Date: March 2025

IDL4 to C++ Language Mapping

Version 1.0

OMG Document Number	formal/25-03-03 [smsc/25-03-03]
Normative Reference:	https://www.omg.org/spec/IDL4-CPP

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, 9C Medway Road, PMB 274, Milford, MA 01757, 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: https://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 https://www.omg.org, under Specifications, Report a Bug/Issue.

Table of Contents

Table of Tables

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 https://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
9C Medway Road
PMB 274
Milford, MA 01757

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 by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Specifications, Report a Bug/Issue.

1 Scope

This specification defines the mapping of OMG Interface Definition Language v4 to the C++ programming language. The language mapping covers all of the IDL constructs in the current Interface Definition Language specification [OMG-IDL4]. The language mapping makes use of C++ language features as appropriate and natural.

The specification also provides mapping rules for the building blocks introduced in IDL4 that are not addressed in the classic C++ and C++11 Language Mappings (see [OMG-C++] and [OMG-C++11]). These set of rules allow implementers of the classic mappings to extend existing IDL compilers and platforms to incorporate concepts from IDL4, such as extended data types and annotations, using a standard set of mapping rules that are consistent with the requirements and conventions of the original specifications.

2 Conformance Criteria

Conformance to this specification can be considered from two perspectives:

1. implementations (for example, a tool [compiler] that applies the mapping to generate C++ source code from IDL); and
2. users (for example, application source code that interacts with the C++ source code generated by a compiler).

Table 2.1: Conformance Points

Implementation	A conformant implementation shall transform IDL input into C++ source code output as specified in Chapter 7.
User	Application source code that conforms to this specification makes use of the C++ data types and APIs as defined in Chapter 7. Conformant application source code shall make no assumptions about the underlying implementation or utilize any unspecified API or behavior beyond what is specified in the language mapping. Conformant application source code, as a result, will be portable across 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.

[ISO/IEC-14882:1998] ISO/IEC, ISO/IEC 14882:1998 Programming Languages – C++, https://www.iso.org/standard/25845.html

[ISO/IEC-14882:2003] ISO/IEC, ISO/IEC 14882:2003 Programming Languages – C++, https://www.iso.org/standard/38110.html

[ISO/IEC-14882:2011] ISO/IEC, ISO/IEC 14882:2011 Programming Languages – C++, https://www.iso.org/standard/64029.html

[ISO/IEC-14882:2017] ISO/IEC, ISO/IEC 14882:2017 Programming Languages – C++, https://www.iso.org/standard/68564.html

[OMG-C++] OMG, C++ Language Mapping, Version 1.3, https://www.omg.org/spec/CPP/1.3

[OMG-C++11] OMG, C++11 Language Mapping, Version 1.6, https://www.omg.org/spec/CPP11/1.6

[OMG-CORBA-COMP] OMG, Common Object Request Broker Architecture, Part 3: CORBA Components, Version 3.4, https://www.omg.org/spec/CORBA/3.4

[OMG-CORBA-IFC] OMG, Common Object Request Broker Architecture, Part 1: CORBA Interfaces, Version 3.4, https://www.omg.org/spec/CORBA/3.4

[OMG-IDL4] OMG, Interface Definition Language, Version 4.2, https://www.omg.org/spec/IDL/4.2

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Building Block

A Building Block is a consistent set of IDL rules that together form a piece of IDL functionality. Building blocks are atomic, meaning that if selected, they shall be totally supported.

Building blocks are described in [OMG-IDL4] Chapter 7, IDL Syntax and Semantics.

C++

C++ is a general-purpose computer programming language.

Language Mapping

An association of elements in one language to elements in another language (from IDL to C++, in this case) that facilitates a transformation from one language to another.

5 Symbols

The acronyms used in this specification are shown in Table 5.1.

Table 5.1: Acronyms

Acronym	Meaning
CCM	Corba Component Model
CORBA	Common Object Request Broker Architecture

Acronym	Meaning
DDS	Data Distribution Service
IDL	Interface Definition Language

6 Additional Information

6.1 Changes to Adopted OMG Specifications

This specification does not change any adopted OMG specification.

6.2 Acknowledgments

The following companies submitted this specification:

- Real-Time Innovations, Inc.
- ZettaScale Technology
- Objective Interface Systems, Inc.
- Micro Focus International Plc.

6.3 Intellectual Property Rights

This page intentionally left blank.

7 IDL to C++ Language Mapping

7.1 General

7.1.1 Names

IDL member names and type identifiers shall map to equivalent C++ names and type identifiers with no change.

7.1.2 Reserved Names

This specification reserves the use the following names for its own purposes:

- The keywords in existing and future versions of the C++ language, which includes but is not limited to the following names[1]:

`alignas`	`delete`	`reinterpret_cast`
`alignof`	`do`	`requires`
`and`	`double`	`return`
`and_eq`	`dynamic_cast`	`short`
`asm`	`else`	`signed`
`atomic_cancel`	`enum`	`sizeof`
`atomic_commit`	`explicit`	`static`
`atomic_noexcept`	`export`	`static_assert`
`auto`	`extern`	`static_cast`
`bitand`	`false`	`struct`
`bitor`	`float`	`switch`
`bool`	`for`	`synchronized`
`break`	`friend`	`template`
`case`	`goto`	`this`
`catch`	`if`	`thread_local`
`char`	`inline`	`throw`
`char8_t`	`int`	`true`
`char16_t`	`long`	`try`
`char32_t`	`mutable`	`typedef`
`class`	`namespace`	`typeid`
`compl`	`new`	`typename`
`concept`	`noexcept`	`union`
`const`	`not`	`unsigned`
`consteval`	`not_eq`	`using`
`constexpr`	`nullptr`	`virtual`
`constinit`	`operator`	`void`

[1] An updated list of reserved keywords in the C++ programming language, along with the C++ version that introduced them, is available at https://en.cppreference.com/w/cpp/keyword.

`const_cast`	`or`	`volatile`
`continue`	`or_eq`	`wchar_t`
`co_await`	`private`	`while`
`co_return`	`protected`	`xorg`
`co_yield`	`public`	`xor_eq`
`decltype`	`reflexpr`	
`default`	`register`	

The use of any of these names for a user-defined IDL type or interface (assuming it is also a legal IDL name) shall result in the mapped name having an underscore ("`_`") prepended.

7.1.3 C++ Language Version Requirements

The language mappings defined in this specification require C++11 [ISO/IEC-14882:2011] as the minimum C++ standard version.

Implementers of this specification who may need to comply with C++98 [ISO/IEC-14882:1998] or C++03 [ISO/IEC-14882:2003] shall follow the Compatibility Rules for C++98 and C++03 defined in Annex C. Such rules provide alternative mappings for IDL constructs that map to C++ code incompatible with those versions of the C++ standard.

7.1.4 IDL Type Traits

The mapping rules for IDL types specified in this chapter shall provide the following type traits to facilitate template metaprogramming. Type traits shall be available under the **`omg::types`** namespace.

Table 7.1: Common Type Traits for Mapped IDL Types

Member	Definition
`template <typename T>` `struct value_type;`	Defines **`type`** - Type to be used as **`return`** C++ type.
`template <typename T>` `struct in_type;`	Defines **`type`** - Type to be used as **`in`** C++ type.
`template <typename T>` `struct out_type;`	Defines **`type`** - Type to be used as **`out`** C++ type.
`template <typename T>` `struct inout_type`	Defines **`type`** - Type to be used as **`inout`** C++ type.

The trait is undefined when the type **`T`** is incorrect (for example **`bound<int>::value`** is not defined).

Each trait **`struct`** with a type or a value member shall have an alias ending in **`_t`** or **`_v`**, respectively. For example:

```
template <typename T>
using in_type_t = typename in_type<T>::type;
```

7.2 Core Data Types

7.2.1 IDL Specification

There is no direct mapping of the IDL Specification itself. The elements contained in the IDL specification are mapped as described in the following sections.

7.2.2 Modules

IDL modules shall be mapped to C++ namespaces of the same name. All IDL type declarations within the IDL module shall be mapped to corresponding C++ declarations within the generated namespace. IDL declarations not enclosed in any module shall be mapped into the global scope.

For example, the following module declaration in IDL:

```
// ...
module my_math {
    // ...
};
```

Would map to the following C++ namespace declaration:

```
// ...
namespace my_math {
    // ...
}
```

7.2.3 Constants

IDL constants of numeric and boolean types shall be mapped to C++ **`constexpr`** declarations of equivalent type with the same name and value within the equivalent scope and namespace where they are defined.

IDL constants of string type shall be mapped to a **`constexpr`** declaration of **`omg::types::string_view`** type that provides the interface and semantics of **`std::string_view`**, as defined in C++17 [ISO/IEC-14882:2017]. Implementations supporting C++17 and above may map IDL constants of **`string`** type to **`constexpr std::string_view`** directly.

IDL constants of **`wstring`** type shall be mapped to a **`constexpr`** of **`omg::types::wstring_view`** type that provides the interface and semantics of **`std::wstring_view`**, as defined in C++17 [ISO/IEC-14882:2017]. Implementations supporting C++17 and above may map IDL constants of **`wstring`** type to **`constexpr std::wstring_view`** directly.

For example, the IDL const declarations below:

```
module my_math {
    const string my_string = "My String Value";
    const double PI = 3.141592;
};
```

would map to the following C++:

```
namespace my_math {
    constexpr omg::types::string_view my_string = "My String Value";
    constexpr double PI = 3.141592;
}
```

The constant value of wide character and wide string constants shall be preceded by **`L`** in C++.

For example, IDL constant:

```
const wstring ws = "Hello World";
```

would map to the following C++:

```
constexpr omg::types::wstring_view ws{ L"Hello World" };
```

7.2.4 Data Types

7.2.4.1 Basic Types

7.2.4.1.1 Integer Types

Integer types shall be mapped as shown in Table 7.2. The default value of a mapped integer type is 0.

Table 7.2: Mapping of Integer Types

IDL Type	C++ Type
`short`	`int16_t`
`unsigned short`	`uint16_t`
`long`	`int32_t`
`unsigned long`	`uint32_t`
`long long`	`int64_t`
`unsigned long long`	`uint64_t`

7.2.4.1.2 Floating-Point Types

IDL floating-point types shall be mapped as shown Table 7.3. The default value of a mapped floating point type is 0.

Table 7.3: Floating-Point Types Mapping

IDL Type	C++ Type
`float`	`float`
`double`	`double`
`long double`	`long double`

7.2.4.1.3 Char Types

The IDL **`char`** type shall be mapped to the C++ type **`char`**. The default value of a mapped **`char`** is 0.

7.2.4.1.4 Wide Char Types

The IDL **`wchar`** type shall be mapped to the C++ type **`wchar_t`**. The default value of a mapped **`wchar_t`** is 0.

7.2.4.1.5 Boolean Types

The IDL **boolean** type shall be mapped to the C++ **bool**, and the IDL constants **TRUE** and **FALSE** shall be mapped to the corresponding C++ boolean literals **true** and **false**. The default value of a mapped **bool** is **false**.

7.2.4.1.6 Octet Type

The IDL type **octet**, which defines an 8-bit quantity, shall be mapped to the C++ **uint8_t** type. The default value of a mapped **uint8_t** is 0.

7.2.4.2 Template Types

7.2.4.2.1 Sequences

IDL **sequence**s shall be mapped to a C++ **std::vector<T>**, or to a type named **omg::types::sequence<T>** that delivers **std::vector<T>** semantics. The template parameter **T** is the C++ mapped type. For interoperability purposes, implementers shall always define **omg::types::sequence<T>**, which may be declared as an alias to **std::vector<T>**. Other implementations of **omg::types::sequence<T>** shall support conversion to and from **std::vector<T>**.

Bounded sequences shall be mapped to a C++ **std::vector<T>**, or to a type named **omg::types::bounded_sequence<T, N>** (where **T** is the mapped type and **N** the bound size) that delivers **std::vector<T>** semantics. For interoperability purposes (e.g., for potential use of trait specializations), implementers shall always define **omg::types::bounded_sequence<T,N>**, which may be declared as an alias to **std::vector<T>**. Other implementations of **omg::types::bounded_sequence<T,N>** shall support conversion to and from **std::vector<T>**.

NOTE—Implementers of this specification shall provide a bound checking mechanism for bounded **sequence**s (e.g., at serialization time). Please refer to Annex A for platform-specific bound checking mechanisms.

Sequences shall provide the following specializations for the type traits defined in Clause 7.1.4:

Table 7.4: Type Trait Specializations for Sequences

Member	Definition
`template<>` `struct is_bounded;`	Inherits **std::true_type** if the sequence is bounded and **std::false_type** if the sequence is unbounded. When mapping to **std::vector<T>**, **is_bounded** shall inherit **std::false_type**.
`template <>` `struct bound;`	Inherits **std::integral_constant<size_t, b>** where **b** is the value of the bound. For an unbounded sequence type, the value of **b** shall be **std::numeric_limits<std::size_t>::max()**.

The example below shows valid mappings for IDL bounded and unbounded sequences:

```
typedef sequence<T> V1;          // unbounded sequence
typedef sequence<T, 3> V2;       // bounded sequence
typedef sequence<V1> V3;         // unbounded sequence of sequence
```

mapping sequences using C++ **std::vector**:

```
namespace omg { namespace types {
```

```
template<typename T> using sequence = typename std::vector<T>;
template<typename T, size_t N> using bounded_sequence = typename std::vector<T>;

}} // namespace omg::types

using V1 = std::vector<T>;
using V2 = std::vector<T>;
using V3 = std::vector<V1>;
```

mapping sequences to **omg::types::sequence<T>** and **omg::types::bounded_sequence<T,N>**:

```
namespace omg { namespace types {

template<typename T> using sequence = typename std::vector<T>;
template<typename T, size_t N> using bounded_sequence = typename std::vector<T>;

}} // namespace omg::types

using V1 = omg::types::sequence<T>;
using V2 = omg::types::bounded_sequence<T, 3>;
using V3 = omg::types::sequence<V1>;
```

7.2.4.2.2 Strings

IDL **string**s shall be mapped to C++ **std::string**, or to a type named **omg::types::string** that delivers **std::string** semantics. For interoperability purposes, implementers shall always define **omg::types::string**, which may be declared as an alias to **std::string**. Other implementations of **omg::types::string** shall support conversion to and from **std::string**.

Bounded strings may be mapped to a type called **omg::types::bounded_string<N>** (where **N** is the bound size), which delivers the semantics of an **std::string**, and supports conversion to and from it. For interoperability purposes (e.g., for potential use of trait specializations), implementers shall always define **omg::types::bounded_string<N>**, which may be declared as an alias to **std::string**. Other implementations of **omg::types::bounded_string<N>** shall support conversion to and from **std::string**.

