ORB Portability Joint Submission (Final), Part 1 of 2, orbos/97-05-15

BEA Systems, Inc.

Digital Equipment Corporation

Expersoft Corporation

Hewlett-Packard Company

International Computers Limited

IONA Technologies, PLC.

Novell, Inc.

SunSoft, Inc.

Telefónica Investigación y Desarrollo S.A.

May 20, 1997

Copyrights

Copyright © 1996 by BEA Systems, Inc.
Copyright © 1996 by Digital Equipment Corporation
Copyright © 1996 by Expersoft Corporation
Copyright © 1996 by Hewlett-Packard Company
Copyright © 1996 by International Computers Limited
Copyright © 1996 by IONA Technologies, PLC.
Copyright © 1996 by Novell, Inc.
Copyright © 1996 by SunSoft, Inc.
Copyright © 1996 by Telefónica Investigación y Desarrollo S.A.

The companies listed above hereby grant a royalty-free license to the Object Management Group, Inc. (OMG) for worldwide distribution of this document or any derivative works thereof, so long as the OMG reproduces the copyright notices and the below paragraphs on all distributed copies.

The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE,THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

This document contains information which is protected by copyright. All Rights Reserved. Except as otherwise provided herein, no part of this work 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 the permission of one of the copyright owners. All copies of this document must include the copyright and other information contained on this page.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to fifty copies) for their internal use as part of the OMG evaluation process.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision © (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013.

CORBA and Object Request Broker are trademarks of Object Management Group.

Contents

Contents . . . . . **iii**

Preface . . . . . **xiii**

0.1 Cosubmitting Companies xiii
0.2 Overview xiii
0.3 Contacts xiv
0.4 Proof of Concept. xv
0.5 IDL Testing xvi
0.6 Typography Conventions xvi

1 Overview and Rationale **1-1**

1.1 Overview 1-1
1.2 Rationale for the Object Adapter approach 1-2
1.2.1 BOA Begone 1-2
1.2.2 Portable Object Adapter 1-2
1.2.2.1 Incompatibility with BOA 1-3
1.2.2.2 Separate Module 1-3
1.2.2.3 Language Mapping for POA 1-3
1.2.2.4 Design Goals 1-3
1.2.2.5 Locality Contraints on Interfaces 1-4
1.3 Concurrency 1-4
1.4 The Current Object and operations. 1-4
1.5 The Policy Object and operations. 1-5
1.6 Other portability considerations 1-5
1.7 Subjects Not Addressed 1-5

2 Changes to CORBA and Services 2-1

2.1 Preface 2-1
2.2 Ch 2: CORBA Overview 2-1
2.3 Ch 3: OMG IDL Syntax and Semantics 2-3
2.4 Ch 4: Dynamic Invocation Interface 2-4
2.4.1 Section 4.1.3, Return Status and Exceptions 2-4
2.5 Ch 5: Dynamic Skeleton Interface 2-5
2.5.2 Section 5.2 Explicit Request State: ServerRequest Pseudo-Object . 2-5
2.5.3 Section 5.3 Dynamic Skeleton Interface: Language Mapping 2-7
2.5.4 Section 5.3.1 ServerRequest's Handling of Operation Parameters . 2-7
2.6 Ch 7: ORB Interface 2-7
2.6.4 Add all new 7.7, Thread-related operations 2-10
2.6.5 Add all new 7.8, Current Object 2-12
2.6.6 Add all new 7.9, Policy Object 2-13
2.7 Ch 8: BOA 2-14
2.8 New chapter: for Dynamic Any 2-14
2.9 Ch 14: C language mapping 2-14
2.10 Ch 16-18: C++ language mapping 2-14
2.11 Changes to Transaction Service 2-14
2.12 Changes to Security Service 2-15
2.13 Changes to Cobol Mapping 2-15

3 The Portable Object Adapter 3-1

3.1 Design Goals 3-1
3.2 Abstract Model Description 3-2
3.2.1 Model Components 3-2
3.2.2 Model Architecture 3-4
3.2.3 POA Creation 3-6
3.2.4 Reference Creation 3-6
3.2.5 Object Activation States 3-8
3.2.6 Request Processing 3-8
3.2.7 Implicit Activation 3-9
3.2.8 Multi-threading 3-10
3.2.8.1 POA Threading Models 3-10
3.2.8.2 Using the Single Thread Model 3-11
3.2.8.3 Using the ORB Controlled Model 3-11
3.2.8.4 Limitations When Using Multiple Threads 3-11
3.2.9 Dynamic Skeleton Interface 3-11
3.2.10 Location Transparency 3-12
3.3 Interfaces 3-13
3.3.1 The Servant IDL Type 3-13

3.3.2 POAManager Interface 3-14
3.3.2.1 Processing States 3-14
3.3.2.1.1 Active State 3-15
3.3.2.1.2 Discarding State 3-15
3.3.2.1.3 Holding State 3-15
3.3.2.1.4 Inactive State 3-16
3.3.2.2 Locality Constraints 3-16
3.3.2.3 activate 3-16
3.3.2.4 hold_requests 3-17
3.3.2.5 discard_requests 3-17
3.3.2.6 deactivate 3-17
3.3.3 AdapterActivator Interface 3-18
3.3.3.1 Locality Constraints 3-18
3.3.3.2 unknown_adapter 3-18
3.3.4 ServantManager Interface 3-19
3.3.4.1 Common information for servant manager types 3-20
3.3.4.2 Locality Constraints 3-20
3.3.5 ServantActivator Interface 3-20
3.3.5.1 incarnate 3-21
3.3.5.2 etherealize 3-22
3.3.6 ServantLocator Interface 3-23
3.3.6.1 preinvoke 3-24
3.3.6.2 postinvoke 3-24
3.3.7 POA Policy Objects 3-25
3.3.7.1 Thread Policy 3-25
3.3.7.2 Lifespan Policy 3-25
3.3.7.3 Object Id Uniqueness Policy 3-26
3.3.7.4 Id Assignment Policy 3-26
3.3.7.5 Servant Retention Policy 3-27
3.3.7.6 Request Processing Policy 3-27
3.3.7.6.1 RETAIN and USE_ACTIVE_OBJECT_MAP_ONLY 3-28
3.3.7.6.2 RETAIN and USE_SERVANT_MANAGER 3-28
3.3.7.6.3 RETAIN and USE_DEFAULT_SERVANT. 3-28
3.3.7.6.4 NON-RETAIN and USE_SERVANT_MANAGER: 3-28
3.3.7.6.5 NON-RETAIN and USE_DEFAULT_SERVANT: . 3-28
3.3.7.7 Implicit Activation Policy 3-29
3.3.8 POA Interface 3-29
3.3.8.1 Locality Constraints 3-29
3.3.8.2 create_POA 3-29
3.3.8.3 find_POA 3-30
3.3.8.4 destroy 3-30
3.3.8.5 Policy Creation Operations 3-31
3.3.8.6 the_name 3-31

3.3.8.7 the_parent . . . 3-32
3.3.8.8 the_POAManager . . . 3-32
3.3.8.9 the_activator . . . 3-32
3.3.8.10 get_servant_manager . . . 3-32
3.3.8.11 set_servant_manager . . . 3-32
3.3.8.12 get_servant . . . 3-33
3.3.8.13 set_servant . . . 3-33
3.3.8.14 activate_object . . . 3-33
3.3.8.15 activate_object_with_id . . . 3-33
3.3.8.16 deactivate_object . . . 3-34
3.3.8.17 create_reference . . . 3-34
3.3.8.18 create_reference_with_id . . . 3-35
3.3.8.19 servant_to_id . . . 3-35
3.3.8.20 servant_to_reference . . . 3-35
3.3.8.21 reference_to_servant . . . 3-36
3.3.8.22 reference_to_id . . . 3-36
3.3.8.23 id_to_servant . . . 3-37
3.3.8.24 id_to_reference . . . 3-37
3.3.9 Current operations . . . 3-37
3.3.9.1 get_POA . . . 3-37
3.3.9.2 get_object_id . . . 3-38
3.4 Consolidated IDL Related to PortableServer . . . 3-38
3.4.1 IDL in CORBA module concerning the POA . . . 3-38
3.4.2 IDL for POA and Related Interfaces . . . 3-39
3.5 UML Description of PortableServer . . . 3-46
3.6 Usage Scenarios . . . 3-48
3.6.1 Getting the root POA . . . 3-48
3.6.2 Creating a POA . . . 3-48
3.6.3 Explicit Activation with POA-assigned Object Ids . . . 3-48
3.6.4 Explicit activation with user assigned Object Ids . . . 3-50
3.6.5 Creating references before activation . . . 3-51
3.6.6 Servant Manager Definition and Creation . . . 3-51
3.6.7 Object activation on demand . . . 3-53
3.6.8 Persistent objects with POA-assigned Ids . . . 3-54
3.6.9 Multiple Object Ids Mapping to a Single Servant . . . 3-54
3.6.10 One Servant for all Objects . . . 3-55
3.6.11 Single Servant, many objects and types, using DSI . . . 3-58

4 Dynamic management of Any values . . . 4-1

4.1 Rationale . . . 4-1
4.2 Overview . . . 4-1
4.3 DynAny API . . . 4-2

4.3.1 Locality and usage constraints 4-5
4.3.2 Creating a DynAny object 4-5
4.3.3 The DynAny interface 4-6
4.3.3.1 Obtaining the TypeCode associated with a DynAny object . 4-7
4.3.3.2 Initializing a DynAny object from another DynAny object . 4-7
4.3.3.3 Initializing a DynAny object from an any value 4-7
4.3.3.4 Generating an any value from a DynAny object 4-7
4.3.3.5 Destroying a DynAny object.. 4-8
4.3.3.6 Creating a copy of a DynAny 4-8
4.3.3.7 Accessing a value of some basic type in a DynAny object . 4-8
4.3.3.8 Iterating through components of a DynAny 4-9
4.3.4 The DynFixed interface 4-10
4.3.5 The DynEnum interface 4-10
4.3.6 The DynStruct interface 4-10
4.3.7 The DynUnion interface. 4-12
4.3.8 The DynSequence interface 4-12
4.3.9 The DynArray interface 4-13
4.4 Usage in C++ language 4-14
4.4.1 Dynamic creation of CORBA::Any values 4-14
4.4.1.1 Creating an any which contains a struct. 4-14
4.4.2 Dynamic interpretation of CORBA::Any values 4-15
4.4.2.1 Filtering of events 4-15

5 Cobol Mapping 5-1

14 C Language Mapping 14-1

14.1 Requirements for a Language Mapping 14-1
14.1.1 Basic Data Types 14-2
14.1.2 Constructed Data Types 14-2
14.1.3 Constants 14-2
14.1.4 Objects 14-2
14.1.5 Invocation of Operations 14-2
14.1.6 Exceptions 14-3
14.1.7 Attributes 14-3
14.1.8 ORB Interfaces. 14-4
14.2 Scoped Names 14-4
14.3 Mapping for Interfaces 14-5
14.4 Inheritance and Operation Names 14-6
14.5 Mapping for Attributes. 14-7
14.6 Mapping for Constants 14-8
14.7 Mapping for Basic Data Types. 14-8
14.8 Mapping Considerations for Constructed Types 14-10

14.9 Mapping for Structure Types 14-10
14.10 Mapping for Union Types 14-11
14.11 Mapping for Sequence Types 14-11
14.12 Mapping for Strings 14-15
14.13 Mapping for Arrays 14-16
14.14 Mapping for Exception Types 14-16
14.15 Implicit Arguments to Operations 14-17
14.16 Interpretation of Functions with Empty Argument Lists 14-18
14.17 Argument Passing Considerations 14-18
14.18 Return Result Passing Considerations 14-19
14.19 Summary of Argument/Result Passing 14-20
14.20 Handling Exceptions 14-22
14.21 Method Routine Signatures 14-25
14.22 Include Files 14-26
14.23 Pseudo-objects 14-26
14.23.1 ORB Operations 14-27
14.24 Mapping for Object Implementations 14-27
14.24.1 Operation-specific Details 14-27
14.24.2 PortableServer Functions 14-28
14.24.3 Mapping for PortableServer::ServantLocator::Cookie 14-28
14.24.4 Servant Mapping 14-29
14.24.5 Interface Skeletons 14-30
14.24.6 Servant Structure Initialization. 14-32
14.24.7 Application Servants 14-34
14.24.8 Method Signatures 14-36
14.25 Mapping of the Dynamic Skeleton Interface to C 14-37
14.25.1 Mapping of ServerRequest to C 14-37
14.25.2 Mapping of Dynamic Implementation Routine to C 14-38
14.26 ORB Initialization Operations 14-42
14.26 ORB Initialization. 14-42
14.27 Operations for Obtaining Initial Object References 14-43

16 Mapping of OMG IDL to C++ 16-1

16.1 Preliminary Information 16-1
16.1.1 Scoped Names 16-1
16.1.2 C++ Type Size Requirements 16-2
16.1.3 CORBA Module. 16-2
16.2 Mapping for Modules 16-3
16.3 Mapping for Interfaces 16-3
16.3.1 Object Reference Types 16-4
16.3.2 Widening Object References 16-5
16.3.3 Object Reference Operations 16-5

