit7;
};

// C++
struct BitSet1 {
  bool bit0 : 1;
```

```
  bool : 1;
  uint16_t bits2_3 : 2;
};

struct BitSet2 {
  bool bit0 : 1;
  bool : 1;
  uint16_t bits2_3 : 2;
  uint8_t : 3;
  bool bit_7 : 1;
};
```

This page intentionally left blank.