NOTE—Implementers of this specification shall provide a bound checking mechanism for bounded **string**s (e.g., at serialization time). Please refer to Annex A for platform-specific bound checking mechanisms.

Strings shall provide the following specializations for the type traits defined in Clause 7.1.4:

Table 7.5: Type Trait Specializations for Strings

Member	Definition
`template<>` `struct is_bounded;`	Inherits **std::true_type** if the **string** is bounded and **std::false_type** if the **string** is unbounded. When mapping to **std::string**, **is_bounded** shall inherit **std::false_type**.
`template <>` `struct bound;`	Inherits **std::integral_constant<size_t, b>** where **b** is the value of the bound. For an unbounded **string** type, the value of **b** shall be **std::numeric_limits<std::size_t>::max()**.

7.2.4.2.3 Wstrings

IDL **wstrings** shall be mapped to C++ **std::wstring**s, or to a type named **omg::types::wstring** that delivers **std::wstring** semantics and supports conversion from and to **std::wstring**[2].

Bounded **wstring**s may be mapped to a type called **omg::types::bounded_wstring<N>** (where **N** is the bound size), which delivers the semantics of an **std::wstring**, and supports conversion to and from it.

NOTE—Implementers of this specification shall provide a bound checking mechanism for bounded **wstring**s (e.g., at serialization time). Please refer to Annex A for platform-specific bound checking mechanisms.

Wstrings shall provide the following specializations for the type traits defined in Clause 7.1.4:

Table 7.6: Type Trait Specializations for Wstrings

Member	Definition
template<> **struct is_bounded;**	Inherits **std::true_type** if the **wstring** is bounded and **std::false_type** if the **wstring** is unbounded. When mapping to **std::wstring**, **is_bounded** shall inherit **std::false_type**.
template <> **struct bound;**	Inherits **std::integral_constant<size_t, b>** where **b** is the value of the bound. For an unbounded **wstring** type, the value of **b** shall be **std::numeric_limits<std::size_t>::max()**.

7.2.4.2.4 Fixed Type

The IDL **fixed** type shall map to a C++ **class** named **fixed**. The **fixed** class shall comply with the following interface and shall include the following methods under the **omg::types** namespace:

```
namespace omg { namespace types {

class fixed {
public:
    // Constructors
    explicit fixed(string_view);
    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&);
```

[2] Supporting the semantics of a **std::wstring** does not limit the mapping of **omg::types::wstring** to use **wchar_t** as the underlying type.

```
    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& a, fixed& b);
std::istream& operator>>(std::istream& is, fixed& obj);
std::ostream& operator<<(std::ostream& os, const fixed& obj);
fixed operator+(const fixed& lhs, const fixed& rhs);
fixed operator-(const fixed& lhs, const fixed& rhs);
fixed operator*(const fixed& lhs, const fixed& rhs);
fixed operator/(const fixed& lhs, const fixed& rhs);
bool operator>(const fixed& lhs, const fixed& rhs);
bool operator<(const fixed& lhs, const fixed& rhs);
bool operator>=(const fixed& lhs, const fixed& rhs);
bool operator<=(const fixed& lhs, const fixed& rhs);
bool operator==(const fixed& lhs, const fixed& rhs);
bool operator!=(const fixed& lhs, const fixed& rhs);

}} // namespace omg::types
```

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 effectively constant digits and scale, that provides the same C++ interface and can be implicitly converted from/to the **`fixed`** class. The name of this alternative class, which may be a template instantiation, is not defined by this mapping.

Fixed types shall provide the following type traits to facilitate template metaprogramming:

Table 7.7: Additional Type Traits for the Fixed Type

Member	Definition
`template <typename T>` `struct digits;`	Inherits **`std::integral_constant<size_t, b>`** where **`b`** indicates the number of digits of the fixed type.
`template <typename T>` `struct scale;`	Inherits **`std::integral_constant<size_t, b>`** where **`b`** indicates the scale of the fixed type.

For example, the following IDL:

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

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

would map to:

```
using F = omg::types::Fixed_Or_Implementation_Defined_Type;

class AnInterface {
```

```
public:
    // ...
    virtual void op(const F& arg) = 0;
    // ...
};
```

7.2.4.3 Constructed Types

7.2.4.3.1 Structures

An IDL **`struct`** shall be mapped to a C++ **`struct`** with the same name. The mapped struct shall provide:

- A default constructor that initializes all built-in data types[3] to their default value as specified in Clause 7.2.4.1, enumerators to their first value, and the rest of members using their default constructor.
- A copy constructor that performs a deep copy from the existing constructed type to create a new constructed type.
- A move constructor that moves all members to their corresponding members.
- A copy assignment operator that performs a deep copy to create the new constructed type with strong type safety.
- A move assignment operator.
- A set of comparison operators, including at least "equal to" and "not equal to."
- A destructor that releases all members.

Implementers of this specification may delegate the creation of such methods on the compiler. Each member field of the IDL **`struct`** shall be mapped to a public member of the C++ **`struct`**, with the same name and equivalent C++ type according to the mapping rules specified in this document. C++ **`struct`** members shall appear in the same order as the corresponding IDL **`struct`** members

Moreover, the mapping shall provide a **`swap`** function in the namespace of the C++ **`struct`** definition with the following signature:

```
void swap(<StructName>& a, <StructName>& b);
```

For example, the IDL **`struct`** declaration below:

```
struct MyStruct {
    long a_long;
    short a_short;
};
```

would map to the following C++:

```
struct MyStruct {
    int32_t a_long {};
    int16_t a_short {};
};
```

[3] Also known as fundamental types, built-in types include the boolean type, integer types, character types, and floating point types. For more information, see https://en.cppreference.com/w/cpp/language/types. Built-in types have a one-to-one correspondence to the IDL Basic Types specified in Clause 7.2.4.1.

```
void swap(MyStruct& a, MyStruct& b) {...}
```

7.2.4.3.2 Unions

An IDL union shall be mapped to a C++ **class** with the same name. The **class** shall provide the same constructors, destructors, and operators for mapped structures defined in Clause 7.2.4.3.1.

Moreover, the mapped class shall provide:

- A public accessor constant method named **_d()** that returns the value of the discriminator, with the following signature:
  <DiscriminatorType> _d() const;
- A public modifier method named **_d()** that sets the value of the discriminator with the following signature:
  void _d(<DiscriminatorType> value);
  Setting the discriminator to an invalid value (e.g., to a value that changes the union member that is currently selected) may result in an error.
  NOTE—The purpose of this method is to reset the discriminator value without having to reset the value of a union member that may already be selected. To set a union member that can be selected by more than one case label, providing the appropriate discriminator value, the C++ **class** provides a public modifier method that is defined below.
- For each union member:
  - A public accessor method with the name of the union member:
    - If the equivalent type of the member after resolving any **typedef** results in a C++ built-in type or an **enum**, the accessor shall return by value:
      <MemberType> <MemberName>();
    - Otherwise, the accessor shall return a reference to its value:
      <MemberType>& <MemberName>();
  - A public constant accessor method with the name of the union member:
    - If the equivalent type of the member after resolving any **typedef** results in a C++ built-in type or an **enum**, the method shall return by value:
      <MemberType> <MemberName>() const;
    - Otherwise, the method shall return a constant reference to its value:**const <MemberType>& <MemberName>() const;**

    Accessing an invalid union member may result in an undefined error.
  - A modifier method that takes as a parameter the value to which the member field shall be set, and sets the discriminator to the appropriate value.
    - If the equivalent type of the member after resolving any **typedef** is a C++ built-in type, or an **enum**, the argument shall be passed by value:
      void <MemberName>(<MemberType> value);
    - Otherwise, the argument shall be passed by reference to **const**, and a move-enabled modifier method shall be provided:
      void <MemberName>(const <MemberType>& value);
      void <MemberName>(<MemberType>&& value);

- For each union member that can be selected by more than one case label, the C++ **`class`** shall include a modifier method that allows selecting the corresponding discriminator value with an extra parameter passed by value named **`discriminator`** of the discriminator type.
    - If the equivalent type member after resolving any **`typedef`** is a C++ built-in type, or an **`enum`**, the argument shall be passed by value:
      ```
      void <MemberName>(<MemberType>& value, <DiscriminatorType> discriminator);
      ```
    - Otherwise, the argument shall be passed by reference to **`const`**, and a move-enabled modifier method shall be provided:
      ```
      void <MemberName>(
              const <MemberType>& value,
              <DiscriminatorType> discriminator);
      void <MemberName>(
              <MemberType>&& value,
              <DiscriminatorType> discriminator);
      ```
- If the union has a default case, the default constructor shall initialize the discriminator, and the selected member field following the initialization rules described in Clause 7.2.4.3.1. If it does not, the default constructor shall initialize the union to the first discriminant value specified in the IDL definition.
- If the IDL union definition has an implicit default member (i.e., if the union does not have a default case and not all permissive discriminator values are listed), the class shall provide a method named **`_default()`** that explicitly sets the discriminator value to a legal default value.
  ```
  void _default();
  ```

Moreover, the mapping shall provide a **`swap`** method in the scope of the C++ **`class`** definition with the following signature:

```
void swap(<UnionName>& a, <UnionName>& b);
```

For example, the IDL **`union`** declaration below:

```
union AUnion switch (octet) {
    case 1:
        long  a_long;
    case 2:
    case 3:
        short a_short;
    case 4:
        AStruct a_struct;
    default:
        octet a_byte_default;
};
```

would map to the following C++:

```
class AUnion {
public:
    AUnion() {...}
    AUnion(const AUnion& other) {...}
    Aunion(AUnion&& other) {...}
    ~AUnion() {...}

    AUnion& operator=(const AUnion&) {...}
    AUnion& operator=(AUnion&& other) {...}

    uint8_t _d() const {...}
    void _d(uint8_t value) {...}

    void a_long(int32_t value) {...}
```

```
    int32_t& a_long() {...}
    int32_t a_long() const {...}

    void a_short(int16_t value) {...}
    void a_short(int16_t value, uint8_t discriminator) {...}
    int16_t& a_short() {...}
    int16_t a_short() const {...}

    void a_struct(const AStruct& value) {...}
    AStruct& a_struct() {...}
    const AStruct& a_struct() const {...}

    void a_byte_default(uint8_t value) {...}
    uint8_t& a_byte_default() {...}
    uint8_t a_byte_default() const {...}
};

void swap(AUnion& a, AUnion& b) {...}
```

Likewise, the IDL declaration below:

```
union AUnion switch (octet) {
    case 1:
        long  a_long;
    case 2:
    case 3:
        short a_short;
};
```

would map to the following C++:

```
class AUnion {
public:
    AUnion() {...}
    AUnion(const AUnion& other) {...}
    Aunion(AUnion&& other) {...}
    ~AUnion() {...}

    AUnion& operator=(const AUnion&) {...}
    AUnion& operator=(AUnion& other) {...}

    uint8_t _d() const {...}
    void _d(uint8_t value) {...}

    void a_long(int32_t value) {...}
    int32_t& a_long() {...}
    int32_t a_long() const {...}

    void a_short(int16_t value) {...}
    void a_short(int16_t value, uint8_t discriminator) {...}
    int16_t& a_short() {...}
    int16_t a_short() const {...}

    void _default() {...}
};

void swap(AUnion& a, AUnion& b) {...}
```

7.2.4.3.3 Enumerations

An IDL **enum** shall be mapped to a C++ scoped **enum** class with the same name as the IDL **enum** type.

Enumerations shall provide the following type traits to facilitate template metaprogramming if they are preceded by the **`@bit_bound`** annotation (see Clause 7.17.4):

Table 7.8: Additional Type Traits for Enumerations

Member	Definition
`template<>` `struct bit_bound;`	Inherits **`std::integral_constant<uint32_t, b>`** where **`b`** indicates the **`bit_bound`** of the **`enum`**
`template<>` `struct underlying_type;`	Defines **`type`** - Type mapped as the underlying type of the enum.

For example, the IDL **`enum`** declaration below:

```
enum AnEnum {
    zero,
    one,
    two
};
```

would map to the following C++:

```
enum class AnEnum {
    zero,
    one,
    two
};
```

7.2.4.3.4 Constructed Recursive Types

Constructed recursive types are supported by mapping the involved types directly to C++ as described elsewhere in Clause 7.

7.2.4.4 Arrays

An IDL array shall be mapped to a C++ **`std::array`** of the mapped element type, or to a type named **`omg::types::array`** that delivers **`std::array`** semantics. For interoperability purposes, implementers shall always define **`omg::types::array`**, which may be declared as an alias to **`std::array`**. Other implementations of **`omg::types::array`** shall support conversion to and from **`std::array`**. Multidimensional arrays shall be constructed nesting **`std::array`** or **`omg::types::array`** definitions.

Arrays shall provide the following type traits to facilitate template metaprogramming:

Table 7.9: Additional Type Traits for Arrays

Member	Definition
`template <typename T>` `struct dimensions;`	Inherits **`std::integral_constant<size_t, b>`** where **`b`** indicates the number of dimensions of the array.

For example, the IDL declaration below:

```
typedef long long_array[100];
typedef string string_array[1][2];
```

would map to the following C++:

```
using long_array = std::array<int32_t, 100>;
```

```
using string_array = std::array<std::array<std::string, 2>, 1>;
```

7.2.4.5 Native Types

IDL provides a declaration to define an opaque type whose representation is specified by the language mapping. This language mapping specification does not define any native types, but compliant implementations may provide the necessary mechanisms to map native types to equivalent type names in C++.

7.2.4.6 Naming Data Types

IDL **typedef**s shall be mapped to type alias declarations.

For example the IDL declaration below:

```
typedef long Length;

struct MyType {
    Length my_type_length;
};
```

would map to the following C++:

```
using Length = int32_t;

struct MyType {
    // ...
    Length my_type_length;
    //...
};
```

7.3 Any

The IDL any type shall be mapped to a C++ **omg::types::Any** type. The implementation of the **omg::types::Any** is platform-specific, and shall include operations that allow programmers to insert and access the value contained in an **any** instance as well as the actual type of that value.

7.4 Interfaces – Basic

Each IDL **interface** shall be mapped to a C++ **class** with the same name as the IDL **interface**. If the IDL **interface** derives from other IDL **interface**s, the equivalent C++ class shall be declared to extend the C++ classes resulting from mapping the base interfaces.

Each attribute defined in the IDL **interface** shall map to a pair of pure virtual methods in the C++ class, an accessor and a modifier, with the same name as the attribute. The accessor method shall return the C++ equivalent type of the attribute according to the mapping rules defined in this chapter. The modifier method shall take as an argument the value to be set. If the equivalent type of the attribute after resolving any **typedef** is a C++ built-in type, or an **enum**, the argument shall be passed by value. Otherwise, the argument shall be passed as a reference to **const**. If the attribute is read only, the mapping shall omit the modifier.