16.3.4 Narrowing Object References 16-6
16.3.5 Nil Object Reference 16-7
16.3.6 Object Reference Out Parameter 16-7
16.3.7 Interface Mapping Example 16-8
16.4 Mapping for Constants 16-10
16.5 Mapping for Basic Data Types 16-11
16.6 Mapping for Enums 16-13
16.7 Mapping For String Types 16-13
16.8 Mapping for Structured Types 16-16
16.8.1 T_var Types 16-17
16.8.2 T_out Types 16-20
16.9 Mapping for Struct Types 16-21
16.10 Mapping for Union Types 16-24
16.11 Mapping for Sequence Types 16-28
16.11.1 Sequence Example 16-30
16.11.2 Using the “release” Constructor Parameter 16-31
16.11.3 Additional Memory Management Functions 16-32
16.11.4 Sequence T_var and T_out Types 16-33
16.12 Mapping For Array Types 16-33
16.13 Mapping For Typedefs 16-35
16.14 Mapping for the Any Type 16-36
16.14.1 Handling Typed Values 16-37
16.14.2 Insertion into Any 16-37
16.14.3 Extraction From Any 16-40
16.14.4 Distinguishing boolean, octet, char, and bounded string 16-42
16.14.5 Widening to Object 16-44
16.14.6 Handling Untyped Values 16-45
16.14.7 Any Constructors, Destructor, Assignment Operator 16-46
16.14.8 The Any Class 16-47
16.14.9 The Any_var Class 16-47
16.15 Mapping for Exception Types 16-47
16.16 Mapping For Operations and Attributes 16-50
16.17 Implicit Arguments to Operations 16-51
16.18 Argument Passing Considerations 16-51
16.18.1 Operation Parameters and Signatures 16-54

17 Mapping of Pseudo Objects to C++ 17-1

17.1 Usage 17-1
17.2 Mapping Rules 17-2
17.3 Relation to the C PIDL Mapping 17-3
17.4 Environment 17-3
17.4.1 Environment Interface 17-4

17.4.2 Environment C++ Class 17-4
17.4.3 Differences from C-PIDL. 17-4
17.4.4 Memory Management 17-4
17.5 NamedValue. 17-5
17.5.1 NamedValue Interface 17-5
17.5.2 NamedValue C++ Class. 17-5
17.5.3 Differences from C-PIDL. 17-5
17.5.4 Memory Management 17-5
17.6 NVList 17-6
17.6.1 NVList Interface. 17-6
17.6.2 NVList C++ Class 17-6
17.6.3 Differences from C-PIDL. 17-7
17.6.4 Memory Management 17-7
17.7 Request 17-7
17.7.1 Request Interface 17-10
17.7.2 Request C++ Class 17-10
17.7.3 Differences from C-PIDL. 17-11
17.7.4 Memory Management 17-12
17.8 Context 17-12
17.8.1 Context Interface 17-12
17.8.2 Context C++ Class 17-13
17.8.3 Differences from C-PIDL. 17-13
17.8.4 Memory Management 17-14
17.9 TypeCode 17-14
17.9.1 TypeCode Interface 17-14
17.9.2 TypeCode C++ Class 17-15
17.9.3 Differences from C-PIDL. 17-16
17.9.4 Memory Management 17-16
17.10 ORB 17-16
17.10.1 ORB Interface 17-16
17.10.2 ORB C++ Class 17-17
17.10.3 Differences from C-PIDL. 17-17
17.10.4 Mapping of ORB Initialization Operations 17-18
17.10.5 Mapping of Operations to Obtain Initial Object References . . 17-19
17.11 Object 17-19
17.11.1 Object Interface 17-20
17.11.2 Object C++ Class 17-20

18 Server-Side Mapping 18-1

18.1 Implementing Interfaces. 18-1
18.1.1 Mapping of PortableServer::Servant 18-1
18.1.2 Skeleton Operations 18-2

18.1.3 Inheritance-Based Interface Implementation 18-4
18.1.4 Delegation-Based Interface Implementation 18-5
18.2 Implementing Operations 18-9
18.2.1 Skeleton Derivation From Object 18-11
18.3 Mapping of Dynamic Skeleton Interface to C++ 18-11
18.3.1 Mapping of ServerRequest to C++ 18-11
18.3.2 Handling Operation Parameters and Results 18-12
18.3.3 Mapping of PortableServer Dynamic Implementation Routine. 18-12
18.4 PortableServer Functions 18-13
18.5 Mapping for PortableServer::ServantManager 18-14
18.5.1 Mapping for Cookie 18-14
18.5.2 ServantManagers and AdapterActivators. 18-14

C C++ Definitions for CORBA C-1

C.1 Primitive Types C-1
C.2 String_var Class C-2
C.3 Any Class C-3
C.4 Any_var Class C-5
C.5 Exception Class. C-5
C.6 SystemException Class C-5
C.7 UserException Class C-6
C.8 UnknownUserException Class C-6
C.9 release and is_nil. C-6
C.10 Object Class C-7
C.11 Environment Class C-8
C.12 NamedValue Class C-8
C.13 NVList Class C-8
C.14 ExceptionList Class C-9
C.15 ContextList Class C-9
C.16 Request Class C-9
C.17 Context Class C-10
C.18 TypeCode Class C-10
C.19 ORB Class. C-11
C.20 ORB Initialization C-12
C.21 General T_out Types. C-12

D Alternative Mappings For C++ Dialects D-1

D.1 64-bit Integers D-1
D.2 Without Namespaces D-1
D.3 Without Exception Handling D-1

G C++ Keywords G-1

Preface

0.1 Cosubmitting Companies

The following companies are pleased to jointly submit this specification in response to the OMG "ORB Portability Enhancement RFP" (OMG TC Document 95-6-26):

BEA Systems, Inc.
Digital Equipment Corporation
Expersoft Corporation
Hewlett-Packard Company
International Computers Limited
IONA Technologies Limited
Novell, Inc.
SunSoft, Inc.
Telefónica Investigación y Desarrollo S.A.

0.2 Overview

This submission consists of two OMG documents: orbos/97-05-15 and orbos/97-05-16. The submission is in two parts because of two different document processing systems being used.

The only difference between these documents and the previous ones (orbos/97-04-14 and orbos/97-04-15) is the application of the errata (orbos/97-05-02). Those documents were recommended by the ORBOS Task Force; these documents are the final form of the recommendation. This Preface has also changed to reflect the state of the submission.

Chapter 1 contains the design justification for the submission.

Chapter 2 contains the detailed editing instructions for changing the CORBA 2.0 document.

Chapter 3 contains the replacement chapter for Chapter 8 in CORBA 2.0, replacing the BOA with the POA.

Chapter 4 contains material for a new chapter for CORBA 2.0 on “Dynamic management of Any values”.

The contents of Chapters 5, 14, 16, 17, 18, and Appendixes C, D, and G are revisions of the language mappings for Cobol, C, and C++. Change bars in those chapters represent changes from the previous version of those mapping, not changes from the previous draft of this submission.

0.3 Contacts

Dan Frantz
BEA Systems, Inc.
ZKO2-2/R80
110 Spit Brook Road
Nashua, NH 03062, USA
phone: +1 603-881-2272
fax: +1 603-881-0120
email: frantz@send.enet.dec.com

Rudolf Riess
Digital Equipment Corporation
AK02-3/D10
50 Nagog Park
Acton, MA 01720-3499, USA
phone: +1 617-279-3260
email: rudolf.riess@ako.mts.dec.com

Joey Garon
Expersoft Corporation
5825 Oberlin Drive
San Diego, CA 92121, USA
phone: +1 619-824-4135
fax: +1 619-824-4110
email: jgaron@expersoft.com

Bob Kukura
Hewlett-Packard Company
300 Apollo Drive
Chelmsford, MA 01824, USA
phone: +1 508 436 4512
fax: +1 508 436 5176
email: kukura@apollo.hp.com

Richard Herbert
ICL High Performance Systems
Wenlock Way
West Gorton
Manchester
M12 5DR
England
phone: +44 161 223 1301 x3266
email: R.Herbert@man0521.wins.icl.co.uk

Martin Chapman
IONA Technologies, PLC.
The IONA Building 8-10 Lower Baggot Street
Dublin 2, Ireland
phone: +353 1-6625255
fax: +353 1-6625244
email: mchapman@iona.com

Mark Ericson
Novell, Inc.
122 East 1700 South
M/S C-11-1
Provo, UT 84606-6194, USA
phone: +1 801-429-5486
email: marke@novell.com

Alan Snyder
SunSoft, Inc.
2500 Garcia Avenue UMTV18-210
Mountain View CA 94043-1100, USA
phone: +1 415 786 4883
fax: +1 415 786 4098
email: alan.snyder@eng.sun.com

Juan José Hierro
Telefónica Investigación y Desarrollo S.A.
Emilio Vargas 6
Madrid 28043, Spain
phone: +34 1-337-4741
fax: +34 1-337-4222
email: jhierro@tid.es

0.4 Proof of Concept

The POA is major new work, but much of it includes techniques and ideas that are in the submitters' experience with implementing the BOA and other object adapters. During the design process, some prototyping and detailed design work has been undertaken by the submitters. We expect that products based on this design will be available, in the customary manner, within the OMG's timelines.

0.5 IDL Testing

At the time of this draft, the consolidated IDL for the portable server in Section 3.4, “Consolidated IDL Related to PortableServer”, and the IDL for the Dynamic Any capability in Section 4.3, “DynAny API”, have been successfully compiled on the following IDL compilers.

- BEA ObjectBroker
- Expersoft CORBAplus
- HP ORBPlus
- IONA Orbix
- SunSoft Neo

0.6 Typography Conventions

Ordinary text appears in this font.

IDL statements appear in this font as do IDL references embedded in the text.

PIDL statements appear in this font as do PIDL references embedded in the text.

```
C and C++ statements appear in this font as do such refer-
ences embedded in the text.
```

Overview and Rationale *1*

1.1 Overview

The ORB Portability Enhancement RFP is somewhat unusual in OMG in that it has no real requirements listed. It has only a list of problems to be addressed, most in the definition of the BOA. It does not require that each of these problems be fixed, but leaves the submitters to treat “ORB Portability” as an open-ended problem.

This specification takes full advantage of the free hand given the submitters by proposing something radical: throw away that part of the CORBA V2.0 spec (the BOA) that created all the problems and replace it with something that works (a somewhat similar entity, called the Portable Object Adapter or POA). The entire list of problems in the RFP are thus answered by the new approach in the POA. In order to create some of the behavior necessary, the submission adds a single IDL keyword, **native**, to provide the POA with special knowledge of interface implementations. In addition, the submission adjusts the C and C++ mappings to take the POA (rather than the BOA) into account, to take care of some slight difficulties in the current server side mappings, and to map the meaning of the IDL keyword **native**.

Topics other than Object Adapters affect portability. Those addressed in this submission are:

- A new “Dynamic Any” interface
- A more portable DSI can now be defined because of work done in the Portable Object Adapter.
- Modifications and generalizations of the Current and Policy objects defined in the Security specification, so that they can effectively be used by the POA.
- Certain aspects of server side mappings for C and C++, as requested in the RFP (signatures, at the very least). In addition, the C and C++ bindings must change to take into account the POA instead of the BOA.
- Problems with the interpretation of ORB_init and CORBA::Status.

Finally, this submission lists topics in portability that are best left to other RFPs in Section 1.7, “Subjects Not Addressed”.

1.2 Rationale for the Object Adapter approach

This section contains the rationale for many of the design decisions that went into the POA specification.

1.2.1 BOA Begone

Since the BOA was underspecified in CORBA 1.0 (and following), each implementor of an ORB had to make many interpretations and extensions to provide a complete product. These interpretations and extensions were incompatible with each other. They are now enshrined in each implementation, with many users depending on them for production applications. In a sense, there are now *many* BOAs, not just one, and none of them are the OMG’s BOA. Each implementation’s BOA has effectively become a proprietary extension of CORBA in that implementation.

What then to do for portability? One possibility considered was to make new interpretations and extensions of the OMG BOA and then require conformance to the new specification. However, such an act would invalidate many CORBA users’ code (application code as well as ORB code). For the sake of compatibility for the current users of CORBA, the proprietary BOAs must remain in place.

Since there are so many incompatible BOAs, *any* change in the CORBA spec to allow portability will require new user code to become portable. It simply is not possible to create a no-effort upgrade that magically makes current applications portable. The only question remains on what changes to make so that new work undertaken by users will be portable.

The submitters propose a solution with the following two parts.

- Do not improve the CORBA BOA. Delete it. Each implementor can keep its proprietary BOA as long as it wishes to support its current customers. Current programs continue to work and are supported by their ORB vendors, but OMG no longer publishes BOA specs.
- Create new object adapters that will be portable. The submitters hope that our experience with the BOA will make these new specifications more portable than before.

1.2.2 Portable Object Adapter

The Portable Object Adapter (POA) is somewhat similar to the BOA, but is more tightly specified to increase portability. It is also richer in functionality since the submitters have had requests from CORBA users for certain features.