Each operation defined in the IDL **interface** shall map to a pure virtual method in the C++ **class**. The name of the mapped method shall be the name of the IDL operation. The number and order of the parameters to the mapped method shall be the same as in the IDL operation. The name of the method parameters shall be the name of the IDL method argument. The type of the method parameter shall be mapped following the mapping rules defined in this chapter for the specific type. IDL **out** and **inout** arguments shall be passed by reference. IDL **in** arguments shall be passed by value if the resulting type after resolving any **typedef** C++ is a built-in type or an **enum**, and as a reference to **const**

otherwise. Parameters of interface type[4] **`T`** shall be mapped to **`std::shared_ptr<T>`**, or to a type named **`omg::types::ref_type<T>`** that delivers **`std::shared_ptr<T>`** semantics. Implementers shall also provide a type named **`omg::types::weak_ref_type<T>`** that delivers **`std::weak_ptr<T>`** semantics.

For example, the IDL **`interface`** declaration below:

```
interface AnInterface {
    attribute long attr;
    readonly attribute long ro_attr;
    void op1(
            in long i_param,
            in MyStruct si_param,
            inout long io_param,
            inout MyStruct ios_param,
            out long o_param,
            out MyStruct so_param);
}
```

would map to the following C++:

```
class AnInterface {
public:
    virtual void attr(int32_t value) = 0;
    virtual int32_t attr() const = 0;

    virtual int32_t ro_attr() const = 0;

    virtual void op1(
            int32_t i_param,
            const MyStruct& si_param,
            int32_t& io_param,
            MyStruct& ios_param,
            int32_t& o_param,
            MyStruct& so_param) = 0;
};
```

7.4.1 Exceptions

An IDL **`exception`** shall be mapped to a C++ **`class`** with the same name as the IDL exception. The mapped class shall extend the **`std::exception`** class.

All exception members shall be initialized to their default value by the default constructor for the exception. All exception members shall be initialized to their default value by the default constructor for the exception. The mapped class shall also provide a copy constructor, move constructor, assignment operator, move operator, and destructor, which shall automatically copy, move, or free the storage associated with the exception. Also, the mapped class shall provide an implementation of the **`what()`** method. For convenience, the mapping shall also define an explicit constructor with one parameter for each exception member, which shall initialize the exception members to the given values, and shall also include a parameter of type **`const char *`** to provide explanatory information. Implementers may provide additional constructors to provide explanatory information as well.

For example, the IDL declarations below:

```
exception AnException {
    long error_code;
};
interface MyInterfaceException {
    void op1(in long in_param) raises(AnException);
```

[4] As described in Clause 7.4.3.4.3.1 of [OMG-IDL4], parameters of interface type are semantically a reference to an object implementing that interface.

```
};
```

would map to the following C++:

```
class AnException : public std::exception {
public:
    AnException() {...}
    AnException(const AnException&) {...}
    AnException(AnException&&) {...}
    AnException(int32_t error_code, const char* what) {...}
    ~AnException() override {...}
    AnException& operator=(const AnException&) {...}
    AnException& operator=(AnException&&) {...}

    void error_code(int32_t value) {...}
    int32_t error_code() const {...}

};

class MyInterfaceException {
    void op1(int32_t in_param);
};
```

7.4.2 Interface Forward Declaration

An IDL interface forward declaration shall be mapped to a partial forward declaration in C++.

For example, the interface forward declaration below:

```
interface Foo;
```

would map to the following C++:

```
class Foo;
```

7.5 Interfaces – Full

This building block complements Interfaces – Basic adding the ability to embed declarations such as types, exceptions, and constants in the interface body.

Interfaces – Full shall follow the mapping rules for Interfaces – Basic, adding to mapped class the declaration every type, exception, or constant declaration in the IDL interface body. Types, exceptions, and constants shall be mapped according to the mapping rules specified in this chapter. In the case of constants, **`constexpr`** declarations shall be marked as **`static`**.

For example, the IDL interface declaration below:

```
interface FullInterface {
    struct S {
        long a_long;
    };
    const double PI = 3.14;
    void op1(in S s_in);
    attribute long an_attribute;
};
```

would map to the following C++:

```
class FullInterface {
public:
    struct S {
        int32_t a_long;
    };
```

```
    static constexpr double PI = 3.14;
    virtual void op1(const S& s_in) = 0;
    virtual void attribute(int32_t value) = 0;
    virtual int32_t an_attribute() const = 0;
```

7.6 Value Types

An IDL **valuetype** shall be mapped to a C++ **class** with the same name as the IDL **valuetype**. Public state members shall be mapped to public pure virtual accessor and modifier methods of the C++ **class**. Private state members shall be mapped to protected pure virtual accessor and modifier methods. If the **valuetype** contains factory operations, the mapping shall declare a class named **<ValueTypeName>_factory** in the same scope as the C++ class that represents the **valuetype**. **<ValueTypeName>_factory** shall declare a pure virtual method for every factory operation, returning either **std::shared_ptr<T>** or **omg::types::ref_type<T>**, where **T** is the mapped **valuetype** class, and accepting the specified in parameters. The **<ValueTypeName>_factory** class shall also provide a protected virtual destructor, and a protected constructor.

If the IDL **valuetype** inherits from a base **valuetype**, the mapped class shall inherit the class that resulted from mapping the base **valuetype** as public virtual. If the IDL **valuetype** supports an **interface** type, all the operations in the interface, and corresponding base interfaces if any, shall be mapped to pure virtual methods in the mapped class.

For example, the IDL **valuetype** declarations below:

```
valuetype VT1 {
    attribute long a_long_attr;
    void vt_op(in long p_long);
    public long a_public_long;
    private long a_private_long;
    factory vt_factory(in long a_long, in short a_short);
};

interface MyInterface {
    void op();
};

valuetype VT2 : VT1 supports MyInterface {
    public long third_long;
};
```

would map to the following C++:

```
class VT1 {
public:
    virtual void a_long_attr(int32_t value) = 0;
    virtual int32_t a_long_attr() const = 0;


    virtual void vt_op(int32_t p_long) = 0;

    virtual void a_public_long(int32_t value) = 0;
    virtual int32_t a_public_long() const = 0;
    virtual int32_t& a_public_long() = 0;


protected:
    virtual void a_private_long(int32_t value) = 0;
    virtual int32_t a_private_long() const = 0;
    virtual int32_t& a_private_long() = 0;
};
```

```
class VT1_factory {
public:
    virtual omg::types::ref_type<VT1> vt_factory(int32_t a_long, int16_t a_short) = 0;
protected:
    ~VT1_factory() override;
    VT1_factory();
};

class VT2 : public virtual VT1 {
public:
    virtual void op() = 0;

    virtual int32_t third_long() const = 0;
    virtual int32_t& third_long() const = 0;
    virtual void third_long(int32_t value) = 0;
};
```

7.7 CORBA-Specific – Interfaces

CORBA-specific mappings are defined in Clause A.1 of Annex A: Platform-Specific Mappings.

7.8 CORBA-Specific – Value Types

CORBA-specific mappings are defined in Clause A.1 of Annex A: Platform-Specific Mappings.

7.9 Components – Basic

Basic components have no direct language mapping; they shall be mapped to equivalent IDL, as specified in [OMG-CORBA-COMP], and mapped to C++ accordingly.

7.10 Components – Homes

Homes have no direct language mapping; they shall be mapped to equivalent IDL, as specified in [OMG-CORBA-COMP], and mapped to C++ accordingly.

7.11 CCM-Specific

CORBA-specific mappings are defined in Clause A.1 of Annex A: Platform-Specific Mappings.

7.12 Components – Ports and Connectors

Ports and Connectors have no direct language mapping; they shall be mapped to intermediate IDL, as specified in [OMG-IDL4], and mapped to C++ accordingly.

7.13 Template Modules

Template module instances have no direct language mapping; they shall be mapped to intermediate IDL, as specified in [OMG-IDL4], and mapped to C++ accordingly.

7.14 Extended Data Types

7.14.1 Structures with Single Inheritance

An IDL **struct** that inherits from a base IDL **struct** shall be mapped to a C++ **struct** with the same name, and shall inherit from the mapped base **struct** using public inheritance.

The mapping shall follow the rules defined in Clause 7.2.4.3.1, including the definition of a **swap** method in the namespace of the **struct**, which shall ensure that all inherited members are swapped.

For example, an IDL struct extending the **MyStruct** structure defined in Clause 7.2.4.3.1:

```
struct ChildStruct : MyStruct {
    float a_float;
};
```

would map to the following C++:

```
struct ChildStruct : MyStruct {
    float a_float {};
};

void swap(ChildStruct& a, ChildStruct& b) {...}
```

7.14.2 Union Discriminators

This building block adds the **wchar**, and **octet** IDL types to the set of valid types for a discriminator. The mapping of union discriminators of such types shall be mapped as specified in Clause 7.2.4.3.2.

NOTE—Any addition to the list of supported integer, char, boolean, or enum types as a result of the implementation of the Extended Data Types building block, makes such types valid union discriminators as well. For example, if 8-bit integer values are supported (see Clause 7.14.4), **int8** and **uint8** are valid union discriminators and shall be mapped as specified in Clause 7.14.4.

7.14.3 Additional Template Types

7.14.3.1 Maps

IDL maps shall be mapped to a C++ **std::map<Key,T>** instantiated with the mapped C++ key type **Key** and mapped value type **T**, or to a type named **omg::types::map<Key,T>** that delivers **std::map<Key,T>** semantics. For interoperability purposes, implementers shall always define **omg::types::map<Key,T>**, which may be declared as an alias to **std::map<Key,T>**. Other implementations of **omg::types::map<Key,T>** shall support conversion to and from **std::map<Key,T>**.

Bounded maps may be mapped to a type named **omg::types::bounded_map<Key, T, N>** (where **Key** is the key type, **T** the mapped value type, and **N** the bound size), which delivers the semantics of a **std::map<Key, T>** and supports transparent conversion to and from it.

NOTE—Implementers of this specification shall provide a bound checking mechanism for bounded **maps** (e.g., at serialization time). Please refer to Annex A for platform-specific bound checking mechanisms.

Maps shall provide the following specializations for the type traits defined in Clause 7.1.4:

Table 7.10: Type Trait Specialization for Maps

Member	Definition
`template<>` `struct is_bounded;`	Inherits `std::true_type` if the map is bounded and `std::false_type` if the map is unbounded. When mapping to `std::map<Key,T>`, `is_bounded` shall inherit `std::false_type`.
`template<>` `struct bound;`	Inherits `std::integral_constant<size_t, b>` where `b` is the value of the bound. For an unbounded map type, the value of `b` shall be `std::numeric_limits<std::size_t>::max()`.

Maps shall also providethe following type traits to facilitate template metaprogramming:

Table 7.11: Additional Type Trait Definitions for Maps

Member	Definition
`template <typename T>` `struct key;`	Defines `type` - Type to be used as `key` type.
`template <typename T>` `struct elements;`	Defines `type` - Type to be used as elements type.

For example, the IDL declarations below show the mapping from IDL bounded and unbounded maps assuming `std::map<Key,T>` as the mapping for both:

```
typedef map<unsigned long, MyStruct> M1; // unbounded map
typedef map<string, MyStruct, 20> M2;    // bounded map
```

would map to the following C++:

```
using M1 = std::map<uint32_t, MyStruct>;
using M2 = std::map<std::string, MyStruct>;
```

7.14.3.2 Bitsets

IDL `bitset` types shall be mapped to C++ `struct`s that meet the C++ requirements for aggregate initialization. The only members of these `struct`s are bit fields. In cases where the IDL `bitset` inherits from a base type, the mapped `struct` shall derive (using public inheritance) from the `struct` resulting from mapping the base type.

The mapped type's bit field members directly correspond to the IDL `bitfield`s, including the use of anonymous (nameless) bit fields. The C++ data type for each bit field is the mapped type of the IDL `bitfield` destination type, according to the mapping rules specified in this chapter.

For example, the IDL declarations below:

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

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

would map to the following C++:

```
struct BitSet1 {
    bool bit0 : 1;
```

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

struct BitSet2 : BitSet1 {

    uint8_t : 3;
    bool bit_7 : 1;
};
```

7.14.3.3 Bitmask Type

IDL **bitmask** declarations shall be mapped to a C++ **struct** type containing the following elements:

- An unscoped enum named **_flags** with an explicitly defined underlying type. That underlying type is the smallest mapped unsigned integer type that has sufficient bits for the **bit_bound** of the **bitmask**: **uint8_t**, for values between 1 and 8; **uint16_t** for bit_bound values between 9 and 16; **uint32_t**, for values between 17 and 32; and **uint64_t** for values between 33 and 64. The **_flags** enumerators are the values defined in the scope of the IDL **bitmask**, with each enumerator explicitly initialized to its corresponding integer value. NOTE—The exact form of this initialization expression (for example, use of non-decimal bases or bit shifts or other operators) is implementation-defined.
- A private member named **_value** of the underlying type of **_flags** (i.e., **uint8_t**, **uint16_t**, **uint32_t**, or **uint64_t** depending on the **bit_bound**).
- A default constructor and a copy constructor.
- An implementation of the **!=**, **&=**, and **^=** bitwise operators.
- An implementation of the function call operator that returns **_value**.

Bitmasks shall provide the following type traits to facilitate template metaprogramming.

Table 7.12: Additional Type Traits for Bitmasks

Member	Definition
`template<>` `struct bit_bound;`	Inherits **std::integral_constant<uint32_t, b>** where **b** indicates the **bit_bound** of the **bitmask**.
`template<>` `struct underlying_type;`	Defines **type** - Type mapped as the underlying type of the bit mask (i.e., of the **_flags enum**).

For example, the IDL bitmask declaration below:

```
@bit_bound(32)
bitmask MyBitMask {
    @position(0) flag0,
    @position(1) flag1

};
```

would map to the following C++:

```
struct MyBitMask {
    enum MyBitMaskBits : std::uint32_t {
        flag0 = 0x01 << 0,
        flag1 = 0x01 <<

    };
```

```
    MyBitMask() : _value(0U) {}
    MyBitMask(std::uint32_t v) : _value(v) {}

    operator uint32_t()
    {
        return _value;
    }

    MyBitMask& operator|=(std::uint32_t other)
    {
        _value != other;
        return *this;
    }

    MyBitMask& operator&=(std::uint32_t from)
    {
        _value &= other;
        return *this;
    }

    MyBitMask& operator^=(std::uint32_t other)
    {
        _value ^= other;
        return *this;
    }

private:
    uint32_t _value;
};
```

7.14.4 8-bit Integer Types

8-bit integer types shall be mapped as shown in Table 7.13. The default value of 8-bit integer types is 0.

Table 7.13: Mapping of 8-bit Integer Types

IDL Type	C++ Type
`int8`	`int8_t`
`uint8`	`uint8_t`

7.14.5 Explicitly-Named Integer Types

Explicitly-named integer types shall be mapped as shown in Table 7.14. The default value of explicitly-named integer types is 0.

Table 7.14: Mapping of Explicitly-Named Integer Types

IDL Type	C++ Type
`int16`	`int16_t`
`uint16`	`uint16_t`
`int32`	`int32_t`
`uint32`	`uint32_t`

IDL Type	C++ Type
`int64`	`int64_t`
`uint64`	`uint64_t`

7.15 Anonymous Types

No impact to the C++language mapping.

NOTE—For anonymously typed members in a **`struct`**, it is possible to use the C++ **`decltype`** to acquire the declared type of each member and use it in a traits declaration. For example:

```
struct S {
    std::vector<int32_t> v;
    std::vector<int32_t> w;
};

traits<decltype(S::v)>::value_type x;
x.push_back(1234);
```

7.16 User-Defined Annotations

User-defined annotations are not propagated to the generated C++ code.

7.17 Standardized Annotations

[OMG-IDL4] defines some annotations and assigns them to logical groups. These annotations may be applied to various constructs throughout an IDL document, and their impact on the language mapping is dependent on the context in which they are applied. The following sections describe the impact these defined annotations have on the language mapping, and provide cross references to earlier document sections where the details are given.

7.17.1 Group of Annotations: General Purpose

Table 7.15 identifies the mapping impact of the IDL defined General Purpose Annotations.

Table 7.15: General Purpose Annotation Impact

General Purpose Annotation	Impact on C++ Language Mapping
`@id`	No impact on language mapping
`@autoid`	No impact on language mapping
`@optional`	IDL elements annotated with **`@optional`** of type **`T`** shall map to: • **`std::optional<T>`** in implementations of this specification supporting C++17 [ISO/IEC-14882:2017] and above. • **`omg::types::optional<T>`** in implementations that need to remain compliant with earlier versions of the C++ standard. The implementation of **`omg::types::optional<T>`** shall deliver **`std::optional<T>`**'s semantics and support transparent conversion to and from **`std::optional<T>`**.
`@position`	Impacts the mapping of **`bitmask`** types as defined in Clause 7.14.3.3.

`@value`	Impacts the mapping of **`enum`** types, providing the value of the annotated enumerator.
	For example:
	`enum Color {`
	`  @value(1) red,`
	`  @value(2) green,`
	`  @value(3) blue`
	`};`
	would map to the following C++:
	`enum class Color {`
	`  red = 1,`
	`  green = 2,`
	`  blue = 3`
	`};`

`@extensibility`	No impact on language mapping
`@final`	No impact on language mapping
`@mutable`	No impact on language mapping
`@appendable`	No impact on language mapping

7.17.2 Group of Annotations: Data Modeling

Table 7.16 identifies the mapping impact of the IDL defined Data Modeling Annotations.

Table 7.16: Data Modeling Annotation Impact

Data Modeling Annotation	Impact on C++ Language Mapping
`@key`	No impact on language mapping.
`@must_understand`	No impact on language mapping.
`@default_literal`	The C++ element declared as result of the mappings defined in this specification shall be initialized to the element indicated by the annotation.

7.17.3 Group of Annotations: Units and Ranges

Table 7.17 identifies the mapping impact of the IDL defined Units and Ranges Annotations.

Table 7.17: Units and Ranges Annotation Impact

Units and Ranges Annotation	Impact on C++ Language Mapping
`@default`	C++ elements declared as result of the mappings defined in this specification containing a **`@default`** annotation shall be initialized to the value of the annotation.
`@range`	IDL elements annotated with **`@range(min, max)`** shall map to C++ **`omg::types::ranged<T, min, max>`** elements, where **`T`** is C++ type equivalent to that of the IDL element.

	The implementation of omg::types::ranged<T, min, max> shall perform the corresponding checks and throw std::out_of_range if fail the value assigned to a ranged element is out of the predefined range.
	
	For example:
	`struct StructureOfRangedValues {`
	`  @range(min=-10, max=10) long x;`
	`};`
	
	would map to:
	`struct StructureOfRangedValues {`
	`  omg::types::ranged<int32_t, -10, 10> x;`
	`};`

`@min`	C++ elements declared as a result of the mappings defined in this specification containing a **`@min`** annotation shall throw a **`std::out_of_range`** if they are set to a value smaller than the value of the **`@min`** annotation. Therefore, the setter for a member created as a result of an IDL element annotated with **`@min`** shall implement the corresponding checks and throw **`std::out_of_range`** if the check fails.
`@max`	C++ elements declared as a result of the mappings defined in this specification containing a **`@max`** annotation shall throw a **`std::out_of_range`** if they are set to a value bigger than the value of the **`@max`** annotation. Therefore, the setter for a member created as a result of an IDL element annotated with **`@max`** shall implement the corresponding checks and throw **`std::out_of_range`** if the check fails.
`@unit`	No impact on language mapping.

7.17.4 Group of Annotations: Data Implementation

Table 7.18 identifies the mapping impact of the IDL defined Data Implementation Annotations.

Table 7.18: Data Implementation Annotation Impact

Data Implementation Annotation	Impact on C++ Language Mapping

`@bit_bound`	If an IDL **`enum`** declaration is preceded by the **`@bit_bound`** annotation, the equivalent C++ scoped **`enum`** class shall have the following underlying type: **`int8_t`**, for bit bound values between 1 and 8; **`int16_t`**, for bit bound values between 9 and 16; **`int32_t`**, for bit bound values between 17 and 32; and **`int64_t`** for bit bound values between 33 and 64.
	
	For example, the IDL **`enum`** declaration below:
	`@bit_bound(6)`
	`enum ABoundEnum {`
	`  @value(1) one,`
	`  @value(2) two`
	`};`

	would map to the following C++:
	`enum class AboundEnum : int8_t {`
	`  one = 1,`
	`  two = 2`
	`};`
	The mapping for an IDL **`bitmask`** declaration preceded by the **`@bit_bound`** annotation is described in Clause 7.14.3.3.

`@external`	IDL member declarations preceded by the **`@external`** annotation shall be mapped to **`std::shared_ptr<T>`** or to the equivalent **`omg::types::ref_type<T>`**, where **`T`** is the type of the IDL external member. Structures containing members annotated with the **`@external`** annotation shall make a deep copy of the external member in their copy constructor.
`@nested`	No impact on the language mapping

7.17.5 Group of Annotations: Code Generation

Table 7.19 identifies the mapping impact of the IDL defined Code Generation Annotations.

Table 7.19: Code Generation Annotation Impact

Code Generation Annotation	Impact on C++ Language Mapping
`@verbatim`	Copies verbatim text to the indicated output position when the indicated language is **`"*"`**, **`"c++"`**, **`"cpp"`**, **`"cc"`**, or **`"cxx"`**.

7.17.6 Group of Annotations: Interfaces

Table 7.20 identifies the mapping impact of the IDL defined Interface Annotations.

Table 7.20: Interface Annotation Impact

Interface Annotation	Impact on C++ Language Mapping
`@service`	Options are **`"CORBA"`**, **`"DDS"`**, **`"*"`**. Impact is platform-specific.
`@oneway`	Impact is platform-specific.
`@ami`	Impact is platform-specific.

8 IDL to C++ Language Mapping Annotations

This chapter defines specialized annotations that extend the standard set defined in [OMG-IDL4] to control the C++ code generation.

8.1 @cpp_mapping Annotation

This annotation provides the means to customize the way a number of IDL constructs are mapped to the C++ programming language. This annotation can therefore be used to modify the default mapping behavior of the mappings specified in Chapter 7.

The IDL definition of the **`@cpp_mapping`** annotation is:

```
@annotation cpp_mapping {
    enum StructMapping {
        STRUCT_WITH_PUBLIC_MEMBERS,
        CLASS_WITH_PUBLIC_ACCESSORS_AND_MODIFIERS
    };
    StructMapping struct_mapping default STRUCT_WITH_PUBLIC_MEMBERS;
};
```

The behavior associated with each parameter is defined below.

8.1.1 struct_mapping Parameter

`struct_mapping` defines the mapping strategy for IDL **`struct`**s. By default, as specified in Clause 7.2.4.3.1, IDL structs are mapped to C++ **`struct`**s with public members; in other words **`struct_mapping`** defaults to **`STRUCT_WITH_PUBLIC_MEMBERS`**.

`CLASS_WITH_PUBLIC_ACCESSORS_AND_MODIFIERS` changes the default behavior, mapping the annotated IDL struct to C++ **`class`** where all members are only accessible via public accessor methods, and may only be modified through public modifier methods.

- Accessor methods shall have the same name as the IDL element, shall return the C++ equivalent type, according to the mapping rules specified in Chapter 7, and shall be declared as **`const`**. Additionally, the implementation shall provide non-const accessors of the same name that return a reference to the member.
- Modifier methods shall also have the same name as the IDL element, and shall take the value to be set as an argument:
    - If the equivalent type of the member after resolving any **`typedef`** results in a C++ built-in type or an **`enum`**, the argument shall be passed by value:
      `void <MemberName>(<MemberType> value);`
    - Otherwise, the argument shall be passed by reference to **`const`**, and a move-enabled modifier method shall be provided:
      `void <MemberName>(const <MemberType>& value);`
      `void <MemberName>(<MemberType>&& value);`

Implementers of this specification may add additional constructors and operator overloads in mappings resulting from setting **`struct_mapping`** to **`CLASS_WITH_PUBLIC_ACCESSORS_AND_MODIFIERS`** parameter. However, at a minimum, they shall provide a constructor that allows the initialization of every element of the **`struct`**. The argument provided to the constructor shall be passed by value or by reference to **`const`** according to the rules applied to modifier methods above.

For example, the IDL **struct** declaration below:

```
struct MyStruct {
    long a_long;
    short a_short;
    string a_string;
};
```

would map to the following C++:

```
class MyStruct {
public:
    MyStruct(int32_t a_long, int16_t a_short, const std::string& a_string);

    void a_long(int32_t value);
    int32_t a_long() const;
    int32_t& a_long();

    void a_short(int16_t value);
    int16_t a_short() const;
    int16_t& a_short();

    void a_string(const std::string& value);
    std::string a_string() const;
    std::string& a_string();

private:
    int32_t a_long_;
    int16_t a_short_;
    std::string a_string_;
};
```

Annex A: Platform-Specific Mappings

(normative)

A.1 CORBA-Specific Mappings

This clause describes platform-specific mapping rules that shall be followed when mapping IDL constructs to the C++ programming language for CORBA. These mappings rules are built upon the platform-independent rules defined in Chapters 7 and 8 for the building blocks that compose the CORBA profiles defined in Clause 9.2 of [OMG-IDL4].

A.1.1 Traits

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

Table A.1: CORBA::traits<> common member types

Member	Definition
`value_type`	The template parameter T.
`in_type`	The type to be used for an in parameter.
`out_type`	The type to be used for an out parameter.
`inout_type`	The type to be used for an inout parameter.

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

A.1.1.1 Interfaces

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

Table A.2: CORBA::traits<> additional member types for Interfaces

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 (e.g., **`omg::types::ref_type<T>`**).
`weak_ref_type`	Weak reference type (e.g., **`omg::types::weak_ref_type<T>`**).

A.1.1.2 Strings

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

Table A.3: CORBA::traits<> additional member types for Unbounded Strings

Member	Definition
`element_traits`	The element type (e.g., `CORBA::traits<char>` or `CORBA::traits<wchar_t>`) of the string.
`is_bounded`	`std::false_type` type indicating that the string is not bounded.

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

Table A.4: CORBA::traits<> additional member types for Bounded Strings

Member	Definition
`element_traits`	The element type (e.g., `CORBA::traits<char>` or `CORBA::traits<wchar_t>`) of the string.
`is_bounded`	`std::true_type` type indicating that the string is bounded.
`bound`	`std::integral_constant` type of value type `uint32_t` indicating the bound of the string.

A.1.1.3 Sequences

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

Table A.5: CORBA::traits<> additional member types for Unbounded Sequences

Member	Definition
`element_traits`	The element type (e.g., `CORBA::traits<T>`) of the sequence.
`is_bounded`	`std::false_type` type indicating that the sequence is unbounded.

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

Table A.6: CORBA::traits<> additional member types for Bounded Sequences

Member	Definition
`element_traits`	The element type (e.g., `CORBA::traits<T>`) of the sequence.
`is_bounded`	`std::true_type` type indicating that the sequence is bounded.
`bound`	`std::integral_constant` type of value type `uint32_t` indicating the bound of the sequence.

A.1.1.4 Arrays

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

Table A.7: CORBA::traits<> additional member types for Arrays

Member	Definition
`element_traits`	The element type (e.g., `CORBA::traits<T>`) of the array.
`dimensions`	`std::integral_constant` type of value type `uint32_t` indicating the number of dimensions of the array.

The element_traits for a multidimensional array shall be the type trait for the core type. For example,