1.2.2.1 Incompatibility with BOA

A possibility for design of the POA was to have it inherit from the BOA and merely add the new technical details. This alternative was rejected because there would still have to be incompatible clarifications of the BOA; these would, as argued above, invalidate current users' code.

1.2.2.2 Separate Module

The design of the POA has its IDL in its own module, not in the CORBA module as the BOA was. This separates further enhancements of the POA from the underlying ORB and perhaps makes it easier in the future to add additional types of object adapters. In addition, this separation may allow clients to be slightly more lightweight because support for the CORBA module no longer needs server side constructs.

1.2.2.3 Language Mapping for POA

There shall be a POA mapping for each language; indeed, the definition of a language mapping includes a mapping for the POA just as it used to for the BOA, although the magnitude of the mapping it much smaller since the POA is not a pseudo-object as was the BOA. The POA is specified in IDL so most of the mapping is automatic. The only part left is the mapping in each language of objects declared using the new IDL **native** keyword.

This proposal supplies language mappings to the POA for C, C++, and COBOL. It was not practical to provide mappings for the other languages (Smalltalk and Ada) at this time. We suggest that the OMG issue new RFPs for these language mappings so that the respective language experts can respond to the need.

1.2.2.4 Design Goals

The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities. More precisely, the POA is designed to allow programmers to build object implementations that can provide consistent service for objects whose lifetimes (from the perspective of a client holding a reference for such an object) span multiple server lifetimes.
- Provide support for transparent activation of objects.
- Allow a single *servant* (an implementation of the methods of an interface) to support multiple object identities simultaneously.
- Allow multiple distinct instances of the POA to exist in a server.
- Provide support for transient objects with minimal programming effort and overhead.
- Provide support for implicit activation of servants with POA-allocated object IDs.

- Allow object implementations to be maximally responsible for an object's behavior. Specifically, an implementation can control an object's behavior by establishing the datum that defines an object's identity, determining the relationship between the object's identity and the object's state, managing the storage and retrieval of the object's state, providing the code that will be executed in response to requests, and determining whether or not the object exists at any point in time.
- Avoid requiring the ORB to maintain persistent state describing individual objects, their identities, where their state is stored, whether certain identity values have been previously used or not, whether an object has ceased to exist or not, and so on.
- Provide an extensible mechanism for associating policy information with objects implemented in the POA.
- Allow programmers to construct object implementations that inherit from static skeleton classes, generated by OMG IDL compilers, or the standard dynamic implementation class defined in DSI.

1.2.2.5 Locality Contraints on Interfaces

Some of the interfaces described in this specification are for objects that are, of necessity, local to the process in which they are created and used. At the time of this writing, the ORBOS TF is discussing potential formal mechanisms for specifying and implementing such aobjects. (This is in the tentative "Pseudo-object RFP".) Though it is not certain, it seems likely that such mechanisms will exist in the future. If and when they do exist, they should be applied to the local interfaces described in this specification. In lieu of a formal mechanism, this specification describes informal locality constraints on the interfaces in question and requires that appropriate exceptions be raised when these references are passed outside the local process or externalized with object_to_string. It is the intent of this specification that the objects so described are first-class CORBA objects whose interfaces are described in IDL and whose references will be of type CORBA::Object (more precisely, the programming language-specific mapping for CORBA::Object).

1.3 Concurrency

The specification for POA supports both single-threaded and multi-threaded ORBs and implementations. This specification intentionally does not supply a CORBA-specific interface to threading primitives, but it does specify how these Object Adapters must behave in a multi-threaded environment.

1.4 The Current Object and operations

The Current object was added to the CORBA module by the Object Transaction Service and extended by the Security Service. This specification's POA uses Current objects and provides a way of obtaining a Current object specific to the POA. The submitters worked with the Security RTF to generalize the Transaction and Security method of obtaining their Current objects in order to be more useful for future CORBA services that may employ this useful concept.

1.5 The Policy Object and operations

The Policy object was added to the CORBA module by the the Security Service. This specification uses Policy objects and generalizes the Security method of using object so that they may be more useful for future CORBA services.

1.6 Other portability considerations

In addition to the Object Adapters, this submission also includes some clarifications to other parts of the CORBA V2.0 specification to enhance portability in other ways. In particular, there is a new interface for dealing with Dynamic Any, making the DSI more portable, a refined definition of the Current object, and some changes of the C, C++, and Cobol bindings for interfaces (deleting the BOA, adding the POA, changing the C, C++, and Cobol server side mappings, since these are directly affected by the POA).

1.7 Subjects Not Addressed

The submitters recognize that there are other portability problems than those presented here. They are currently being addressed by revision task forces or RFP. For completeness, the issues considered for this submission but deferred to other groups are the following.

- Semantics of IDL **#include** are cloudy. Some vendors treat it as including only data types and others include the entire body of the referenced file. While the effect of the difference is somewhat ameliorated by administrative functions of a vendor's IDL compiler, more should be specified to avoid confusion.
- Interceptors as specified in the approved Security submission need more work, so this submission does not deal with them. Issues we might have addressed (and that the Security RTF may choose to specify) are how interceptors fit into the different levels of operations of the POA. In particular, POAs might have to be created, instance managers started, servants started, and object activated.
- Interpretation of PIDL constructs is a current ORBOS topic and seems likely to take a considerable amount of discussion. This specification doesn't attempt to solve it. In the meantime, we have need of two kinds of behaviors and specify them in some detail.
  - Some objects need to have locality constraints. We have made a first attempt at describing what those constraints might be but have not proposed any syntax to specify such constraints or objects. That is, this spec gives only word constraints on certain objects.
  - An IDL-specified object adapter requires an escape mechanism from IDL in order to connect the programming language constructs that implement objects with the object adapter. To this end, the **native** IDL keyword is introduced. Although this mechanism might be applicable in situations currently addressed via PIDL, the submitters recognize that a complete solution to the PIDL issues might need to modify this or provide additional mechanisms.

Changes to CORBA and Services 2

All the changes are to the July, 1996 version of the CORBA 2.0 document.

Relatively small changes are noted by lifting whole paragraphs from CORBA 2.0 and underlining the changes/additions. This is to allow the evaluators of the proposal to avoid having to go back to the CORBA 2.0 document to understand them. By far, the largest number of the small items are in changing references to BOA to be references to the POA. In some cases, significant deletions are show with ~~strikethroughs~~; these are usually whole paragraphs; where changes aren't significant, deletions aren't shown in order to maintain readability.

There are several large changes. Chapters are added to the CORBA 2.0 document for the POA and Dynamic Any. Changes to the C and C++ bindings are most easily specified by including all their chapters, with changes being shown with change-bars; deletions are not shown.

2.1 Preface

2.1.1 Section 0.5, Structure of this Manual

Delete "Basic Object Adapter" and insert the following.

> **Portable Object Adapter** describes a group of IDL interfaces that an implementation uses to access ORB functions.

2.2 Ch 2: CORBA Overview

2.2.1 Section 2.4, Structure of an Implementation.

First paragraph after the diagram, change to:

Because of the range of possible object implementations, it is difficult to be definitive about how an object implementation is structured. See the chapters on the Portable Object Adapter.

2.2.2 Section 2.4, Structure of an Object Implementation.

Last paragraph, change to:

Most object implementations provide their behavior using facilities in addition to the ORB and object adapter. For example, although the Portable Object Adapter provides some persistent data associated with an object (its OID or Object ID), that relatively small amount of data is typically used as an identifier for the actual object data stored in a storage service of the object implementation's choosing. With this structure, it is not only possible for different object implementations to use the same storage service, it is also possible for objects to choose the service that is most appropriate for them.

2.2.3 Section 2.6 Example Object Adapters

Replace with the following sections. Note that this deletes the old Sections 2.6.2, "Library Object Adapter" (since this tended to confuse users) and 2.6.3 Object-Oriented Database Adapter since it is described in more detail as an example in the last paragraph of Section 2.5. It appears that most vendors implement the kind of functionality originally intended as an LOA as part of their products without distinguishing it from their BOA (or POA). A few words at the beginning of the new 2.6 are the same as at the beginning of the old 2.6, but it isn't worth showing cross-outs and underlines since almost the entire section is new.

2.6 CORBA Required Object Adapter

There are a variety of possible object adapters. However, since the object adapter interface is something that object implementations depend on, it is desirable that there be as few as practical. Most object adapters are designed to cover a range of object implementations, so only when an implementation requires radically different services or interfaces should a new object adapter be considered. In this section, we briefly describe the object adapter defined in this specification.

2.6.1 Portable Object Adapter

This specification defines a Portable Object Adapter that can be used for most ORB objects with conventional implementations. (See the Portable Object Adapter chapter for more information.) The intent of the POA, as its name suggests, is to provide an Object Adapter that can be used with multiple ORBs with a minimum of rewriting needed to deal with different vendors' implementations.

This specification allows several ways of using servers but it does not deal with the administrative issues of starting server programs. Once started, however, there can be a servant started and ended for a single method call, a separate servant for each object, or a shared servant for all instances of the object type. It allows for groups

of objects to be associated by means of being registered with different instances of the POA object and allows implementations to specify their own activation techniques. If the implementation is not active when an invocation is performed, the POA will start one. The POA is specified in IDL, so its mapping to languages is largely automatic, following the language mapping rules. (The primary task left for a language mapping is the definition of the Servant type.)

2.2.4 Section 2.7, The Integration of Foreign Object Systems

In several places, change references to “BOA” to “POA” and references to “Basic Object Adapter” to “Portable Object Adapter”.

2.3 Ch 3: OMG IDL Syntax and Semantics

2.3.1 Section 3.4 OMG IDL Grammar.

Extend the IDL grammar with a new type declaration alternative:

(27) <type_dcl> ::= ...
...
| "native" <simple_declarator>

2.3.2 Section 3.8, Type Declaration

In the first paragraph, replace "struct, union, and enum" with "struct, union, enum, and native". Add the above syntax change to the grammar.

2.3.3 Add a new section 3.8.5

3.8.5 Native Types

OMG IDL provides a declaration for use by object adapters to define an opaque type whose representation is specified by the language mapping for that object adapter.

The syntax is:

<type_dcl> ::= "native" <simple_declarator>
<simple_declarator> ::= <identifier>

This declaration defines a new type with the specified name. A native type is similar to an IDL basic type. The possible values of a native type are language-mapping dependent, as are the means for constructing them and manipulating them. Any interface that defines a native type requires each language mapping to define how the native type is mapped into that programming language.

A native type may be used to define operation parameters and results. However, there is no requirement that values of the type be permitted in remote invocations, either directly or as a component of a constructed type. Any attempt to transmit a value of a native type in a remote invocation may raise the MARSHAL standard exception.

It is recommended that native types be mapped to equivalent type names in each programming language, subject to the normal mapping rules for type names in that language. For example, in a hypothetical Object Adapter IDL module

```
module HypotheticalObjectAdapter {
    native Servant;
    interface HOA {
        Object activate_object(in Servant x);
    };
};
```

the IDL type Servant would map to HypotheticalObjectAdapter::Servant in C++ and the activate_object operation would map to the following C++ member function signature:

```
CORBA::Object_ptr activate_object(
    HypotheticalObjectAdapter::Servant x);
```

The definition of the C++ type HypotheticalObjectAdapter::Servant would be provided as part of the C++ mapping for the HypotheticalObjectAdapter module.

Note – The native type declaration is provided specifically for use in object adapter interfaces, which require parameters whose values are concrete representations of object implementation instances. It is strongly recommended that it not be used in service or application interfaces. The native type declaration allows object adapters to define new primitive types without requiring changes to the OMG IDL language or to OMG IDL compilers.

2.4 Ch 4: Dynamic Invocation Interface

2.4.1 Section 4.1.3, Return Status and Exceptions

Replace Section 4.1.2 with the following.

> In the Dynamic Invocation interface, routines typically indicate errors or exceptional conditions either via programming language exception mechanisms, or via an Environment parameter for those languages that do not support exceptions. Thus, the return type of these routines is void.

It is uncertain how how these changes work their way into the CORBA document. If the change is part of a major revision (for example, CORBA 3), the above change is sufficient. If they become part of CORBA 2 (a minor revision), the following should also be included.

```
// IDL
typedef void Status;
```

Previous versions of CORBA allowed implementations to choose the type they returned from these routines by specifying the return type as a typedef named CORBA::Status. Implementations were allowed to define this typedef as either type void or as unsigned long. Due to the portability problems resulting from this approach, the unsigned long definition of Status is deprecated.

The Status type has been left in the CORBA module for reasons of backwards compatibility. In the next major revision of CORBA it will be removed entirely and all instances of it will be replaced with void.

2.5 Ch 5: Dynamic Skeleton Interface

2.5.1 Section 5.1, Overview

Replace last paragraph with:

This chapter describes the elements of the DSI that are common to all object adapters that provide a DSI. See Section 3.6.11, "Single Servant, many objects and types, using DSI' for the specification of the DSI for the Portable Object Adapter.

2.5.2 Section 5.2 Explicit Request State: ServerRequest Pseudo-Object

Replace the entire section with:

The ServerRequest pseudo-object captures the explicit state of a request for the DSI, analogous to the Request pseudo-object in the DII. The object adapter dispatches an invocation to a DSI-based object implementation by passing an instance of ServerRequest to the DIR associated with the object implementation. The following shows how it provides access to the request information:

```
module CORBA {
    ...
    pseudo interface ServerRequest {
        readonly attribute Identifier    operation;
        void          arguments(inout NVList nv);
        Context       ctx();
        void          set_result(in Any val);
        void          set_exception(in Any val);
    };
};
```

The identity and/or reference of the target object of the invocation is provided by the object adapter and its language mapping. In the context of a bridge, the target object will typically be a proxy for an object in some other ORB.

The **operation** attribute provides the identifier naming the operation being invoked; according to OMG IDL's rules, these names must be unique among all operations supported by the object's "most-derived" interface. Note that the operation names for getting and setting attributes are **_get_<attribute_name>** and **_set_<attribute_name>**, respectively. The operation attribute can be accessed by the DIR at any time.

Operation parameter types will be specified, and "in" and "inout" argument values will be retrieved, with **arguments**. Unless it calls **set_exception**, the DIR must call **arguments** exactly once, even if the operation signature contains no parameters. Once **arguments** or **set_exception** has been called, calling **arguments** on the same **ServerRequest** will result in a **BAD_INV_ORDER** system exception. The DIR must pass in to **arguments** an **NVList** initialized with **TypeCodes** and **Flags** describing the parameter types for the operation, in the order in which they appear in the IDL specification (left to right). A potentially-different **NVList** will be returned from **arguments**, with the "in" and "inout" argument values supplied. If it does not call **set_exception**, the DIR must supply the returned **NVList** with return values for any "out" arguments before returning, and may also change the return values for any "inout" arguments.

When the operation is not an attribute access, and the operation's IDL definition contains a context expression, **ctx** will return the context information specified in IDL for the operation. Otherwise it will return a nil **Context** reference. Calling **ctx** before **arguments** has been called or after **ctx**, **set_result** or **set_exception** has been called will result in a **BAD_INV_ORDER** system exception.

The **set_result** operation is used to specify any return value for the call. Unless **set_exception** is called, if the invoked operation has a non-void result type, **set_result** must be called exactly once before the DIR returns. If the operation has a void result type, **set_result** may optionally be called once with an **Any** whose type is **tk_void**. Calling **set_result** before **arguments** has been called or after **set_result** or **set_exception** has been called will result in a **BAD_INV_ORDER** system exception. Calling **set_result** without having previously called **ctx** when the operation IDL contains a context expression, or when the **NVList** passed to **arguments** did not describe all parameters passed by the client, may result in a **MARSHAL** system exception.

The DIR may call **set_exception** at any time to return an exception to the client. The **Any** passed to **set_exception** must contain either a system exception or one of the user exceptions specified in the **raises** expression of the invoked operation's IDL definition. Passing in an **Any** that does not contain an exception will result in a **BAD_PARAM** system exception. Passing in an unlisted user exception will result in either the DIR receiving a **BAD_PARAM** system exception or in the client receiving an **UNKNOWN_EXCEPTION** system exception.

See each language mapping for a description of the memory management aspects of the parameters to the **ServerRequest** operations.

2.5.3 Section 5.3 Dynamic Skeleton Interface: Language Mapping

Change the last paragraph to:

> Section 14.24, Mapping of the Dynamic Skeleton Interface to C, and Section 16.17, Mapping of Dynamic Skeleton Interface to C++, provide mappings of the Dynamic Skeleton Interface (supporting the POA) to the C language and C++ languages.

2.5.4 Section 5.3.1 ServerRequest's Handling of Operation Parameters

Remove third paragraph (which deals with setting exceptions).

2.5.5 Section 5.3.2, Registering Dynamic Implementation Routines.

Replace Section 5.3.2 with the following.

> In an ORB implementation, the Dynamic Skeleton Interface is supported entirely through the Object Adapter. An Object Adapter does not have to support the Dynamic Skeleton Interface but, if it does, the Object Adapter is responsible for the details.

2.6 Ch 7: ORB Interface

2.6.1 Section 7.2.1, Object Reference Operations

7.2 has the CORBA::Object PIDL. 7.2.1's last sentence says "See the Basic Object Adapter chapter for information about the Implementation Repository." Delete the sentence.

2.6.2 Sections 7.3-7.5, ORB and OA Initialization

Section 7.3, Overview: ORB and OA Initialization and Initial References; Section 7.4, ORB Initialization; Section 7.5, OA and BOA Initialization

Replace 7.3 and 7.4 with following and delete 7.5.

Section 7.3, Overview: ORB and OA Initialization and Initial References

Before an application can enter the CORBA environment, it must first:

- Be initialized into the ORB and possibly the object adapter (POA) environments.

- Get references to ORB pseudo-object (for use in future ORB operations) and perhaps other objects (including the root POA or some Object Adapter objects).

CORBA V2.0 provides operations, specified in PIDL, to initialize applications and obtain the appropriate object references. The following is provided:

- Operations providing access to the ORB. These operations reside in the CORBA module, but not in the ORB interface and are described in Section 7.4, "ORB Initialization," on page 7-6.
- ~~Operations providing access to the Basic Object Adapter (BOA) and other object adapters (OAs) These operations reside in the ORB interface and are described in Section 7.5, "OA and BOA Initialization," on page 7-8.~~
- Operations providing access to Object Adapters, Interface Repository, Naming Service, and other Object Services. These operations reside in the ORB interface and are described in Section 7.6, "Obtaining Initial Object References," on page 7-10.

In addition, this manual provides a mapping of the PIDL initialization and object reference operations to the C and C++ programming languages. For mapping information, refer to Section 14.26, "ORB and OA~~/BOA~~ Initialization Operations," on page 14-31 and to Section 17.12, "ORB," on page -18.

Section 7.4, ORB Initialization

When an application requires a CORBA environment it needs a mechanism to get the ORB pseudo-object reference and possibly an OA object reference (such as the root POA). This serves two purposes. First, it initializes an application into the ORB and OA environments. Second, it returns the ORB pseudo-object reference and the OA object reference to the application for use in future ORB and OA operations.

The ORB and OA initialization operations must be ordered with ORB occurring before OA: an application cannot call OA initialization routines until ORB initialization routines have been called for the given ORB. The operation to initialize an application in the ORB and get its pseudo-object reference is not performed on an object. This is because applications do not initially have an object on which to invoke operations. The ORB initialization operation is an application's bootstrap call into the CORBA world. The PIDL for the call (Figure 7-1) shows that the ORB_init call is part of the CORBA module but not part of the ORB interface.

Applications can be initialized in one or more ORBs. When an ORB initialization is complete, its pseudo reference is returned and can be used to obtain other references for that ORB.

In order to obtain an ORB pseudo-object reference, applications call the **ORB_init** operation. The parameters to the call comprise an identifier for the ORB for which the pseudo-object reference is required, and an **arg_list**, which is used to allow environment-specific data to be passed into the call. PIDL for the ORB initialization is as follows:

```
// PIDL
module CORBA {
    typedef string ORBid;
    typedef sequence <string> arg_list;
    ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

Figure 7-1

The identifier for the ORB will be a name of type CORBA::ORBid. All ORBid strings other than the empty string are allocated by ORB administrators and are not managed by the OMG. ORBid strings other than the empty string are intended to be used to uniquely identify each ORB used within the same address space in a multi-ORB application. These special ORBid strings are specific to each ORB implementation and the ORB administrator is responsible for ensuring that the names are unambiguous.

If an empty ORBid string is passed to ORB_init, then the arg_list arguments shall be examined to determine if they indicate an ORB reference that should be returned. This is achieved by searching the arg_list parameters for one preceded by "-ORBid," for example, "-ORBid example_orb" (the whitespace after the "-ORBid" tag is ignored) or "-ORBidMyFavoriteORB" (with no whitespace following the "-ORBid" tag). Alternatively, two sequential parameters with the first being the string "-ORBid" indicates that the second is to be treated as an ORBid parameter. If an empty string is passed and no arg_list parameters indicate the ORB reference to be returned, the default ORB for the environment will be returned.

Other parameters of significance to the ORB can also be identified in arg_list, for example, "Hostname," "SpawnedServer," and so forth. To allow for other parameters to be specified without causing applications to be re-written, it is necessary to specify the parameter format that ORB parameters may take. In general, parameters shall be formatted as either one single arg_list parameter:

–ORB<suffix><optional whitespace> <value>

or as two sequential arg_list parameters:

-ORB<suffix>

<value>

Regardless of whether an empty or non-empty ORBid string is passed to ORB_init, the arg_list 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 arg_list are ignored. All other -ORB<suffix> parameters in the arg_list may be of significance during the ORB initialization process.

The ORB_init operation may be called any number of times and shall return the same ORB reference when the same ORBid string is passed, either explicitly as an argument to ORB_init or through the arg_list. All other -ORB<suffix> parameters in the arg_list may be considered on subsequent calls to ORB_init.

2.6.3 Section 7.6, Obtaining Initial Object References.

Change the first two sentences to:

> Applications require a portable means by which to obtain their initial object references. References are required for the root POA, POA Current, the Interface Repository and various Object Service instances. (The POA is described in Chapter 8 of this manual; the Interface Repository is described in Chapter 6 of this manual; Object Services are described in CORBAservices.)

Change the single paragraph sentence that says "Currently, reserved **ObjectIds** are **InterfaceRepository** and **NameService**." to

> Currently, reserved **ObjectId**s are **RootPOA**, **POACurrent**, **InterfaceRepository**, **NameService**, **TradingService**, **SecurityCurrent**, and **TransactionCurrent**.

2.6.4 Add all new 7.7, Thread-related operations

To support single-threaded ORBs, as well as multi-threaded ORBs that run multi-thread-unaware code, several operations are included in the ORB interface. These operations can be used by single-threaded and multi-threaded applications. An application that is a pure ORB client would not need to use these operations. Both the ORB::run() and ORB::shutdown() are useful in fully multi-threaded programs.

Note – These operations are defined on the ORB rather than on an object adapter to allow the main thread to be used for all kinds of asynchronous processing by the ORB. Defining these operations on the ORB also allows the ORB to support multiple object adapters, without requiring the application main to know about all the object adapters. The interface between the ORB and an object adapter is not standardized.

```
module CORBA
{
        …
    interface ORB {
        ...
        boolean work_pending( );
        void perform_work();
        void shutdown( in boolean wait_for_completion );
        void run();
```

7.7.1 work_pending

boolean work_pending( );

This operation returns an indication of whether the ORB needs the main thread to perform some work.

A result of TRUE indicates that the ORB needs the main thread to perform some work and a result of FALSE indicates that the ORB does not need the main thread.

7.7.2 perform_work

void perform_work();

If called by the main thread, this operation performs an implementation-defined unit of work. Otherwise, it does nothing.

It is platform specific how the application and ORB arrange to use compatible threading primitives.

The **work_pending()** and **perform_work()** operations can be used to write a simple polling loop that multiplexes the main thread among the ORB and other activities. Such a loop would most likely be needed in a single-threaded server. A multi-threaded server would need a polling loop only if there were both ORB and other code that required use of the main thread.

Here is an example of such a polling loop:

```
// C++
for (;;) {
   if (orb->work_pending()) {
      orb->perform_work();
   }
   // do other things
   // sleep?
}
```

7.7.3 run

void run();

This operation returns when the ORB has shut down. If called by the main thread, it enables the ORB to perform work using the main thread. Otherwise, it simply waits until the ORB has shut down.

This operation can be used instead of **perform_work()** to give the main thread to the ORB if there are no other activities that need to share the main thread. Even in a pure multi-threaded server, calling **run()** in the main thread is useful to ensure that the process does not exit until the ORB has been shut down.

7.7.4 shutdown

void shutdown(in boolean wait_for_completion);

This operation instructs the ORB to shut down. Shutting down the ORB causes all object adapters to be shut down. If the **wait_for_completion** parameter is TRUE, this operation blocks until all ORB processing (including request processing and object deactivation or other operations associated with object adapters) has completed.

2.6.5 Add all new 7.8, Current Object

[Editor's note: the changes made by the Transaction and Security services have not yet been published as part of the CORBA document. The text below is written with underlines (changes) as if those changes had indeed been made, to show what the Portability specification is changing.]

7.8 Current Object

ORB and CORBA services may wish to provide access to information (context) associated with the thread of execution in which they are running. This information is accessed in a structured manner using interfaces derived from the **Current** interface defined in the **CORBA** module.

Each ORB or CORBA service that needs its own context derives an interface from the CORBA module's **Current**. Users of the service can obtain an instance of the appropriate **Current** interface by making a call on the **ORB** operation **resolve_initial_references**. A CORBA service does not have to use this method of keeping context but may choose to do so.