```
// IDL
typedef long MyArray[2][3];
// C++
using MyArray = std::array<std::array<std::int32_t, 2>, 3>;
```

The element_traits for `MyArray` is the same as `CORBA::traits<std::int32_t>`.

A.1.1.5 Valuetypes

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

Table A.8: CORBA::traits<> additional member types for Valuetypes

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.
`factory_type`	The `CORBA::traits<T>::ref_type` of the valuetype.
`obv_type`	The C++ `CORBA::traits<>::obv_type` trait is provided for referring to the OBV class that provides default implementations for the accessors and modifiers of the abstract base class.

A.1.1.6 Valueboxes

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

Table A.9: CORBA::traits<> additional member types for Valueboxes

Member	Definition
`boxed_traits`	The `CORBA::traits<>` for the boxed type of the valuebox.

A.1.1.7 Maps

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

Table A.10: CORBA::traits<> additional member types for Unbounded Maps

Member	Definition
`key_traits`	The `CORBA::traits<>` for the key type.
`value_traits`	The `CORBA::traits<>` for the value type
`is_bounded`	`std::false_type` type indicating that the map is unbounded.

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

Table A.11: CORBA::traits<> additional members for Bounded Maps

Member	Definition
`key_traits`	The `CORBA::traits<>` for the key type.
`value_traits`	The `CORBA::traits<>` for the value type
`is_bounded`	`std::true_type` type indicating that the map is bounded.
`bound`	`std::integral_constant` type of value type `uint32_t` indicating the bound of the map.

A.1.1.8 Bitmasks

For a bitmask type, the following additional members shall be available as part of its type trait. Because the bitmask type itself is just an alias for a built-in type, the traits template is specialized on the `<Bitmask>Bits` type.

Table A.12: CORBA::traits<> additional member types for Bitmasks

Member	Definition
`bit_bound`	`std::integral_constant` type of value type `uint32_t` indicating the bit bound of the bitmask.
`underlying_type`	The type mapped as the underlying type of the bitmask.

A.1.1.9 Parameters

The mapping of parameter passing modes is focused at simplicity and ease of use.

For IDL interfaces, valuetypes, CORBA::TypeCode, enums, and basic types, the arguments are:

Table A.13: Parameter passing modes for Interfaces, Valuetypes, CORBA::TypeCode, Enums, and Basic Types

Mode	Parameter Type
`in`	`CORBA::traits<T>::ref_type` `omg::types::ref_type<T>`
`out`	`CORBA::traits<T>::ref_type&` `omg::types::ref_type<T>&`
`inout`	`CORBA::traits<T>::ref_type&`

	`omg::types::ref_type<T>&`
`return`	`CORBA::traits<T>::ref_type` `omg::types::ref_type<T>`

For structured types (e.g. struct, union), sequences, and strings, the arguments are:

Table A.14: Parameter passing modes for Structured Types, Sequences, and Strings

Mode	Parameter Type
`in`	`const T&`
`out`	`T&`
`inout`	`T&`
`return`	`T`

A.1.1.10 Helper Methods

There are two static helper methods provided in the `CORBA::traits<T> struct`. These methods are:

- `CORBA::traits<T>::narrow(CORBA::traits<CORBA::Object>::ref_type p)`
- `CORBA::make_reference<T>(Args&&... args)`

The object traits for type `T` define the method `CORBA::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 to an object of any interface type (`CORBA::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 shall return a valid object reference; otherwise, the operation shall return a nil object reference.

A reference can only be created from a `nullptr`, another reference, or using the `CORBA::make_reference<>(...)` template which shall deliver `std::make_shared` semantics. Any other creation of a reference type is not allowed and shall be considered an ill-formed application. Declaring a reference and initializing it with its default constructor shall result in a nil reference.

A.1.1.11 Servant References

The following traits are provided for servants in the `CORBA::servant_traits<>` template `struct`.

Table A.15: CORBA::servant_traits<> member types

Member	Definition
`base_type`	`PortableServer::Servant`
`ref_type`	`CORBA::traits<PortableServer::Servant>::ref_type`
`weak_ref_type`	`CORBA::traits<PortableServer::Servant>::weak_ref_type`

For instance, given the following IDL:

```
interface foo {...};
```

The generated C++ code would provide:

```
namespace CORBA {
```

```
template<>
struct servant_traits<foo> {
    using base_types = POA_foo;
    using ref_type = CORBA::traits< POA_foo>::ref_type;
    using weak_ref_type = CORBA::traits< POA_foo>::weak_ref_type;
};

} // namespace CORBA
```

A.1.2 Exceptions

An IDL exception shall be mapped to a C++ class that derives from the standard **`CORBA::UserException`**. All exception members shall 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 shall initialize all members to their default values as described in 7.2.4.3.1.

```
namespace CORBA {

class Exception : public std::exception {
public:
    ~Exception() override;
    virtual void raise() const = 0;
    virtual const char* _name() const;
    virtual const char* _rep_id() const;
    const char* what() const noexcept override;
protected:
    Exception();
    Exception(const Exception &);
    Exception(Exception &&);
    Exception& operator=(const Exception&);
    Exception& operator=(Exception&&);
};

} // namespace CORBA
```

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:

```
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 shall 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 the base **`Exception`** class.

For example, the following IDL:

```
exception AnException {
    long error_code;
};
```

would map to the following C++ for CORBA:

```
class AnException : public CORBA::UserException {
public:
    AnException() {...}
    AnException(const AnException&) {...}
    AnException(AnException&&) {...}
    AnException(int32_t error_code, const char* what) {...}
    ~AnException() override {...}
    AnException& operator=(const AnException&) {...}
    AnException& operator=(AnException&&) {...}

    void error_code(int32_t value) {...}
    int32_t error_code() const {…}
};
```

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.

```
namespace CORBA {

enum class CompletionStatus : uint32_t {
    COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
};

class SystemException : public Exception {
public:
    ~SystemException() override;
    uint32_t minor() const;
    void minor(uint32_t);
    virtual void raise() const = 0;
    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&&);
};

} // namespace CORBA
```

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

Each specific system exception is derived from SystemException:

```
namespace CORBA {

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

} // namespace CORBA
```

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

```
try {
    // ...
} catch (const CORBA::Exception& e) {
    //  ...
}
```

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

```
try {
    // ...
} catch (const CORBA::UserException& ue) {
    // ..
} catch (const CORBA::SystemException& se) {
    ...
}
```

Naturally, more specific types can also appear in catch clauses. Also the exceptions can be caught as **std::exception**.

A.1.2.1 UnknownUserException

Request invocations made through the Dynamic Invocation Interface (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:

```
namespace CORBA {

class UnknownUserException final : public UserException {
public:
    const Any& exception() const;
};

} // namespace CORBA
```

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 shall never explicitly throw exceptions of type **UnknownUserException**—it is intended for use with the DII.

A.1.2.2 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 shall support **Any** insertion (but not extraction) for **Exception**. This is required to allow Dynamic Skeleton Interface (DSI)-based applications to catch exceptions as **Exception&** and store them into a **ServerRequest**:

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

Note that this shall result in both the **TypeCode** and the value for the actual derived exception type being stored into the **Any**.

The following **Any** insertion for **Exception** shall be provided:

```
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:

```
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, the operator assigns the actual exception to **se**. If the implementation does not have static type information for the system exception, the operator returns true and assigns an instance of **UNKNOWN** to **se**. Otherwise, the operator returns **false** and the value of se is unchanged.

A.1.2.3 Union Field Method Exceptions

Clause 7.2.4.3.2 states that "Accessing an invalid union member may result in an undefined error." The CORBA mapping for an IDL union, shall throw a **CORBA::BAD_PARAM** system exception when an invalid field is accessed whose discriminant does not match that already set in the union object.

A.1.2.4 Bounded String, Wstring, Sequence, and Map Exceptions

If an IDL **string**, **wstring**, **sequence**, or **map** is defined with a bound, the mapped C++ entity shall throw a **CORBA::BAD_PARAM** system exception when the application detects that specified bounds have been exceeded.

A.1.3 TypeCode

A **TypeCode** is a pseudo object and represents a local IDL interface but is not strictly mapped as an IDL local interface. A CORBA **TypeCode** represents type information. **TypeCode** objects are implemented as a reference type as described in Clause 7.4. For the strong reference, the type **omg::types::ref_type<Y>** shall be used.

No public constructors for **TypeCodes** are defined. However, in addition to the mapped interface, for each basic and defined IDL type, an implementation provides access to a **TypeCode** reference (**omg::types::ref_type<CORBA::TypeCode>**) 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 shall be defined at the same scoping level as its matching type.

For example, for the following IDL structure:

```
struct S {...};
```

The C++ TypeCode reference would be:

```
omg::types::ref_type<CORBA::TypeCode> _tc_S;
```

The IDL **TypeCode** type shall map to a C++ class named **CORBA::TypeCode** according to the following definition:

```
namespace CORBA {

class TypeCode {
public:
    class Bounds final : public CORBA::UserException { ... };
```

```
    class BadKind final : public CORBA::UserException { ... };

    bool equal(omg::types::ref_type<CORBA::TypeCode>) const;
    bool equivalent(omg::types::ref_type<CORBA::TypeCode>) const;
    TCKind kind() const;

    omg::types::ref_type<CORBA::TypeCode> 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;

    omg::types::ref_type<CORBA::TypeCode> member_type(uint32_t index) const;

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

    uint32_t length() const;

    omg::types::ref_type<CORBA::TypeCode> content_type() const;

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

    Visibility member_visibility(uint32_t index) const;
    ValueModifier type_modifier() const;
    omg::types::ref_type<CORBA::TypeCode> concrete_base_type() const;
};

} // namespace CORBA
```

Except **Any** (which is defined Clause A.1.7) and **TypeCode**, all types used in the declaration of **TypeCode** shall be derived from their IDL definition in [OMG-CORBA-IFC] following the mapping rules defined in Chapter 7. The resulting C++ definitions shall be placed in the **CORBA** namespace.

A.1.4 ORB

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

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 [OMG-CORBA-IFC] (see ORB Interface clause, ORB Initialization sub clause).

```
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:

```
namespace CORBA {

using ORBid = omg::types::string;
omg::types::ref_type<ORB> ORB_init(
          int& argc,
          char** argv,
          const omg::types::string& orb_identifier = omg::types::string ());
```

```
} // namespace CORBA
```

The C++ mapping for **ORB_init** deviates from the regular C++ mapping in its handling of the **arg_list** parameter. This is intended to provide a meaningful 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 shall 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.

A.1.5 Object

The CORBA **Object** interface as defined in Clause 8.3, Object Reference Operations. of [OMG-CORBA-IFC] shall be mapped to C++ according to the mapping rules for Interfaces – Full defined in Clause 7.5. The resulting Object class shall be placed in the **CORBA** namespace. In addition to these rules, all operation names in interface Object have leading underscores in the corresponding C++ class.

A.1.5.1 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** shall be supported by a compliant implementation:

- B to A.
- B to Object.

For example:

```
omg::types::ref_type<B> bp = ...;
omg::types::ref_type<A> ap = bp;          // implicit widening
omg::types::ref_type<Object> objp = bp; // implicit widening
objp = ap;                                // implicit widening
```

A.1.5.2 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 **omg::types::ref_type<Object>**.

A.1.5.3 Nil Object Reference

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

```
bool true_result = (A == nullptr);
```

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

A.1.5.4 Narrowing Object Reference

The object traits for type **T** define the method **CORBA::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 to an object of any interface type (**omg::types::ref_type<Object>**). If the actual (runtime) type of the parameter object can be narrowed to the requested interface's type, then the operation shall return a valid object reference; otherwise, the operation shall 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:

- **CORBA::traits<A>::narrow(ap)** returns a valid object reference
- **CORBA::traits<B>::narrow(ap)** returns a valid object reference
- **CORBA::traits<C>::narrow(ap)** returns a valid object reference
- **CORBA::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 **CORBA::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 **CORBA::traits<B>::narrow** or **CORBA::traits<C>::narrow**, a new object reference shall be returned. A call to **CORBA::traits<D>::narrow** shall return a nil reference.

If successful, the narrow method creates a new object reference and does not change the given object reference.

NOTE—The narrow operations can throw system exceptions.

A.1.6 LocalObject

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 shall be (indirectly) derived from **LocalObject** and shall be available as **CORBA::traits<>::base_type**. An object reference referring to a local object shall be created using the **CORBA::make_reference<T>** factory method.

Here is an example of how to implement the following local interface:

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

In C++:

```
class LocalIF : public virtual CORBA::Object {
public:
    MyLocalIF ();
    ~MyLocalIF() override;
    void an_op(int32_t an_arg) override {...}
};

omg::types::ref_type<LocalIF> myref = CORBA::make_reference<LocalIF>();
```

A.1.7 Any

The IDL type **any** maps to a C++ class named **CORBA::Any**. A C++ mapping for the any type shall fulfill to 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.

A.1.7.1 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 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 namespace pollution. The nature of these functions, which are described in detail below, is that the appropriate **TypeCode** (see Clause A.1.3) 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 IDL specifications to be distinct.

A.1.7.2 Insertion into an Any

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

```
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 (**omg::types::ref_type<T>**).
- Valuetype reference (**omg::types::ref_type<T>**).
- Typecode reference (**omg::types::ref_type<CORBA::TypeCode>**).
- Abstract base reference (**omg::types::ref_type<T>**).

For values of type **T** that are too large to be passed efficiently, such as an array, **string**s **wstring**, **struct**, **union**, **sequence**, **any**, and **exception**, the following functions are provided:

```
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.

```
int32_t value = 42;
Any a;
a <<= value;
```

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

A.1.7.3 Extraction from an Any

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

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

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

```
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** shall be able to determine whether the **Any** truly contains 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**:

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

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

```
CORBA::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 shall contain the value that was stored by the **Any**, and **operator>>=** shall return **true**. If the extraction is not successful, the **operator>>=** returns **false**.

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

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

A.1.7.4 TypeCode Replacement

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

```
omg::types::ref_type<CORBA::TypeCode> type() const;
```

Because C++ aliases 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;
omg::types::ref_type<CORBA::TypeCode> 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** shall be preserved as a **tk_alias TypeCode**, applications can use the type modifier function on the **Any** to replace its **TypeCode** with an equivalent one.

```
void type(omg::types::ref_type<CORBA::TypeCode>);
```

Revising the previous example:

```
// C++
Any any;
LongType val = 1234;
any <<= val;
any.type(_tc_LongType); // replace TypeCode
omg::types::ref_type<CORBA::TypeCode> 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** are not equivalent, the type modifier function raises the **BAD_TYPECODE** exception.

A.1.8 Value Types

An IDL **valuetype** shall be mapped to the C++ trait **CORBA::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++ **CORBA::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 **CORBA::traits<>::obv_type** trait.

In C++, **valuetype**s map to valuetype references that behave as reference type as described in Clause 7.4. The reference type **omg::types::ref_type<T>** is available for each **valuetype**. The strong reference is delivered as **omg::types::ref_type<T>** and the weak reference as **omg::types::weak_ref_type<T>**.

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 **CORBA::traits<>::factory_type** trait.

A.1.8.1 Valuetype Data Members

The C++ mapping for valuetype data members follows the same rules as the C++ mapping for structured types using public accessors and modifiers described in Clause 8.1.1, 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 shall be declared **private**.