```
module CORBA
{
    // interface for the Current object and its service identifiers
    interface Current { };
    ...
    interface ORB {
        ...
        //deprecate get_current operation
    };
};
```

Operations on interfaces derived from **Current** access state associated with the thread in which they are invoked, not state associated with the thread from which the **Current** was obtained. This prevents one thread from manipulating another thread's state, and avoids the need to obtain and narrow a new **Current** in each method's thread context.

Current objects must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a **MARSHAL** system exception. **Current**s are per-process singleton objects, so no destroy operation is needed.

Note – The **CORBA::ORB::get_current** operation described in the CORBA Security specification is deprecated and replaced by use of **CORBA::ORB::resolve_initial_references**. The **get_current** operation did not provide a means to specify which derived **Current** to return, and therefore did not allow ORB and CORBA services to be independently implemented.

[Editors Note: Remove all reference to get_current from the next major revision of CORBA.]

2.6.6 Add all new 7.9, Policy Object

[Editor's note: the changes made by the Security service have not yet been published as part of the CORBA document. The text below is written with underlines (changes) as if those changes had indeed been made, to show what the Portability specification is changing.]

7.9 Policy Object

An ORB or CORBA service may choose to allow access to certain choices that affect its operation. This information is accessed in a structured manner using interfaces derived from the **Policy** interface defined in the **CORBA** module. A CORBA service does not have to use this method of accessing operating options but may choose to do so.

```
// basic Policy definition
interface Policy
{
    Policy copy( );
    void destroy( );
};

typedef sequence <Policy>  PolicyList;
```

7.9.1 Copy

Policy copy( );

This operation copies the policy object. The copy does not retain any relationships that the policy had with any domain, or object.

7.9.2 Destroy

void destroy( );

This operation destroys the policy object. It is the responsibility of the policy object to determine whether it can be destroyed.

2.7 Ch 8: BOA

Delete all of Chapter 8 and replace with this submission's Chapter entitled: "The Portable Object Adapter".

2.8 New chapter: for Dynamic Any

Insert a new chapter from this submission's Chapter 4, "Dynamic management of Any values".

2.9 Ch 14: C language mapping

Replace the C language mapping with Chapter 14, "C Language Mapping", from this submission.

2.10 Ch 16-18: C++ language mapping

Replace the C++ language mapping chapters with the following parts of this submission:

- Chapter 16, "Mapping of OMG IDL to C++"
- Chapter 17, "Mapping of Pseudo Objects to C++"
- Chapter 18, "Server-Side Mapping"
- Appendix C, "C++ Definitions for CORBA"
- Appendix D, "Alternative Mappings for C++ Dialects"
- Appendix G, "C++ Keywords'

2.11 Changes to Transaction Service

The following changes need to be made to the Transaction Service to make its use of **Current** consistent with **CORBA::Current**. The changes are to be made in OMG Document Number ptc/96-11-02, all on Page 10-19.

Fourth line, change "...using a **get_current** operation..." to "...using a **resolve_initial_references("TransactionCurrent")** operation...".

Change the first line of the OMG IDL in the box from

```
interface Current : CORBA::ORB::Current {
```

to

```
interface Current : CORBA::Current {
```

Add the following note after the box containing the IDL for **Current**.

Note: In order to pass the transaction from one thread to another, a program should not use the **Current** object. It should pass the **Control** object to the other thread.

2.12 Changes to Security Service

The following **CORBA::Current**-related changes need to be made to the Security Service specification.

In OMG document adopted/97-02-20, Section 15.4.5, page 15-56, subheading “Current”, add the following note at the end of the subsection.

Note: In order to pass credentials from one thread to another, a program should not use the Current object. It should pass the Control object to the other thread.

In OMG document adopted/97-02-20, Section 15.5.6, fourth line, change
“**ORB::get_current()**” to
“**ORB::resolve_initial_references("SecurityCurrent")**”

In OMG document adopted/97-02-21, replace Section B3 with the following.

B.3 Extension to the Use of Current

The Transaction Service introduced a **Current** interface to allow an application to demarcate and manage the transaction associated with the current thread of activity (that is, the execution context of the client or target object).

This specification generalizes this use of **Current** so it can be used to handle other information associated with the execution context at both client and target objects. In particular, it associates security information, such as credentials, with **Current** and provides means for accessing it.

For security, there are two new interfaces: **SecurityLevel1::Current** and **SecurityLevel2::Current**, derived from the **CORBA::Current** interface. The OMG IDL for these are presented in Appendix A, Consolidated OMG IDL.

The mechanism for obtaining a reference to the **Current** object is provided by the **resolve_initial_references** operation of the ORB. The reference returned by **resolve_initial_references("SecurityCurrent")** can be narrowed to **SecurityLevel1::Current** in an ORB that provides Security Functionality Level 1, and also to **SecurityLevel2::Current** in an ORB that provice Security Functionality Level 2.

2.13 Changes to Cobol Mapping

Replace the previously approved Cobol Mapping with that from Chapter 5, “Cobol Mapping”.

The Portable Object Adapter 3

This chapter describes the Portable Object Adapter, or POA. It presents the design goals, a description of the abstract model of the POA and its interfaces, followed by a detailed description of the interfaces themselves.

3.1 Design Goals

The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities. More precisely, the POA is designed to allow programmers to build object implementations that can provide consistent service for objects whose lifetimes (from the perspective of a client holding a reference for such an object) span multiple server lifetimes.
- Provide support for transparent activation of objects.
- Allow a single servant to support multiple object identities simultaneously.
- Allow multiple distinct instances of the POA to exist in a server.
- Provide support for transient objects with minimal programming effort and overhead.
- Provide support for implicit activation of servants with POA-allocated Object Ids.
- Allow object implementations to be maximally responsible for an object's behavior. Specifically, an implementation can control an object's behavior by establishing the datum that defines an object's identity, determining the relationship between the object's identity and the object's state, managing the storage and retrieval of the object's state, providing the code that will be executed in response to requests, and determining whether or not the object exists at any point in time.

- Avoid requiring the ORB to maintain persistent state describing individual objects, their identities, where their state is stored, whether certain identity values have been previously used or not, whether an object has ceased to exist or not, and so on.
- Provide an extensible mechanism for associating policy information with objects implemented in the POA.
- Allow programmers to construct object implementations that inherit from static skeleton classes, generated by OMG IDL compilers, or a DSI implementation.

3.2 Abstract Model Description

The POA interfaces described in this chapter imply a particular abstract computational model. This section presents that model and defines terminology and basic concepts that will be used in subsequent sections.

This section provides the rationale for the POA design, describes some of its intended uses, and provides a background for understanding the interface descriptions.

3.2.1 Model Components

The model supported by the POA is a specialization of the general object model described in the OMA guide. Most of the elements of the CORBA object model are present in the model described here, but there are some new components, and some of the names of existing components are defined more precisely than they are in the CORBA object model. The abstract model supported by the POA has the following components:

- *Client*—A client is a computational context that makes requests on an object through one of its references.
- *Server*—A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process. Note that *client* and *server* are roles that programs play with respect to a given object. A program that is a client for one object may be the server for another. The same process may be both client and server for a single object.
- *Object*—In this discussion, we use *object* to indicate a CORBA object in the abstract sense, that is, a programming entity with an identity, an interface, and an implementation. From a client's perspective, the object's identity is encapsulated in the object's reference. This specification defines the server's view of object identity, which is explicitly managed by object implementations through the POA interface.
- *Servant*—A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made on an object's references are mediated by the ORB and transformed into invocations on a particular servant. In the course of an object's lifetime it may be associated with (that is, requests on its references will be targeted at) multiple servants.

- *Object Id*—An Object Id is a value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients, encapsulated by references. Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences.

Note – The Object Id defined in this specification is a mechanical device used by an object implementation to correlate incoming requests with references it has previously created and exposed to clients. It does not constitute a unique logical identity for an object in any larger sense. The assignment and interpretation of Object Id values is primarily the responsibility of the application developer, although the **SYSTEM_ID** policy enables the POA to generate Object Id values for the application.

- *Object Reference*—An object reference in this model is the same as in the CORBA object model. This model implies, however, that a reference specifically encapsulates an Object Id and a POA identity.

Note – A concrete reference in a specific ORB implementation will contain more information, such as the location of the server and POA in question. For example, it might contain the full name of the POA (the names of all POAs starting from the root and ending with the specific POA). The reference might not, in fact, actually contain the Object Id, but instead contain more compact values managed by the ORB which can be mapped to the Object Id. This is a description of the abstract information model implied by the POA. Whatever encoding is used to represent the POA name and the Object Id must not restrict the ability to use any legal character in a POA name or any legal octet in an Object Id.

- *POA*—A POA is an identifiable entity within the context of a server. Each POA provides a namespace for Object Ids and a namespace for other (nested or child) POAs. Policies associated with a POA describe characteristics of the objects implemented in that POA. Nested POAs form a hierarchical name space for objects within a server.
- *Policy*—A Policy is an object associated with a POA by an application in order to specify a characteristic shared by the objects implemented in that POA. This specification defines policies controlling the POA's threading model as well as a variety of other options related to the management of objects. Other specifications may define other policies that affect how an ORB processes requests on objects implemented in the POA.
- *POA Manager*—A POA manager is an object that encapsulates the processing state of one or more POAs. Using operations on a POA manager, the developer can cause requests for the associated POAs to be queued or discarded. The developer can also use the POA manager to deactivate the POAs.
- *Servant Manager*—A servant manager is an object that the application developer can associate with a POA. The ORB will invoke operations on servant managers to activate servants on demand, and to deactivate servants. Servant managers are responsible for managing the association of an object (as characterized by its Object

Id value) with a particular servant, and for determining whether an object exists or not. There are two kinds of servant managers, called **ServantActivator** and **ServantLocator**; the type used in a particular situation depends on policies in the POA.

- *Adapter Activator*—An adapter activator is an object that the application developer can associate with a POA. The ORB will invoke an operation on an adapter activator when a request is received for a child POA that does not currently exist. The adapter activator can then create the required POA on demand.

3.2.2 Model Architecture

This section describes the architecture of the abstract model implied by the POA, and the interactions between various components. Figure 3-1 illustrates this architecture. The ORB is an abstraction visible to both the client and server. The POA is an object visible to the server. User-supplied implementations are registered with the POA (this statement is a simplification; more detail is provided below). Clients hold references upon which they can make requests. The ORB, POA, and implementation all cooperate to determine which servant the operation should be invoked on, and to perform the invocation.

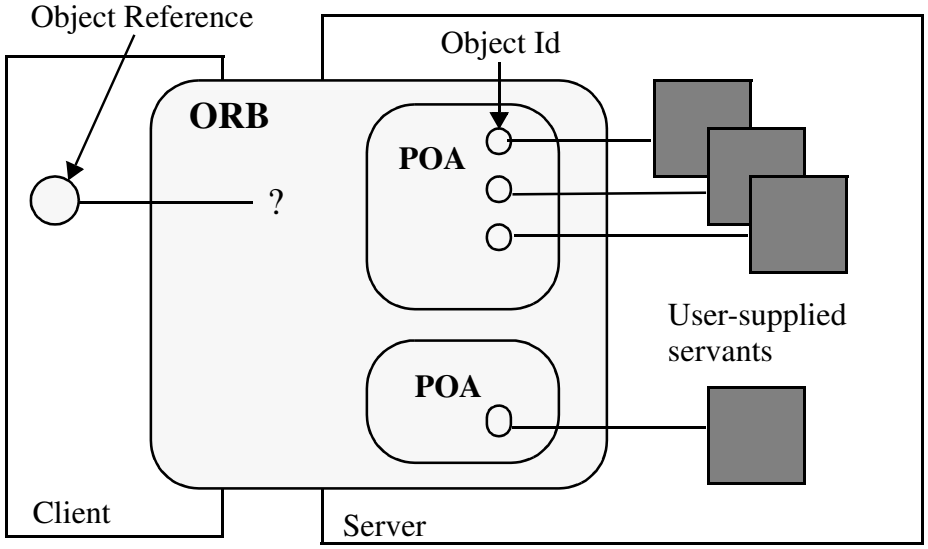

Figure 3-1 Abstract POA model

Figure 3-2 shows the detail of the relationship between the POA and the implementation. Ultimately, a POA deals with an Object Id and an active servant. By *active servant*, we mean a programming object that exists in memory and has been presented to the POA with one or more associated object identities. There are several ways for this association to be made.

If the POA supports the **RETAIN** policy, it maintains a map, labeled *Active Object Map*, that associates Object Ids with active servants, each association constituting an active object. If the POA has the **USE_DEFAULT_SERVANT** policy, a default servant may be registered with the POA. Alternatively, if the POA has the **USE_SERVANT_MANAGER** policy, a user-written servant manager may be registered with the POA. If the Active Object Map is not used, or a request arrives for an object not present in the Active Object Map, the POA either uses the default servant to perform the request or it invokes the servant manager to obtain a servant to perform the request. If the **RETAIN** policy is used, the servant returned by a servant manager is retained in the Active Object Map. Otherwise, the servant is used only to process the one request.

In this specification, the term *active* is applied equally to servants, Object Ids, and objects. An object is active in a POA if the POA's Active Object Map contains an entry that associates an Object Id with an existing servant. When this specification refers to *active Object Ids* and *active servants*, it means that the Object Id value or servant in question is part of an entry in the Active Object Map.

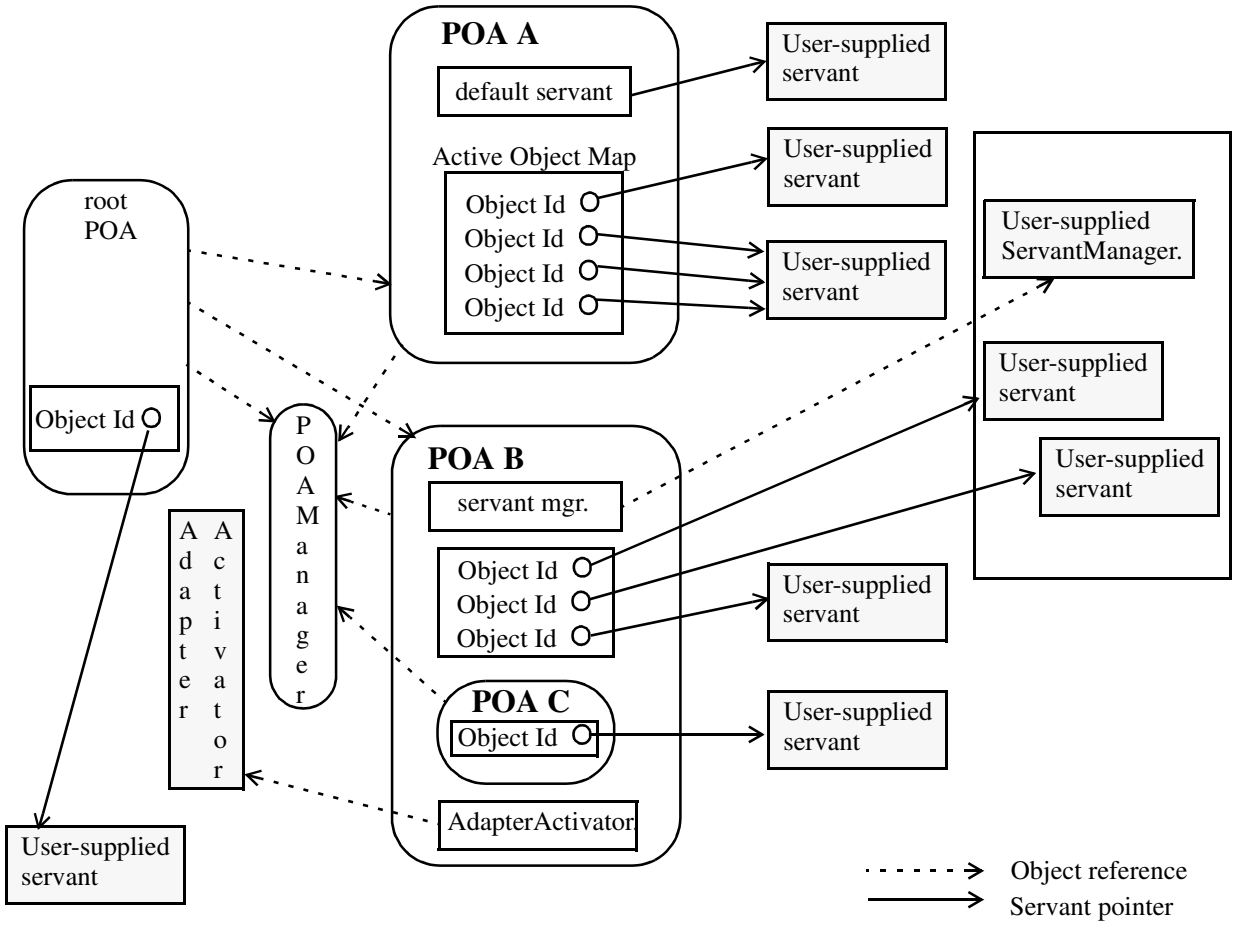

Figure 3-2 POA Architecture

3.2.3 POA Creation

To implement an object using the POA requires that the server application obtain a POA object. A distinguished POA object, called the *root POA*, is managed by the ORB and provided to the application using the ORB initialization interface under the initial object name “RootPOA”. The application developer can create objects using the root POA if those default policies are suitable. The root POA has the following policies.

- Thread Policy: **ORB_CTRL_MODEL**
- Lifespan Policy: **TRANSIENT**
- Object Id Uniqueness Policy: **UNIQUE_ID**
- Id Assignment Policy: **SYSTEM_ID**
- Servant Retention Policy: **RETAIN**
- Request Processing Policy: **USE_ACTIVE_OBJECT_MAP_ONLY**
- Implicit Activation Policy: **IMPLICIT_ACTIVATION**

The developer can also create new POAs. Creating a new POA allows the application developer to declare specific policy choices for the new POA and to provide a different adapter activator and servant manager (these are callback objects used by the POA to activate objects and nested POAs on demand). Creating new POAs also allows the application developer to partition the name space of objects, as Object Ids are interpreted relative to a POA. Finally, by creating new POAs, the developer can independently control request processing for multiple sets of objects.

A POA is created as a child of an existing POA using the **create_POA** operation on the parent POA. When a POA is created, the POA is given a name that must be unique with respect to all other POAs with the same parent.

POA objects are not persistent. No POA state can be assumed to be saved by the ORB. It is the responsibility of the server application to create and initialize the appropriate POA objects during server initialization or to set an AdapterActivater to create POA objects needed later.

Creating the appropriate POA objects is particularly important for persistent objects, objects whose existence can span multiple server lifetimes. To support an object reference created in a previous server process, the application must recreate the POA that created the object reference as well as all of its ancestor POAs. To ensure portability, each POA must be created with the same name as the corresponding POA in the original server process and with the same policies. (It is the user’s responsibility to create the POA with these conditions.)

A portable server application can presume that there is no conflict between its POA names and the POA names chosen by other applications. It is the responsibility of the ORB implementation to provide a way to support this behavior.

3.2.4 Reference Creation

Object references are created in servers. Once they are created, they may be exported to clients.

From this model's perspective, object references encapsulate object identity information and information required by the ORB to identify and locate the server and POA with which the object is associated (that is, in whose scope the reference was created.) References are created in the following ways:

- The server application may directly create a reference with the **create_reference** and **create_reference_with_id** operations on a POA object. These operations collect the necessary information to constitute the reference, either from information associated with the POA or as parameters to the operation. These operations only create a reference. In doing so, they bring the abstract object into existence, but do not associate it with an active servant.
- The server application may explicitly activate a servant, associating it with an object identity using the **activate_object** or **activate_object_with_id** operations. Once a servant is activated, the server application can map the servant to its corresponding reference using the **servant_to_reference** or **id_to_reference** operations.
- The server application may cause a servant to implicitly activate itself. This behavior can only occur if the POA has been created with the **IMPLICIT_ACTIVATION** policy. If an attempt is made to obtain an object reference corresponding to an inactive servant, the POA may automatically assign a generated unique Object Id to the servant and activate the resulting object. The reference may be obtained by invoking **POA::servant_to_reference** with an inactive servant, or by performing an explicit or implicit type conversion from the servant to a reference type in programming language mappings that permit this conversion.

Once a reference is created in the server, it can be made available to clients in a variety of ways. It can be advertised through the OMG Naming and Trading Services. It can be converted to a string via **ORB::object_to_string** and published in some way that allows the client to discover the string and convert it to a reference using **ORB::string_to_object**. It can be returned as the result of an operation invocation.

Once a reference becomes available to a client, that reference constitutes the identity of the object from the client's perspective. As long as the client program holds and uses that reference, requests made on the reference should be sent to the "same" object.

Note – It should be noted here that the meaning of object identity and "sameness" is at present the subject of heated debate in the OMG. This specification does not attempt to resolve that debate in any way, particularly by defining a concrete notion of identity that is exposed to clients, beyond the existing notions of identity described in the CORBA specifications and the OMA guide.

The states of servers and implementation objects are opaque to clients. This specification deals primarily with the view of the ORB from the server's perspective.

3.2.5 Object Activation States

At any point in time, a CORBA object may or may not be associated with an active servant.

If the POA has the **RETAIN** policy, the servant and its associated Object Id are entered into the Active Object Map of the appropriate POA. This type of activation can be accomplished in one of the following ways.

- The server application itself explicitly activates individual objects (via the **activate_object** or **activate_object_with_id** operations).
- The server application instructs the POA to activate objects on demand by having the POA invoke a user-supplied servant manager. The server application registers this servant manager with **set_servant_manager**.
- Under some circumstances (when the **IMPLICIT_ACTIVATION** policy is also in effect and the language binding allows such an operation), the POA may implicitly activate an object when the server application attempts to obtain a reference for a servant that is not already active (that is, not associated with an Object Id).

If the **USE_DEFAULT_SERVANT** policy is also in effect, the server application instructs the POA to activate unknown objects by having the POA invoke a single servant no matter what the Object Id is. The server application registers this servant with **set_servant**.

If the POA has the **NON_RETAIN** policy, for every request, the POA may use either a default servant or a servant manager to locate an active servant. From the POA's point of view, the servant is active only for the duration of that one request. The POA does not enter the servant-object association into the Active Object Map.

3.2.6 Request Processing

A request must be capable of conveying the Object Id of the target object as well as the identification of the POA that created the target object reference. When a client issues a request, the ORB first locates an appropriate server (perhaps starting one if needed) and then it locates the appropriate POA within that server.

If the POA does not exist in the server process, the application has the opportunity to re-create the required POA by using an adapter activator. An adapter activator is a user-implemented object that can be associated with a POA. It is invoked by the ORB when a request is received for a non-existent child POA. The adapter activator has the opportunity to create the required POA. If it does not, the client receives the **OBJECT_NOT_EXIST** exception.