For example, the following 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;
};
```

Would map to C++ as follows:

```
using Bytes = std::array<uint8_t, 64>;
struct S { ...  };

class Val : public virtual CORBA::ValueBase {
public:
// ...
    virtual omg::types::ref_type <Val> t() const = 0;
    virtual omg::types::ref_type <Val>& t() = 0;
    virtual void t(omg::types::ref_type <Val>) = 0;
    virtual const Bytes& w() const = 0;
    virtual Bytes& w() = 0;
    virtual void w(const Bytes&) = 0;
    virtual void w(Bytes&&) = 0;

    virtual const omg::types::string& x() const = 0;
    virtual omg::types::string& x() = 0;
    virtual void x(const omg::types::string&) = 0;
    virtual void x(omg::types::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 omg::types::ref_type<A> z() const = 0;
    virtual omg::types::ref_type<A>& z() = 0;
    virtual void z(omg::types::ref_type<A>) = 0;
    // ...
};
```

These rules for the accessors correspond directly to the parameter passing rules for structured types as explained in Clause A.1.1.9.

A.1.8.2 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 data types to their default value as defined in Clause 7.2.4.1, 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.

A.1.8.3 Valuetype Operations

Operations declared on a **`valuetype`** are mapped to public pure virtual member functions in the corresponding **`valuetype`** C++ class. (Note that state **`valuetype`** 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 Clause A.1.9.2 for client side interface operations.

As part of the **`valuetype`** traits **`CORBA::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.

A.1.8.4 Valuetype Example

For example, consider the following IDL valuetype:

```
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:

```
class Example : public virtual ValueBase {
public:
    virtual int16_t op1() = 0;
    virtual int32_t op2(omg::types::ref_type<Example>) = 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&&);
    ~Example() override;
```

```
    virtual int16_t val1() const = 0;
    virtual int16_t& val1() = 0;
    virtual void val1(int16_t) = 0;

    virtual const omg::types::string& val3() const = 0;
    virtual omg::types::string& val3() = 0;
    virtual void val3(const omg::types::string&) = 0;
    virtual void val3(omg::types::string&&) = 0;

    virtual omg::types::ref_type<Example> val5() const = 0;
    virtual omg::types::ref_type<Example>& val5() = 0;
    virtual void val5(omg::types::ref_type<Example>) = 0;
private:
    Example& operator=(const Example&) = delete;
    Example& operator=(Example&&) = delete;
};

class OBV_Example : public virtual Example {
public:
    void val2 (int32_t) override;
    int32_t val2 () const override;
    int32_t& val2 () override;
protected:
    OBV_Example();
    OBV_Example (const OBV_Example&);
    OBV_Example (OBV_Example&&);
    OBV_Example(
            int16_t,
            int32_t,
            omg::types::string,
            omg::types::ref_type<Example>);
    ~OBV_Example() override;

    int16_t val1() const override;
    int16_t& val1() override;
    void val1(int16_t) override;

    const omg::types::string& val3() const override;
    omg::types::string& val3() override;
    void val3(const omg::types::string&) override;
    void val3(omg::types::string&&) override;

    omg::types::ref_type<Example> val5() const override;
    omg::types::ref_type<Example>& val5() override;
    void val5(omg::types::ref_type<Example>) override;
    // ...
};
```

A.1.8.5 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:

```
class ValueBase {
public:
    virtual omg::types::ref_type<ValueBase> _copy_value();
protected:
    ValueBase();
    ValueBase (ValueBase&&);
    ValueBase(const ValueBase&);
    ~ValueBase() override;
```

```
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**.

A.1.8.6 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 factory has to be provided by the user.

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++
omg::types::ref_type<StringValue> 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:

```
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.

Value box classes follow the rules of structured types using public accessors and modifiers described in Clause 8.1.1,. Additionally to these rules value boxes are final and have:

- Accessors are always named **_value**.
- A protected destructor and protected constructors.
- 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;
    ~ColorValue() override;
};
```

A.1.8.7 Abstract Valuetypes

Abstract IDL valuetypes follow the same C++ mapping rules as concrete IDL valuetypes, except that because they have no data members, the IDL compiler does not generate the OBV traits for them.

A.1.8.8 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 Clause A.1.5.1).
- 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 CORBA::traits<B>::base_type {
public:
    void op() override;
    // ...
};
```

A.1.8.9 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.

A.1.8.9.1 ValueFactoryBase Class

Just as they provide concrete C++ **valuetype** classes, applications shall 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:

```
class ValueFactoryBase {
protected:
    ~ValueFactoryBase() override;
    ValueFactoryBase();
private:
    virtual omg::types::ref_type<ValueBase> 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 unmarshalling 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 Clause A.1.8.9.2) 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 unmarshalling, 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 unmarshalled 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 Clause A.1.8.9.2).

A.1.8.9.2 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 **CORBA::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**:

```
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 shall be generated as follows:

```
class V_factory : public ... {
public:
    virtual omg::types::ref_type<V> create_bool(bool val) = 0;
    virtual omg::types::ref_type<V> create_char(char val) = 0;
    virtual omg::types::ref_type<V> create_octet(uint8_t val) = 0;
    virtual omg::types::ref_type<V> create_other(uint16_t s, const std::string& p) = 0;
protected:
    ~V_factory() override;
    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. Each also supplies a public pure virtual function corresponding to each initializer. Applications derive concrete factory classes from the **CORBA::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 **`valuetype`**s that have no operations or initializers, a concrete type-specific factory class is generated whose implementation of the **`create_for_unmarshal`** function simply constructs an instance of the **`CORBA::traits<>::obv_type`** trait class for the **`valuetype`** using **`CORBA::traits<>::make_reference`**.

For **`valuetype`**s that have operations, but no initializers, there are no type-specific abstract factory classes, but applications shall still supply concrete factory classes. These classes, which are derived directly from **`CORBA::traits<>::factory_type`** only need to override the **`create_for_unmarshal function`**.

A.1.8.9.3 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 shall 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 Dynamic Invocation Interface (DII) or Dynamic Skeleton Interface (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 shall be registered with the implementation and available for lookup if the application attempts extraction of the **`valuetype`**s via the statically-typed Any extraction functions. See Clause A.1.7.3 for more details.

A.1.8.10 Custom Marshaling

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

A.1.9 Abstract Interfaces

The C++ mapping for abstract interfaces is almost identical to the mapping for Interfaces – Full. Rather than defining a complete C++ mapping for abstract interfaces, which would only duplicate much of the specification of the mapping for Interfaces – Full defined in Clause 7.5, only the ways in which the abstract interface mapping differs from the regular interface mapping are described here.

A.1.9.1 Abstract Interface Base

For abstract interfaces the **`CORBA::traits<>`** trait shall be provided. This trait delivers a strong reference type as **`omg::types::ref_type<>`** and a weak reference type as **`omg::types::ref_type<>::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 C++ **AbstractBase** class is shown below:

```
class AbstractBase {
public:
    virtual omg::types::ref_type<Object> _to_object();
    virtual omg::types::ref_type<ValueBase> _to_value();
protected:
    AbstractBase();
    AbstractBase(const AbstractBase&);
    AbstractBase(AbstractBase&&);
    AbstractBase& operator=(const AbstractBase&);
    AbstractBase& operator=(AbstractBase&&);
    ~AbstractBase() override;
};
```

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.

A.1.9.2 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.

A.1.10 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 IDL interface.

A.1.10.1 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 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 shall 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.

A.1.10.2 PortableServer::Servant

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

```
namespace PortableServer {
class Servant {
public:
    virtual omg::types::ref_type<PortableServer::POA> _default_POA();
    virtual omg::types::ref_type<CORBA::InterfaceDef> _get_interface();
    virtual bool _is_a(const omg::types::string& logical_type_id);
    virtual bool _non_existent();
protected:
    ~Servant() override;
    Servant();
    Servant(const Servant &);
    Servant(Servant &&);
    Servant& operator=(const Servant &);
    Servant& operator=(Servant &&);
};

} // namespace PortableServer
```

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, shall 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) shall be provided so that derived servants can be constructed portably. Both a copy constructor and a protected default assignment operator shall 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.

A.1.10.3 Servant References

Given an interface **Foo** the mapping shall provide a **CORBA::servant_traits<Foo>** trait. The strong reference type is provided as **CORBA::servant_traits<Foo>::ref_type** trait (also available as **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 (i.e., **CORBA::weak_servant_reference<>**) shall be provided. These servant reference types behave as reference types.

This trait together with the **CORBA::make_reference<>** factory method shall be used to write exception-safe and type-safe code for heap-allocated servants (a C++ 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**:

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

A.1.10.4 Servant Argument Passing

The POA shall maintain servants as servant references with the semantics as described in Clause 7.4. For each POA the **ServantActivator** and **ServantLocator** provide operations that either pass a **Servant** as a parameter or returns a **Servant** a **omg::types::ref_type<PortableServer::Servant>** shall be passed.

A.1.10.5 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 shall 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:

```
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:

```
class A_skel : public virtual ... {
public:
    omg::types::ref_type<A> _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:

```
omg::types::ref_type<A_impl> my_a = CORBA::make_reference<A_impl>();
omg::types::ref_type<A> a = my_a->_this();
```

A.1.10.6 Inheritance-Based Interface Implementation

Implementation shall be derived from a generated base class based on the 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:

```
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 IDL interface. It has a form similar to the following:

```
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 shall 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 shall be a virtual base class of the type related to the trait to allow portable implementations to inherit from both skeleton classes and implementation classes for other base interfaces without error or ambiguity.

The **PortableServer::Servant** shall 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:

```
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 ();
    ~A_skel() override;
};
```

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 shall derive from this trait and implement each of the operations in the IDL interface. An implementation class declaration for interface **A** would take the form:

```
class A_impl : public virtual CORBA::servant_traits<A>::base_type {
public:
    int16_t op1() override;
    void op2(const int32_t val) override;
    ...
protected:
    ~A_impl() override;
};
```

Note that the presence of the _this() function implies that C++ servants shall 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 shall 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 Clause A.1.11.

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:

```
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 { ... };
```

A.1.10.7 Implementing Operations

The signature of an implementation member function is the mapped signature of the 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:
    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:
    void f() override;
    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<>`**.

A.1.10.8 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 shall 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:

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

In addition to generating 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`**.

```
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 shall 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 shall 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,
            omg::types::ref_type<PortableServer::POA> poa = {}) :
                tied_object_(std::move(t)), poa_(std::move(poa))
    {
    }

    ~TIE() override = 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 Servant operations
    omg::types::ref_type<PortableServer::POA> _default_POA() override
    {
        if (poa_) {
            return poa_;
        }
        else {
            // return root POA
        }
    }
private:
    omg::types::ref_type<PortableServer::POA> 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 **`Servant::_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:

```
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 shall 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:

```
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 shall not access the **`TIE`** class type and data members directly, since the names of those data members are not standardized.

A.1.11 Mapping DSI to C++

Clause 12.3 of [OMG-CORBA-IFC] contains general information about mapping the Dynamic Skeleton Interface (DSI) to programming languages, including:

- Dynamic Skeleton Interface's ServerRequest Operations
- Portable Object Adapter's Dynamic Implementation Routine

A.1.11.1 Mapping of ServerRequest

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

A.1.11.2 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 DSI is implemented through servants that are members of classes that inherit from dynamic skeleton classes.

```
namespace PortableServer {

class DynamicImplementation : public virtual Servant {
public:
    omg::types::ref_type<Object> _this();
    virtual void invoke(omg::types::ref_type<ServerRequest> request) = 0;
    virtual RepositoryId _primary_interface(
        const ObjectId& oid,
        omg::types::ref_type<POA> poa) = 0;
    };

} // namespace PortableServer
```

The **_this()** function returns an **omg::types::ref_type<Object>** 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 shall 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.

A.1.12 PortableServer Functions

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

```
namespace PortableServer {

omg::types::string ObjectId_to_string(const ObjectId&);
omg::types::wstring ObjectId_to_wstring(const ObjectId&);
ObjectId string_to_ObjectId(const omg::types::string&);
ObjectId wstring_to_ObjectId(const omg::types::wstring&);

} // namespace PortableServer
```

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.

A.1.13 Mapping for PortableServer::ServantManager

A.1.13.1 Mapping for Cookie

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

```
namespace PortableServer {

class ServantLocator {
    // ...
    using Cookie = void*;
};

} // namespace PortableServer
```

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**.

A.1.13.2 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.

A.1.13.3 Server Side Mapping for Abstract Interfaces

The only circumstances under which an IDL compiler shall 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.

A.2 DDS-Specific Mappings

DDS requires no additional platform-specific language mappings. Implementations of this specification targeting DDS shall therefore be based solely on the IDL to C++ mappings defined in Chapters 7 and 8 for the building blocks that compose the DDS profiles defined in Clause 9.3 of [OMG-IDL4].

Annex B: Building Block Traceability Matrix

(non-normative)

The building block traceability matrix in Table B.1 provides an indication of which clause within this specification addresses each IDL building block.

Table B.1: Building Block Traceability Matrix

Building Block	Section(s)
Core DataTypes	7.2 Core Data Types
Any	7.3 Any
Interfaces – Basic	7.4 Interfaces – Basic
Interfaces – Full	7.5 Interfaces – Full
Value Types	7.6 Value Types
CORBA-Specific – Interfaces	7.7 CORBA-Specific – Interfaces
CORBA-Specific – Value Types	7.8 CORBA-Specific – Value Types
Components – Basic	7.9 Components – Basic
Components – Homes	7.10 Components – Homes
CCM-Specific	7.11 CCM-Specific
Components – Ports and Connectors	7.12 Components – Ports and Connectors
Template Modules	7.13 Template Modules
Extended Data Types	7.14 Extended Data Types
Anonymous Types	7.15 Anonymous Types
Annotations	7.16 User-Defined Annotations

This page intentionally left blank.

Annex C: Compatibility Rules for C++98 and C++03

(normative)

C.1 Overview

The language mappings defined in this specification assume the target compiler supports C++11 [ISO/IEC-14882:2011] or above. Thus, some mapping rules generate code that is incompatible with older versions of the C++ standard, such as C++98 [ISO/IEC-14882:1998] or C++03 [ISO/IEC-14882:2003].

Implementers of this specification that require support for older C++ standard versions shall follow the rules specified in this specification, except for the specific constructs listed below, which provide mapping rules compliant with C++98 and C++03.

C.2 IDL to C++ Language Mapping

C.2.1 Core Data Types

Core data types shall be mapped according to the rules specified in Clause 7.2. The following clauses define the mapping rules that need to be altered to support C++98 and C++03.

C.2.1.1 Constants

IDL constants of shall be mapped to C++ constant declarations of equivalent type with the same name and value within the equivalent scope and namespace where they are defined.

For example, the IDL const declarations below:

```
module my_math {
    const string my_string = "My String Value";
    const double PI = 3.141592;
};
```

would map to the following C++:

```
namespace my_math {
    const std::string my_string = "My String Value";
    const double PI = 3.141592;
}
```

The constant value of wide character and wide string constants shall be preceded by `L` in C++.

For example, IDL constant:

```
const wstring ws = "Hello World";
```

would map to the following C++:

```
const std::wstring ws = L"Hello World";
```

C.2.1.2 Data Types

C.2.1.2.1 Basic Types

C.2.1.2.1.1 Integer Types

Integer types shall be mapped to the types shown in Table C.1. These types shall be made available to users ensuring that the underlying C++ data type has the appropriate size in the target platform.