Once the ORB has located the appropriate POA, it delivers the request to that POA. The further processing of that request depends both upon the policies associated with that POA as well as the object's current state of activation.

If the POA has the **RETAIN** policy, the POA looks in the Active Object Map to find if there is a servant associated with the Object Id value from the request. If such a servant exists, the POA invokes the appropriate method on the servant.

If the POA has the **NON_RETAIN** policy or has the **RETAIN** policy but didn't find a servant in the Active Object Map, the POA takes the following actions:

- If the POA has the **USE_DEFAULT_SERVANT** policy, a default servant has been associated with the POA so the POA will invoke the appropriate method on that servant. If no servant has been associated with the POA, the POA raises the **OBJ_ADAPTER** system exception.
- If the POA has the **USE_SERVANT_MANAGER** policy, a servant manager has been associated with the POA so the POA will invoke **incarnate** or **preinvoke** on it to find a servant that may handle the request. (The choice of method depends on the **NON_RETAIN** or **RETAIN** policy of the POA.) If no servant manager has been associated with the POA, the POA raises the **OBJ_ADAPTER** system exception.
- If the **USE_OBJECT_MAP_ONLY** policy is in effect, the POA raises the **OBJECT_NOT_EXIST** system exception.

If a servant manager is located and invoked, but the servant manager is not directly capable of incarnating the object, it (the servant manager) may deal with the circumstance in a variety of ways, all of which are the application's responsibility. Any system exception raised by the servant manager will be returned to the client in the reply. In addition to standard CORBA exceptions, a servant manager is capable of raising a **ForwardRequest** exception. This exception includes an object reference. The ORB will process this exception as stated below.

3.2.7 Implicit Activation

A POA can be created with a policy that indicates that its objects may be implicitly activated. This policy, **IMPLICIT_ACTIVATION**, also requires the **SYSTEM_ID** and **RETAIN** policies. When a POA supports implicit activation, an inactive servant may be implicitly activated in that POA by certain operations that logically require an Object Id to be assigned to that servant. Implicit activation of an object involves allocating a system-generated Object Id and registering the servant with that Object Id in the Active Object Map. The interface associated with the implicitly activated object is determined from the servant (using static information from the skeleton, or, in the case of a dynamic servant, using the **_primary_interface()** operation).

The operations that support implicit activation include:

- The **POA::servant_to_reference** operation, which takes a servant parameter and returns a reference.
- The **POA::servant_to_id** operation, which takes a servant parameter and returns an Object Id.
- Operations supported by a language mapping to obtain an object reference or an Object Id for a servant. For example, the **`_this()`** servant member function in C++ returns an object reference for the servant.
- Implicit conversions supported by a language mapping that convert a servant to an object reference or an Object Id.

The last two categories of operations are language mapping dependent.

If the POA has the **UNIQUE_ID** policy, then implicit activation will occur when any of these operations are performed on a servant that is not currently active (that is, it is associated with no Object Id in the POA's Active Object Map).

If the POA has the **MULTIPLE_ID** policy, the **servant_to_reference** and **servant_to_id** operations will *always* perform implicit activation, even if the servant is already associated with an Object Id. The behavior of language mapping operations in the **MULTIPLE_ID** case is specified by the language mapping. For example, in C++, the **`_this()`** servant member function will not implicitly activate a **MULTIPLE_ID** servant if the invocation of **`_this()`** is immediately within the dynamic context of a request invocation directed by the POA to that servant; instead, it returns the object reference used to issue the request.

Note – The exact timing of implicit activation is ORB implementation dependent. For example, instead of activating the object immediately upon creation of a local object reference, the ORB could defer the activation until the Object Id is actually needed (for example, when the object reference is exported outside the process).

3.2.8 Multi-threading

The POA does not require the use of threads and does not specify what support is needed from a threads package. However, in order to allow the development of portable servers that utilize threads, the behavior of the POA and related interfaces when used within a multiple thread environment must be specified.

Specifying this behavior does not require that an ORB must support being used in a threaded environment, nor does it require that an ORB must utilize threads in the processing of requests. The only requirement given here is that if an ORB does provide support for multi-threading, these are the behaviors that will be supported by that ORB. This allows a programmer to take advantage of multiple ORBs that support threads in a portable manner across those ORBs.

The POA's processing is affected by the thread-related calls available in the ORB: **work_pending**, **perform_work**, **run**, and **shutdown**.

3.2.8.1 POA Threading Models

The POA supports two models of threading when used in conjunction with multi-threaded ORB implementations; ORB controlled and single thread behavior. The two models can be used together or independently. Either model can be used in environments where a single-threaded ORB is used.

The threading model associated with a POA is indicated when the POA is created by including a **ThreadPolicy** object in the policies parameter of the POA's **create_POA** operation. Once a POA is created with one model, it cannot be changed to the other. All uses of the POA within the server must conform to that threading model associated with the POA.

3.2.8.2 Using the Single Thread Model

Requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all upcalls made by this POA to implementation code (servants, servant managers, and adapter activators) are made in a manner that is safe for code that is multi-thread-unaware.

3.2.8.3 Using the ORB Controlled Model

The ORB controlled model of threading is used in environments where the developer wants the ORB/POA to control the use of threads in the manner provided by the ORB. This model can also be used in environments that do not support threads.

In this model, the ORB is responsible for the creation, management, and destruction of threads used with one or more POAs.

3.2.8.4 Limitations When Using Multiple Threads

There are no guarantees that the ORB and POA will do anything specific about dispatching requests across threads with a single POA. Therefore, a server programmer who wants to use one or more POAs within multiple threads must take on all of the serialization of access to objects within those threads.

There may be requests active for the same object being dispatched within multiple threads at the same time. The programmer must be aware of this possibility and code with it in mind.

3.2.9 Dynamic Skeleton Interface

The POA is designed to enable programmers to connect servants to:

- type-specific skeletons, typically generated by OMG IDL compilers; or
- dynamic skeletons

Servants that are members of type-specific skeleton classes are referred to as type-specific servants. Servants connected to dynamic skeletons are used to implement the Dynamic Skeleton Interface (DSI) and are referred to as DSI servants.

Whether a CORBA object is being incarnated by a DSI servant or a type-specific servant is transparent to its clients. Two CORBA objects supporting the same interface may be incarnated one by a DSI servant and the other with a type-specific servant. Furthermore, a CORBA object may be incarnated by a DSI servant only during some period of time, while the rest of the time is incarnated by a static servant.

The mapping for POA DSI servants is language specific, with each language providing a set of interfaces to the POA. These interfaces are used only by the POA. The interfaces required are the following.

- Take a **CORBA::ServerRequest** object from the POA and perform the processing necessary to execute the request.

- Return the Interface Repository Id identifying the most-derived interface supported by the target CORBA object in a request.

The reason for the first interface is the entire reason for existence of the DSI: to be able to handle any request in the way the programmer wishes to handle it. A single DSI servant may be used to incarnate several CORBA objects, potentially supporting different interfaces.

The reason for the second interface can be understood by comparing DSI servants to type-specific servants.

A type-specific servant may incarnate several CORBA objects but all of them will support the same IDL interface as the most-derived IDL interface. In C++, for example, an IDL interface **`Window`** in module **`GraphicalSystem`** will generate a type-specific skeleton class called **`Window`** in namespace **`POA_GraphicalSystem`**. A type-specific servant which is directly derived from the **`POA_GraphicalSystem::Window`** skeleton class may incarnate several CORBA objects at a time, but all those CORBA objects will support the **`GraphicalSystem::Window`** interface as the most-derived interface.

A DSI servant may incarnate several CORBA objects, not necessarily supporting the same IDL interface as the most-derived IDL interface.

In both cases (type-specific and DSI) the POA may need to determine, at runtime, the Interface Repository Id identifying the most-derived interface supported by the target CORBA object in a request. The POA should be able to determine this by asking the servant that is going to serve the CORBA object.

In the case of type-specific servants, the POA obtains that information from the type-specific skeleton class from which the servant is a directly derived. In the case of DSI servants, the POA obtains that information by using the second language-specific interface above.

3.2.10 Location Transparency

The POA supports location transparency for objects implemented using the POA. Unless explicitly stated to the contrary, all POA behavior described in this specification applies regardless of whether the client is local (same process) or remote. For example, like a request from a remote client, a request from a local client may: cause object activation if the object is not active; may block indefinitely if the target object's POA is in the holding state; may be rejected if the target object's POA is in the discarding or inactive states; may be delivered to a thread-unaware object implementation; or may be delivered to a different object if the target object's servant manager raises the **ForwardRequest** exception. The Object Id and POA of the target object will also be available to the server via the **Current** object, regardless of whether the client is local or remote.

Note – The implication of these requirements on the ORB implementation is to require the ORB to mediate all requests to POA based objects, even if the client is co-resident in the same process. This specification is not intended to change CORBAServices

specifications that allow for behaviors that are not location transparent. This specification does not prohibit (nonstandard) POA extensions to support object behavior that is not location transparent.

3.3 Interfaces

The POA-related interfaces are defined in a module separate from the **CORBA** module, the **PortableServer** module. It consists of several interfaces:

- **POA**
- **POAManager**
- **ServantManager**
- **ServantActivator**
- **ServantLocator**
- **AdapterActivator**
- **ThreadPolicy**
- **LifespanPolicy**
- **IdUniquenessPolicy**
- **IdAssignmentPolicy**
- **ImplicitActivationPolicy**
- **ServantRetentionPolicy**
- **RequestProcessingPolicy**
- **Current**

In addition, the POA defines the **Servant** native type.

3.3.1 The Servant IDL Type

This specification defines a native type **PortableServer::Servant**. Values of the type **Servant** are programming-language-specific implementations of CORBA interfaces. Each language mapping must specify how **Servant** is mapped to the programming language data type that corresponds to an object implementation. The **Servant** type has the following characteristics and constraints.

- Values of type **Servant** are opaque from the perspective of CORBA application programmers. There are no operations that can be performed directly on them by user programs. They can be passed as parameters to certain POA operations. Some language mappings may allow **Servant** values to be implicitly converted to object references under appropriate conditions.
- Values of type **Servant** support a language-specific programming interface that can be used by the ORB to obtain a default POA for that servant. This interface is used only to support implicit activation. A language mapping may provide a default implementation of this interface that returns the root POA of a default ORB.
- Values of type **Servant** must be testable for identity.

- Values of type **Servant** have no meaning outside of the process context or address space in which they are generated.

3.3.2 POAManager Interface

Each POA object has an associated **POAManager** object. A POA manager may be associated with one or more POA objects. A POA manager encapsulates the processing state of the POAs it is associated with. Using operations on the POA manager, an application can cause requests for those POAs to be queued or discarded, and can cause the POAs to be deactivated.

POA managers are created and destroyed implicitly. Unless an explicit POA manager object is provided at POA creation time, a POA manager is created when a POA is created and is automatically associated with that POA. A POA manager object is implicitly destroyed when all of its associated POAs have been destroyed.

3.3.2.1 Processing States

A POA manager has four possible processing states; *active*, *inactive*, *holding*, and *discarding*. The processing state determines the capabilities of the associated POAs and the disposition of requests received by those POAs. Figure 3-1 illustrates the processing states and the transitions between them. For simplicity of presentation, this specification sometimes describes these states as POA states, referring to the POA or POAs that have been associated with a particular POA manager. A POA manager is created in the *holding* state. The root POA is therefore initially in the *holding* state.

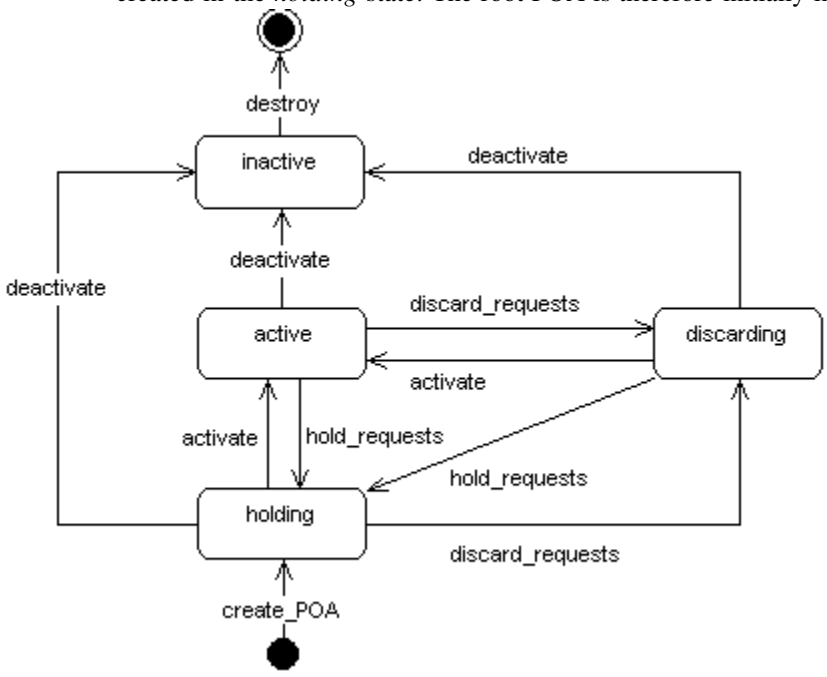

Figure 3-3 Processing States

3.3.2.1.1 Active State

When a POA manager is in the *active* state, the associated POAs will receive and start processing requests (assuming that appropriate thread resources are available). Note that even in the active state, a POA may need to queue requests depending upon the ORB implementation and resource limits. The number of requests that can be received and/or queued is an implementation limit. If this limit is reached, the POA should return a **TRANSIENT** system exception to indicate that the client should re-issue the request.

A user program can legally transition a POA manager from the *active* state to either the *discarding*, *holding*, or *inactive* state by calling the **discard_requests**, **hold_requests**, or **deactivate** operations, respectively. The POA enters the *active* state through the use of the **activate** operation when in the *discarding* or *holding* state.

3.3.2.1.2 Discarding State

When a POA manager is in the *discarding* state, the associated POAs will discard all incoming requests (whose processing has not yet begun). When a request is discarded, the **TRANSIENT** system exception must be returned to the client-side to indicate that the request should be re-issued. (Of course, an ORB may always reject a request for other reasons and raise some other system exception.)

In addition, when a POA manager is in the *discarding* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be discarded, as described in the previous paragraph.

The primary purpose of the *discarding* state is to provide an application with flow-control capabilities when it determines that an object's implementation or POA is being flooded with requests. It is expected that the application will restore the POA manager to the *active* state after correcting the problem that caused flow-control to be needed.

A POA manager can legally transition from the *discarding* state to either the *active*, *holding*, or *inactive* state by calling the **activate**, **hold_requests**, or **deactivate** operations, respectively. The POA enters the *discarding* state through the use of the **discard_requests** operation when in the *active* or *holding* state.

3.3.2.1.3 Holding State

When a POA manager is in the *holding* state, the associated POAs will queue incoming requests. The number of requests that can be queued is an implementation limit. If this limit is reached, the POAs may discard requests and return the **TRANSIENT** system exception to the client to indicate that the client should reissue the request. (Of course, an ORB may always reject a request for other reasons and raise some other system exception.)

In addition, when a POA manager is in the *holding* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be queued, as described in the previous paragraph.

A POA manager can legally transition from the *holding* state to either the *active*, *discarding*, or *inactive* state by calling the **activate**, **discard_requests**, or **deactivate** operations, respectively. The POA enters the *holding* state through the use of the **hold_requests** operation when in the *active* or *discarding* state. A POA manager is created in the holding state.

3.3.2.1.4 Inactive State

The *inactive* state is entered when the associated POAs are to be shut down. Unlike the *discarding* state, the *inactive* state is not a temporary state. When a POA manager is in the *inactive* state, the associatied POAs will reject new requests. The rejection mechanism used is specific to the vendor. The GIOP location forwarding mechanism and CloseConnection message are examples of mechanisms that could be used to indicate the rejection. If the client is co-resident in the same process, the ORB could raise the **OBJ_ADAPTER** exception to indicate that the object implementation is unavailable

In addition, when a POA manager is in the *inactive* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be rejected, as described in the previous paragraph.

The *inactive* state is entered using the **deactivate** operation. It is legal to enter the *inactive* state from either the *active*, *holding*, or *discarding* states.

If the transition into the *inactive* state is a result of calling **deactivate** with an **etherealize_objects** parameter of

- TRUE - the associated POAs will call **etherealize** for each active object associated with the POA once all currently executing requests have completed processing (if the POAs have the **RETAIN** and **USE_SERVANT_MANAGER** policies). If a servant manager has been registered for the POA, the POA will get rid of the object. If there are any queued requests that have not yet started executing, they will be treated as if they were new requests and rejected.
- FALSE - No deactivations or etherealizations will be attempted.

3.3.2.2 Locality Constraints

A **POAManager** object must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a **MARSHAL** system exception. An attempt to use a **POAManager** object with the DII may raise the **NO_IMPLEMENT** exception.

3.3.2.3 activate