Table C.1: Mapping of Integer Types

IDL Type	C++ Type	Default Value
`short`	`omg::types::int16_t`	`0`
`unsigned short`	`omg::types::uint16_t`	`0`
`long`	`omg::types::int32_t`	`0`
`unsigned long`	`omg::types::uint32_t`	`0`
`long long`	`omg::types::int64_t`	`0`
`unsigned long long`	`omg::types::uint64_t`	`0`

C.2.1.2.2 Constructed Types

In general, an IDL constructed type shall be mapped to a C++ class with the same name following the mapping rules specified in Clause 7.2.4.3. However, the mapped classes shall not provide a move constructor or an assignment move operator as these are unsupported in C++98 and C++03.

The other difference is the mapping rules for Enumerations, which are specified below.

C.2.1.2.2.1 Enumerations

An IDL **`enum`** shall be mapped to a C++ **`enum`** with the same name as the IDL **`enum`** type.

For example, the IDL **`enum`** declaration below:

```
enum AnEnum {
    zero,
    one,
    two
};
```

would map to the following C++:

```
enum AnEnum {
    zero,
    one,
    two
};
```

C.2.1.2.3 Arrays

IDL arrays shall map to C-style arrays of the mapped element type. Likewise, multidimensional arrays shall be mapped to C-style multidimensional arrays of the mapped element type.

For example the IDL declaration below:

```
typedef long long_array[100];
typedef string string_array[1][2];
```

would map to the following C++:

```
typedef omg::types::int32_t long_array[100];
typedef std::string string_array[1][2];
```

C.2.1.2.4 Naming Data Types

IDL **`typedef`**s shall be mapped to C++ **`typedef`** declarations.

For example the IDL declaration below:

```
typedef long Length;

struct MyType {
    Length my_type_length;
};
```

would map to the following C++:

```
typedef omg::types::int32_t Length;


struct MyType {
    Length my_type_length;
};
```

C.2.2 Interfaces – Basic

Each IDL interface shall be mapped to a C++ class following the mapping rules specified in Clause 7.4, taking into account that parameters of interface type **`T`** shall be mapped to **`T*`**, instead of **`std::shared_ptr<T>`**.

C.2.3 Interfaces – Full

Interfaces – Full shall follow the mapping rules specified in Clause 7.4, taking into account the rules for mapping parameters of interface type defined in Clause C.2.2.

C.2.4 Extended Data Types

C.2.4.1 Additional Template Types

Additional template types shall be mapped as defined in Clause 7.14.3 with the exception of the **`bitmask`** type, which shall be defined as specified below.

C.2.4.1.1 Bitmask Type

IDL **`bitmask`** declarations shall be mapped to a C++ **`struct`** type containing the following elements:

- An **enum** named **_flags** that includes an enumerator for each of the values defined in the scope of the IDL **bitmask**, with each enumerator explicitly initialized to its corresponding integer value.
- A private member named **_value** of an unsigned integer type that can be safely cast to any value of **_flags**; that is, **omg::types::uint8_t** for values of **bit_bound** between 1 and 8; **omg::types::uint16**_t for values of **bit_bound** between 9 and 16, **omg::types::uint32_t** for bit_**bound values** between 17 and 32, and **omg::types::uint64_t** for bit_bound values between 33 and 64.
- A default constructor and a copy constructor.
- An implementation of the **!=**, **&=**, and **^=** bitwise operators.
- An implementation of the function call operator that returns **_value**.

For example, the IDL bitmask declaration below:

```
@bit_bound(32)
bitmask MyBitMask {
    @position(0) flag0,
    @position(1) flag1

};
```

would map to the following C++:

```
struct MyBitMask {
    enum MyBitMaskBits {
        flag0 = 0x01 << 0,
        flag1 = 0x01 << 1

    };

    MyBitMask() : _value(0U) {}
    MyBitMask(omg::types::uint32_t v): _value(v) {}

    operator omg::types::uint32_t()
    {
        return _value;
    }

    MyBitMask& operator|=(omg::types::uint32_t other)
    {
        _value != other;
        return *this;
    }

    MyBitMask& operator&=(omg::types::uint32_t other)
    {
        _value &= other;
        return *this;
    }

    MyBitMask& operator^=(omg::types::uint32_t other)
    {
        _value ^= other;
        return *this;
    }

private:
    omg::types::uint32_t _value;
};
```

NOTE—In C++98 and C++03, the underlying type of an enumeration cannot be fixed. Therefore in the mapping defined above, the underlying type of `_flags` will be “an integral type that can represent all enumeration values defined in the enumeration.” The standard also states that “it is implementation-defined which integral type is used as the underlying type except that the underlying type shall not be larger than int unless the value of an enumerator cannot fit in an int or unsigned int.”

C.2.4.2 8-bit Integer Types

8-bit integer types shall be mapped as shown in Table C.2. These types shall be made available to users ensuring that the underlying C++ data type has the appropriate size in the target platform.

Table C.2: Mapping of 8-bit Integer Types

IDL Type	C++ Type	Default Value
`int8`	`omg::types::int8_t`	`0`
`uint8`	`omg::types::uint8_t`	`0`

C.2.4.3 Explicitly-Named Integer Types

Explicitly-named integer types shall be mapped as shown in Table C.3. These types shall be made available to users ensuring that the underlying C++ data type has the appropriate size in the target platform.

Table C.3: Mapping of Explicitly-Named Integer Types

IDL Type	C++ Type	Default Value
`int16`	`omg::types::int16_t`	`0`
`uint16`	`omg::types::uint16_t`	`0`
`int32`	`omg::types::int32_t`	`0`
`uint32`	`omg::types::uint32_t`	`0`
`int64`	`omg::types::int64_t`	`0`
`uint64`	`omg::types::uint64_t`	`0`

C.2.5 Standardized Annotations

Except for the group of annotations listed below, standardized annotations shall be mapped as defined in Clause 7.17 of this specification.

C.2.5.1 Group of Annotations: General Purpose

Table C.4 identifies the mapping impact of the IDL defined General Purpose Annotations.

Table C.4: General Purpose Annotation Impact

General Purpose Annotation	Impact on C++ Language Mapping
`@id`	No impact on language mapping
`@autoid`	No impact on language mapping

`@optional`	IDL member declarations preceded by the **`@optional`** annotation shall be mapped to **`T*`**, where **`T`** is the type of the IDL optional member. In such cases, a **`NULL`** pointer indicates an omitted value.
`@position`	Impacts the mapping of **`bitmask`** types as defined in Clause 7.14.3.3.

`@value`	Impacts the mapping of **`enum`** types, providing the value of the annotated enumerator.
	For example:
	`enum Color {`
	`  @value(1) red,`
	`  @value(2) green,`
	`  @value(3) blue`
	`};`
	would map to the following C++:
	`enum Color {`
	`  red = 1,`
	`  green = 2,`
	`  blue = 3`
	`};`

`@extensibility`	No impact on language mapping
`@final`	No impact on language mapping
`@mutable`	No impact on language mapping
`@appendable`	No impact on language mapping

C.2.5.2 Group of Annotations: Data Implementation

Table C.5 identifies the mapping impact of the IDL defined Data Implementation annotations.

Table C.5: Data Implementation Annotation Impact

Data Implementation Annotation	Impact on C++ Language Mapping
`@bit_bound`	A **`@bit_bound`** annotation preceding an IDL **`enum`** declaration has no impact on the language mapping. The mapping for a IDL **`bitmask`** declaration preceded by the **`@bit_bound`** annotation is described in Clause C.2.4.1.1.
`@external`	IDL member declarations preceded by the **`@external`** annotation shall be mapped to **`T*`**, where **`T`** is the type of the IDL external member.
`@nested`	No impact on the language mapping

C.3 IDL to C++ Language Mapping Annotations

C.3.1 @cpp_mapping Annotation

The compatibility rules for C++98 and C++03 add the following parameters to the list of parameters for the **`@cpp_mapping`** annotation defined in Clause 8.1.

Therefore, the annotation declaration defined in Clause 8.1 shall be augmented with the following members:

```
@annotation cpp_mapping {
    // ...
    string enum_prefix default "";
    string enum_suffix default "";
};
```

C.3.1.1 enum_prefix

`enum_prefix` provides a way to scope enumerator identifiers as they are mapped to C++. When this parameter is present, the mapping shall prepend the identifier of each enumerator with the value of **`enum_prefix`**.

For example, the IDL enum declaration below:

```
@cpp_mapping(enum_prefix="color_")
enum Colors { red, green, blue };
```

would map to the following C++:

```
enum Colors { color_red, color_green, color_blue };
```

C.3.1.2 enum_suffix

`enum_suffix` provides a way to scope enumerator identifiers as they are mapped to C++. When this parameter is present, the mapping shall append the value of **`enum_suffix`** to the identifier of each enumerator.

For example, the IDL **`enum`** declaration below:

```
@cpp_mapping(enum_suffix="_color")
enum Colors { red, green, blue };
```

would map to the following C++:

```
enum Colors { red_color, green_color, blue _color };
```

C.4 Platform-Specific Mappings

C.4.1 CORBA-Specific Mappings

CORBA-specific mappings shall apply the rules defined in Clause A.1 of this specification, with the exceptions listed below.

C.4.1.1 Exceptions

The **`Exception`** abstract shall not provide a move constructor or an assignment move operator, as these are unsupported in C++98 and C++03.

C.4.1.2 TypeCode

The IDL TypeCode type shall mapped to a C++ class named **CORBA::TypeCode** according to the following definition:

```
namespace CORBA {

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

    bool equal(CORBA::TypeCode*) const;
    bool equivalent(CORBA::TypeCode*) const;
    TCKind kind() const;

    CORBA::TypeCode* get_compact_typecode() const;
    const std::string& id() const;
    const std::string& name() const;

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

    CORBA::TypeCode* member_type(omg::types::uint32_t index) const;

    const Any& member_label(omg::types::uint32_t index) const;
    CORBA::TypeCode* discriminator_type() const;
    omg::types::int32_t default_index() const;

    omg::types::uint32_t length() const;

    CORBA::TypeCode* content_type() const;

    omg::types::uint16_t fixed_digits() const;
    omg::types::int16_t fixed_scale() const;

    Visibility member_visibility(omg::types::uint32_t index) const;
    ValueModifier type_modifier() const;
    CORBA::TypeCode* concrete_base_type() const;
};

} // namespace CORBA
```

Except **Any** (which shall be mapped as defined in Clause C.4.1.3) and **TypeCode**, all types used in the declaration of **TypeCode** shall be derived from their IDL definition in [OMG-CORBA-IFC] following the mapping rules defined in Chapter 7 with the exceptions for C++98 and C++03 defined in Annex C. The resulting C++ definitions shall be placed in the **CORBA** namespace.

C.4.1.3 Any

The IDL type shall be mapped according to the rules defined in Clause A.1.7 of this specification, taking into account the following considerations for C++98 and C++03:

- In Clause A.1.7.2, the implementation shall not include a move insert, as it is unsupported in C++98 and C++03.
- In Clause A.1.7.4, the type accessor function shall return a pointer to the **TypeCode** associated with the any, as opposed to a reference to it:
  CORBA::TypeCode* type() const;
  Likewise the type modifier function on the any shall take a pointer to the **TypeCode** as an argument, as

opposed to a reference to it:

```
void type(CORBA::TypeCode*);
```

This page intentionally left blank.

Annex D: IDL4 Mapping Rules for Classic C++ Language Mapping Specifications

(normative)

D.1 Overview

This Annex provides mapping rules for the building blocks introduced in IDL4 that are not addressed in the classic C++ and C++11 Language Mappings (see [OMG-C++] and [OMG-C++11], respectively). These set of rules allow implementers of the classic mappings to extend existing IDL compilers and platforms to incorporate concepts from IDL4, such as extended data types and annotations, using a standard set of mapping rules that are consistent with the requirements and conventions of the original specifications.

D.2 IDL4 Mappings Rules for C++ Language Mapping Specification

The following clauses provide mapping rules for the portions of IDL4 that are not covered by the classic C++ Language Mapping [OMG-C++]. Such mapping rules are consistent with the requirements and mapping style of the original specification.

D.2.1 Extended Data Types

D.2.1.1 Structures with Single Inheritance

An IDL **`struct`** that inherits from a base IDL **`struct`** shall be mapped to a C++ **`struct`**, with each **`struct`** member mapped to a corresponding member of the C++ **`struct`** in the same order. The mapped **`struct`** shall inherit from the mapped base **`struct`** using public inheritance. The resulting C++ **`struct`** shall comply with the requirements set forth for structured types and struct types in Clauses 5.11 and 5.12 of [OMG-C++].

For example, an IDL **`struct`** extending the **`MyStruct`** structure define in Clause 7.2.4.3.1 of this specification:

```
struct ChildStruct : MyStruct {
    float a_float;
};
```

would map to the following C++:

```
struct ChildStruct : public MyStruct {
    Float a_float;
};
```

D.2.1.2 Union Discriminators

This building block defined in [OMG-IDL4] adds the **`wchar`** and **`octet`** IDL types to the set of valid types for a union discriminator. The mapping of union discriminators of such types shall be mapped as specified in Clause 5.14 of [OMG-C++].

Any addition to the list of supported integer, **`char`**, **`boolean`**, or **`enum`** types as a result of the implementation of the Extended Data Types building block makes such types valid union discriminators as well. Therefore, if 8-bit integer values are supported (see Clause D.2.1.4), **`int8`** and **`uint8`** shall be treated as valid union discriminators and shall be mapped as specified in Clause D.2.1.4.

D.2.1.3 Additional Template Types

D.2.1.3.1 Maps

IDL **`map`** declarations shall be mapped to a C++ class. Classes representing bounded and unbounded maps shall implement at a minimum the following constructors and methods:

- A default constructor that sets the map size to zero and initializes the internal representation of the map.
- An **`operator[]`** overload that returns a reference to the value associated with the given key.
- **`ULong maximum() const`**, which returns the maximum size of the map. In other words, the maximum number of entries the map can hold at any given time.
- **`ULong length() const`**, which returns the current size of the map. In other words, the current number of entries the map holds at this time.
- **`void clear()`**, which clears all the entries from the map. The size of the map is set to 0 and the maximum number of entries does not change.
- **`Boolean insert(const K& key, const T& element, Boolean replace = TRUE)`**, which inserts a new entry into the map with the given key and element values.
    - If replace is set to **`FALSE`**, and an element with the given key already exists in the map, the method shall fail and return **`FALSE`**.
    - If the process of adding a new element exceeds the maximum value of allowed elements (i.e., if **`length + 1`** is greater than **`maximum`**), the method shall fail and return **`FALSE`**.
    - Otherwise, the method shall add the new element, increase the length in **`length + 1`**, and return **`TRUE`**.
- **`Boolean insert_or_assign(const K& key, const T& element)`**, which behaves as the **`insert()`** method if the replace parameter is set to **`TRUE`**.
- **`Boolean erase(const K& key)`**, which removes the given key from the map. If successful, it decreases the length in **`length - 1`** and returns **`TRUE`**. Otherwise, it returns **`FALSE`**.

Unbounded maps shall also implement the following methods:

- A copy constructor that creates a new map with the same maximum and length as the given map, and copies every element from the given map.
- An **`operator=`** overload that replaces the contents of the map with a copy of the content of a given map.
- A “maximum constructor” that takes the maximum size of the unbounded map as a parameter, allowing internal preallocations if necessary.
- **`void maximum(ULong)`**, which sets the maximum size of the unbounded map. Reallocation is conceptually equivalent to creating a new map of the desired new length, copying the old map elements zero through **`length - 1`** into the new map, and then assigning the old map to be the same as the new map.

Implementers may add methods to classes representing bounded and unbounded maps to provide additional functionality.

For example, the following :

```
typedef map<unsigned long, T> M1; // unbounded map
typedef map<string, T, 20> M2;    // bounded map
```

would map to C++ as follows:

```
class M1 {
public:
    M1();
    M1(ULong);
    M1(const M1&);
    ~M1();
    M1 &operator=(const M1&);

    T &operator[](ULong key);
    const T &operator[](ULong key) const;

    ULong maximum() const;
    void maximum(ULong max_size);

    ULong length() const;

    void clear();

    Boolean insert(
             ULong key,
             const T& element,
             Boolean replace = true);

    Boolean insert_or_assign(ULong key, const T& element);

    Boolean erase(ULong key);
};

class M2 {
public:
    M2();
    ~M2();

    T &operator[](const std::string& key);
    const T &operator[](const std::string& key) const;

    ULong maximum() const;

    ULong length() const;

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

    void clear();

    Boolean insert(
             const std::string& key,
             const T& element,
             Boolean replace = true);

    Boolean erase(ULong key);
};
```

D.2.1.3.2 Bitsets

IDL **`bitset`** declarations shall be mapped as defined in Clause 7.14.3.2 of this specification.

D.2.1.3.3 Bitmask Type

IDL **`bitmask`** declarations shall be mapped as defined in Clause C.2.4.1.1 of this specification.

D.2.1.4 8-bit Integer Types

8-bit integer types shall be mapped as shown in Table D.1

Table D.1: Mapping of 8-bit Integer Types

IDL Type	C++ Type	C++ Out Type
`int8`	`CORBA::Int8`	`CORBA::Int8_out`
`uint8`	`CORBA::UInt8`	`CORBA::UInt8_out`

Each 8-bit integer type is mapped to a `typedef` in the `CORBA` module. The `typedef` shall guarantee that the underlying C++ type guarantees the size and sign requirements in the target platform (i.e., an 8-bit signed integer for `int8`, and an 8-bit unsigned integer for `uint8`). The rest of considerations that apply to the mapping of basic types defined in Clause 5.7 of [OMG-C++] apply to `int8` and `uint8`.

Implementers of this specification that target a technology different than CORBA may map `Int8` and `UInt8` to `typedef`s in a different module. For example, a `typedef` for `int8` for DDS could be mapped to `DDS::Int8`.

D.2.1.5 Explicitly-Named Integer Types

Explicitly-named integer types shall be mapped as shown in Table D.2.

Table D.2: Mapping of Explicitly-Named Integer Types

IDL Type	C++ Type	C++ Out Type
`int16`	`CORBA::Short`	`CORBA::Short_out`
`uint16`	`CORBA::UShort`	`CORBA::UShort_out`
`int32`	`CORBA::Long`	`CORBA::Long_out`
`uint32`	`CORBA::ULong`	`CORBA::ULong_out`
`int64`	`CORBA::LongLong`	`CORBA::LongLong_out`
`uint64`	`CORBA::ULongLong`	`CORBA::ULongLong_out`

Implementers of this specification that target a technology different than CORBA may map explicitly-named integer types to `typedef`s in a different module. For example, a `typedef` for `int16` for DDS could be mapped to `DDS:Short`.

D.2.2 User-Defined Annotations

User-defined annotations are not propagated to the generated C++ code.

D.2.3 Standardized Annotations

D.2.3.1 Group of Annotations: General Purpose

Table D.3 identifies the mapping impact of the IDL defined General Purpose Annotations.

Table D.3: General Purpose Annotation Impact

General Purpose Annotation	Impact on C++ Language Mapping
`@id`	No impact on language mapping
`@autoid`	No impact on language mapping

`@optional`	Each IDL `struct` member annotated with `@optional` shall be represented using plain pointers.
	• In cases where the mapping of non-optional members already uses a plain pointer, it shall remain unchanged.
	• In cases where the mapping of non-optional members uses a `_var` smart pointer, the `_var` type shall be replaced by the corresponding plain pointer type. For example, `MyType_var` is replaced by `MyType*`.
	• In cases where the mapping of non-optional members uses an automatic member of type `T`, `T` shall be replaced by pointer-to-`T`. For example, `Short` shall be replaced by `Short*`.
	A `NULL` pointer indicates an omitted value.
	For example:
	`struct S {`
	`  string name;`
	`  @optional float age;`
	`};`
	would map to the following C++:
	`struct S {`
	`  char* name;`
	`  Float* age;`
	`};`

`@position`	Impacts the mapping of `bitmask` types as defined in Clause D.3.1.3.3 of this specification.

`@value`	Impacts the mapping of `enum` types, providing the value of the annotated enumerator.
	For example:
	`enum Color {`
	`  @value(1) red,`
	`  @value(2) green,`
	`  @value(3) blue`
	`};`
	would map to the following C++:
	`enum Color {`
	`  red = 1,`
	`  green = 2,`
	`  blue = 3`
	`};`

`@extensibility`	No impact on language mapping
`@final`	No impact on language mapping

@mutable	No impact on language mapping
@appendable	No impact on language mapping

D.2.3.2 Group of Annotations: Data Modeling

IDL defined Data Modeling annotations shall be mapped as defined in Clause 7.17.2 of this specification.

D.2.3.3 Group of Annotations: Units and Ranges

Table D.4 identifies the mapping impact of the IDL defined Units and Ranges annotations.

Table D.4: Units and Ranges Annotation Impact

Units and Ranges Annotation	Impact on C++ Language Mapping
@default	C++ elements declared as result of the mappings defined in this specification containing a **@default** annotation shall be initialized to the value of the annotation.
@range	No impact on language mapping.
@min	No impact on language mapping.
@max	No impact on language mapping.
@unit	No impact on language mapping.

D.2.3.4 Group of Annotations: Data Implementation

Table D.5 identifies the mapping impact of the IDL defined Data Implementation annotations.

Table D.5: Data Implementation Annotation Impact

Data Implementation Annotation	Impact on C++ Language Mapping
@bit_bound	A **@bit_bound** annotation preceding an IDL **enum** declaration has no impact on the language mapping. The mapping for an IDL **bitmask** declaration preceded by the **@bit_bound** annotation is described in Clause D.2.1.3.2.
@external	IDL member declarations preceded by the **@external** annotation shall be mapped to any type that behaves similarly to a pointer (e.g., a plain pointer or a **_var** type). The chosen type shall support the concept of being “unset.” For example, a plain pointer is considered unset if its value is **NULL**. • In cases where the non-external mapping already uses a type similar to a pointer, it shall remain unchanged. • In cases where the non-external mapping uses a member of type **T**, **T** shall be replaced by pointer-to- **T**. For example, if plain pointers are used, **Short** shall be replaced by **Short***.

	The behavior of the construct, destructor, and copy functions of the enclosing object shall be the following: • The constructor shall set external member pointers to **`NULL`**. • The destructor shall delete the objects referenced by non- **`NULL`** external member pointers. It is the responsibility of the application to set the external member pointers to **`NULL`** before destroying the enclosing object if they do not want to delete specific referenced objects. • The copy function shall do a deep copy of the external members. If the destination of the external member is **`NULL`**, it shall be allocated. If the destination external member is not **`NULL`**, it shall be filled with a copy of the source member. If the copy operation of the external member fails, then the copy function of the containing object shall fail as well. This may happen when the destination member is not large enough to hold a copy of the source. • There may be an additional copy function that takes in arguments to allow the user to control the behavior of the copy operation. Such operation may allow the user to choose whether a shallow or deep copy is made, as well as whether any existing memory pointed by the member is reused, released, or replaced during the copy. In the case that a shallow copy is made, and the destination member is **`NULL`**, then the destination member pointer shall be set to the source member pointer. In the case that a deep copy is made, and the destination member pointer is **`NULL`**, memory for the destination member shall be allocated and then copied into. A member that is both external and optional shall be mapped as if it were external.
`@nested`	No impact on the language mapping

D.2.3.5 Group of Annotations: Code Generation

IDL defined Code Generation annotations shall be mapped as defined in Clause 7.17.5 of this specification.

D.2.3.6 Group of Annotations: Interfaces

IDL defined Interface annotations shall be mapped as defined in Clause 7.17.6 of this specification.

D.3 IDL4 Mappings Rules for C++11 Language Mapping Specification

The following clauses provide mapping rules for the portions of IDL4 that are not covered by the classic C++11 Language Mapping [OMG-C++11]. Such mapping rules are consistent with the requirements and mapping style of the original specification.

D.3.1 Extended Data Types

D.3.1.1 Structures with Single Inheritance

In addition to the requirements in Clause 6.14.1 of [OMG-C++11], 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 of [OMG-C++11] 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.
- The using declaration **`Base::Base`** which enables C++'s inheriting constructors.

For example:

```
struct VariableExt : Variable { // from Clause 6.14.1 of [OMG-C++11]
    boolean b;
};
```

would map to C++ as follows:

```
class VariableExt : public Variable {
public:
    VariableExt(const Variable&, bool);
    using Variable::Variable;
    // other members as specified in 6.14 and 6.14.1 of [OMG-C++11]
};
```

D.3.1.2 Union Discriminators

This building block defined in [OMG-IDL4] adds the **`wchar`** and **`octet`** IDL types to the set of valid types for a union discriminator. The mapping of union discriminators of such types shall be mapped as specified in Clause 6.14.2 of [OMG-C++11].

Any addition to the list of supported integer, **`char`**, **`boolean`**, or **`enum`** types as a result of the implementation of the Extended Data Types building block makes such types valid union discriminators as well. Therefore, if 8 bit integer values are supported (see Clause D.3.1.4), **`int8`** and **`uint8`** shall be treated as valid union discriminators and shall be mapped as specified in Clause D.3.1.4.

D.3.1.3 Additional Template Types

D.3.1.3.1 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++ type corresponding to the key type and the **`T`** parameter is the C++ 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 shall 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 runtime.

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 shall 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 shall 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 shall raise a **`BAD_PARAM`** system exception to signal the error.

For example, the following full declarations for both a bounded and an unbounded map:

```
typedef map<unsigned long, T> M1; // unbounded map
typedef map<string, T, 20> M2;    // bounded map
```

would map to C++ as follows:

```
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 D.6: 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 D.7: 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`**

D.3.1.3.2 Bitsets

IDL **`biset`** types shall be mapped as defined in Clause 7.14.3.2 of this specification.

D.3.1.3.3 Bitmask Types

IDL **`bitmask`** types shall be mapped as defined in Clause 7.14.3.3 of this specification.

For a **`bitmask`** type, the following additional members shall be available as part of its type trait. Because the bit mask type itself is just an alias for a built-in type, the traits template is specialized on the **`<Bitmask>Bits`** type.

Table D.8: Additional Traits Members for Bit Masks

Member	Definition
`bit_bound`	**`std::integral_constant`** type of value type **`uint32_t`** indicating the **`bit_bound`** of the Bit Mask.
`underlying_type`	The type mapped as the underlying type of the bit mask

D.3.1.4 8-bit Integer Types

IDL **`int8`** and **`uint8`** data types shall be mapped as defined in Clause 7.14.4 of this specification.

D.3.1.5 Explicitly-Named Integer Types

IDL explicitly-named integer types shall be mapped as defined in Clause 7.14.5 of this specification.

D.3.2 User-Defined Annotations

User-defined annotations are not propagated to the generated C++ code.

D.3.3 Standardized Annotations

D.3.3.1 Group of Annotations: General Purpose

Table D.9 identifies the mapping impact of the IDL defined General Purpose Annotations.

Table D.9: General Purpose Annotation Impact

General Purpose Annotation	Impact on C++ Language Mapping
`@id`	No impact on language mapping
`@autoid`	No impact on language mapping
`@optional`	Given the mapping rules defined in Clause 6.14 of [OMG-C++11], each IDL **`struct`** member has a corresponding C++ type (**`T`**) that is used as the type of that member in constructor and accessor parameter lists. Each IDL struct member annotated with **`@optional`** uses the template instantiation **`IDL::optional<T>`** in place of **`T`** itself. Implementations that support **`@optional`** shall provide the class template named optional in namespace **`IDL`** either as an alias of the ISO C++ standard library's **`std::optional`** or as an independent implementation with the

	same API and semantics.
	
	For example:
	`struct S {`
	`  string name;`
	`  @optional float age;`
	`};`
	
	would map to the following C++:
	`class S {`
	`public:`
	`  // other members not shown`
	`  void age(IDL::optional<float>);`
	`  IDL::optional<float> age() const;`
	`  IDL::optional<float>& age();`
	`};`

`@position`	Impacts the mapping of **`bitmask`** types as defined in Clause D.3.1.3.3 of this specification.

`@value`	Impacts the mapping of **`enum`** types, providing the value of the annotated enumerator.
	
	For example:
	`enum Color {`
	`  @value(1) red,`
	`  @value(2) green,`
	`  @value(3) blue`
	`};`
	
	would map to the following C++:
	`enum class Color : uint32_t {`
	`  red = 1,`
	`  green = 2,`
	`  blue =3`
	`};`

`@extensibility`	No impact on language mapping
`@final`	No impact on language mapping
`@mutable`	No impact on language mapping
`@appendable`	No impact on language mapping

D.3.3.2 Group of Annotations: Data Modeling

IDL defined Data Modeling annotations shall be mapped as defined in Clause 7.17.2 of this specification.

D.3.3.3 Group of Annotations: Units and Ranges

IDL defined Units and Ranges annotations shall be mapped as defined in Clause 7.17.3 of this specification.

D.3.3.4 Group of Annotations: Data Implementation

IDL defined Data Implementation annotations shall be mapped as defined in Clause 7.17.4 of this specification.

D.3.3.5 Group of Annotations: Code Generation

IDL defined Code Generation annotations shall be mapped as defined in Clause 7.17.5 of this specification.

D.3.3.6 Group of Annotations: Interfaces

IDL defined Interface annotations shall be mapped as defined in Clause 7.17.6 of this specification.