```
void activate()
    raises (AdapterInactive);
```

This operation changes the state of the POA manager to *active*. If issued while the POA manager is in the *inactive* state, the **AdapterInactive** exception is raised. Entering the *active* state enables the associated POAs to process requests.

3.3.2.4 hold_requests

```
void hold_requests( in boolean wait_for_completion )
    raises( AdapterInactive );
```

This operation changes the state of the POA manager to *holding*. If issued while the POA manager is in the *inactive* state, the **AdapterInactive** exception is raised. Entering the *holding* state causes the associated POAs to queue incoming requests. Any requests that have been queued but have not started executing will continue to be queued while in the *holding* state.

If the **wait_for_completion** parameter is FALSE, this operation returns immediately after changing the state. If the parameter is TRUE, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than *holding*.

3.3.2.5 discard_requests

```
void discard_requests( in boolean wait_for_completion )
    raises (AdapterInactive);
```

This operation changes the state of the POA manager to *discarding*. If issued while the POA manager is in the *inactive* state, the **AdapterInactive** exception is raised. Entering the *discarding* state causes the associated POAs to discard incoming requests. In addition, any requests that have been queued but have not started executing are discarded. When a request is discarded, a **TRANSIENT** system exception is returned to the client.

If the **wait_for_completion** parameter is FALSE, this operation returns immediately after changing the state. If the parameter is TRUE, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than *discarding*.

3.3.2.6 deactivate

```
void deactivate(    in boolean etherealize_objects,
                    in boolean wait_for_completion);
    raises (AdapterInactive);
```

This operation changes the state of the POA manager to *inactive*. If issued while the POA manager is in the *inactive* state, the **AdapterInactive** exception is raised. Entering the inactive state causes the associated POAs to reject requests that have not begun to be executed as well as any new requests.

After changing the state, if the **etherealize_objects** parameter is

- TRUE - the POA manager will cause all associated POAs that have the **RETAIN** and **USE_SERVANT_MANAGER** policies to perform the **etherealize** operation on the associated servant manager for all active objects.
- FALSE - the **etherealize** operation is not called. The purpose is to provide developers with a means to shut down POAs in a crisis (for example, unrecoverable error) situation.

If the **wait_for_completion** parameter is FALSE, this operation will return immediately after changing the state. If the parameter is TRUE, this operation does not return until there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) and, in the case of a TRUE **etherealize_objects**, all invocations of **etherealize** have completed for POAs having the **RETAIN** and **USE_SERVANT_MANAGER** policies.

If the **ORB::shutdown** operation is called, it makes a call on **deactivate** with a TRUE **etherealize_objects** parameter for each POA manager known in the process; the **wait_for_completion** parameter to **deactivate** will be the same as the similarly named parameter of **ORB::shutdown**.

3.3.3 AdapterActivator Interface

Adapter activators are associated with POAs. An adapter activator supplies a POA with the ability to create child POAs on demand, as a side-effect of receiving a request that names the child POA (or one of its children), or when **find_POA** is called with an activate parameter value of TRUE. An application server that creates all its needed POAs at the beginning of execution does not need to use or provide an adapter activator; it is necessary only for the case in which POAs need to be created during request processing.

While a request from the POA to an adapter activator is in progress, all requests to objects managed by the new POA (or any descendant POAs) will be queued. This serialization allows the adapter activator to complete any initialization of the new POA before requests are delivered to that POA.

3.3.3.1 Locality Constraints

An **AdapterActivator** object must be local to the process containing the POA objects it is registered with.

3.3.3.2 unknown_adapter

boolean unknown_adapter(in POA parent, in string name);

This operation is invoked when the ORB receives a request for an object reference that identifies a target POA that does not exist. The ORB invokes this operation once for each POA that must be created in order for the target POA to exist (starting with the ancestor POA closest to the root POA). The operation is invoked on the adapter

activator associated with POA that is the parent of the POA that needs to be created. That parent POA is passed as the **parent** parameter. The name of the POA to be created (relative to the parent) is passed as the **name** parameter.

The implementation of this operation should either create the specified POA and return TRUE, or it should return FALSE. If the operation returns TRUE, the ORB will proceed with processing the request. If the operation returns FALSE, the ORB will return **OBJECT_NOT_EXIST** to the client. If multiple POAs need to be created, the ORB will invoke **unknown_adapter** once for each POA that needs to be created. If the parent of a nonexistent POA does not have an associated adapter activator, the ORB will return the **OBJECT_NOT_EXIST** exception.

If **unknown_adapter** raises a system exception, the ORB will report an **OBJ_ADAPTER** exception.

For example, if the target object reference was created by a POA whose full name is "A", "B", "C", "D" and only POAs "A" and "B" currently exist, the **unknown_adapter** operation will be invoked on the adapter activator associated with POA "B" passing POA "B" as the parent parameter and "C" as the name of the missing POA. Assuming that the adapter activator creates POA "C" and returns TRUE, the ORB will then invoke **unknown_adapter** on the adapter activator associated with POA "C", passing POA "C" as the parent parameter and "D" as the name.

The **unknown_adapter** operation is also invoked when **find_POA** is called on the POA with which the **AdapterActivator** is associated, the specified child does not exist, and the **activate_it** parameter to **find_POA** is TRUE. If **unknown_adapter** creates the specified POA and returns TRUE, that POA is returned from **find_POA**.

Note – This allows the same code, the **unknown_adapter** implementation, to be used to initialize a POA whether that POA is created explicitly by the application or as a side-effect of processing a request. Furthermore, it makes this initialization atomic with respect to delivery of requests to the POA.

3.3.4 ServantManager Interface

Servant managers are associated with POAs. A servant manager supplies a POA with the ability to activate objects on demand when the POA receives a request targeted at an inactive object. A servant manager is registered with a POA as a callback object, to be invoked by the POA when necessary. An application server that activates all its needed objects at the beginning of execution does not need to use a servant manager; it is used only for the case in which an object must be activated during request processing.

The **ServantManager** interface is itself empty. It is inherited by two other interfaces, **ServantActivator** and **ServantLocator**.

The two types of servant managers correspond to the POA's **RETAIN** policy (**ServantActivator**) and to the **NON_RETAIN** policy (**ServantLocator**). The meaning of the policies and the operations that are available for POAs using each policy are listed under the two types of derived interfaces.

Each servant manager type contains two operations, the first called to find and return a servant and the second to deactivate a servant. The operations differ according to the amount of information usable for their situation.

3.3.4.1 Common information for servant manager types

The two types of servant managers have certain semantics that are identical.

The **incarnate** and **preinvoke** operation may raise any system exception deemed appropriate (for example, **OBJECT_NOT_EXIST** if the object corresponding to the Object Id value has been destroyed).

Note – If a user-written routine (servant manager or method code) raises the **OBJECT_NOT_EXIST** exception, the POA does nothing but pass on that exception. It is the user's responsibility to deactivate the object if it had been previously activated.

The **incarnate** and **preinvoke** operation may also raise a **ForwardRequest** exception. If this occurs, the ORB is responsible for delivering the current request and subsequent requests to the object denoted in the **forward_reference** member of the exception. The behavior of this mechanism must be the functional equivalent of the GIOP location forwarding mechanism. If the current request was delivered via an implementation of the GIOP protocol (such as IIOP), the reference in the exception should be returned to the client in a reply message with **LOCATION_FORWARD** reply status. If some other protocol or delivery mechanism was used, the ORB is responsible for providing equivalent behavior, from the perspectives of the client and the object denoted by the new reference.

3.3.4.2 Locality Constraints

A **ServantManager** object must be local to the process containing the POA objects it is registered with.

3.3.5 ServantActivator Interface

When the POA has the **RETAIN** policy it uses servant managers that are **ServantActivator**s. When using such servant managers, the following statements apply for a given **ObjectId** used in the **incarnate** and **etherealize** operations:

- Servants incarnated by the servant manager will be placed in the Active Object Map with objects they have activated.
- Invocations of **incarnate** on the servant manager are serialized.
- Invocations of **etherealize** on the servant manager are serialized.
- Invocations of **incarnate** and **etherealize** on the servant manager are mutually exclusive.

- Incarnations of a particular servant may not overlap; that is, if a servant is incarnated by a servant manager, **incarnate** shall not be invoked using that same Object Id until that servant is etherealized.

It should be noted that there may be a period of time between an object's deactivation and the etherealization (during which outstanding requests are being processed) in which arriving requests on that object should not be passed to its servant. During this period, requests targeted for such an object act as if the POA were in *holding* state until **etherealize** completes. If **etherealize** is called as a consequence of a **deactivate** call with a **etherealize_objects** parameter of TRUE, incoming requests are rejected.

It should also be noted that a similar situation occurs with **incarnate**: there may be a period of time after the POA invokes **incarnate** and before that method returns in which arriving requests bound for that object should not be passed to the servant.

A single servant manager object may be concurrently registered with multiple POAs. Invocations of **incarnate** and **etherealize** on a servant manager in the context of different POAs are not necessarily serialized or mutually exclusive. There are no assumptions made about the thread in which **etherealize** is invoked.

3.3.5.1 *incarnate*

```
Servant incarnate (
        in ObjectId             oid,
        in POA                  adapter )
    raises (ForwardRequest);
```

This operation is invoked by the POA whenever the POA receives a request for an object that is not currently active, assuming the POA has the **USE_SERVANT_MANAGER** and **RETAIN** policies.

The **oid** parameter contains the **ObjectId** value associated with the incoming request. The **adapter** is an object reference for the POA in which the object is being activated.

The user-supplied servant manager implementation is responsible for locating or creating an appropriate servant that corresponds to the **ObjectId** value if possible. **incarnate** returns a value of type **Servant**, which is the servant that will be used to process the incoming request (and potentially subsequent requests, since the POA has the **RETAIN** policy).

The POA enters the returned **Servant** value into the Active Object Map so that subsequent requests with the same **ObjectId** value will be delivered directly to that servant without invoking the servant manager.

If the **incarnate** operation returns a servant that is already active for a different Object Id and if the POA also has the **UNIQUE_ID** policy, the **incarnate** has violated the POA policy and is considered to be in error. The POA will raise an **OBJ_ADAPTER** system exception for the request.

Note – If the same servant is used in two different POAs, it is legal for the POAs to use that servant even if the POAs have different Object Id uniqueness policies. The POAs do not interact with each other in this regard.

3.3.5.2 etherealize

```
void etherealize (
        in ObjectId         oid,
        in POA              adapter,
        in Servant          serv,
        in boolean          cleanup_in_progress,
        in boolean          remaining_activations );
```

This operation is invoked whenever a servant for an object is deactivated, assuming the POA has the **USE_SERVANT_MANAGER** and **RETAIN** policies. Note that an active servant may be deactivated by the servant manager via **etherealize** even if it was not incarnated by the servant manager.

The **oid** parameter contains the Object Id value of the object being deactivated. The **adapter** parameter is an object reference for the POA in whose scope the object was active. The **serv** parameter contains a reference to the servant that is associated with the object being deactivated. If the servant denoted by the **serv** parameter is associated with other objects in the POA denoted by the **adapter** parameter (that is, in the POA's Active Object Map) at the time that **etherealize** is called, the **remaining_activations** parameter has the value TRUE. Otherwise, it has the value FALSE.

If the **cleanup_in_progress** parameter is TRUE, the reason for the **etherealize** operation is that either the **deactivate** or **destroy** operation was called with an **etherealize_objects** parameter of TRUE. If the parameter is FALSE, the **etherealize** operation is called for other reasons.

Deactivation occurs in the following circumstances:

- When an object is explicitly deactivated by an invocation of **POA::deactivate_object**.
- When the ORB or POA determines internally that an object must be deactivated. For example, an ORB implementation may provide policies that allow objects to be deactivated after some period of quiescence, or when the number of active objects reaches some limit.
- If **POAManager::deactivate** is invoked on a POA manager associated with a POA that has currently active objects.

Destroying a servant that is in the Active Object Map or is otherwise known to the POA can lead to undefined results.

In a multi-threaded environment, the POA makes certain guarantees that allow servant managers to safely destroy servants. Specifically, the servant's entry in the Active Object Map corresponding to the target object is removed before **etherealize**() is called.

Because calls to **incarnate**() and **etherealize**() are serialized, this prevents new requests for the target object from being invoked on the servant during etherealization. After removing the entry from the Active Object Map, if the POA determines before invoking **etherealize**() that other requests for the same target object are already in progress on the servant, it delays the call to **etherealize**() until all active methods for the target object have completed. Therefore, when **etherealize**() is called, the servant manager can safely destroy the servant if it wants to, unless the **remaining_activations** argument is TRUE.

3.3.6 ServantLocator Interface

When the POA has the **NON_RETAIN** policy it uses servant managers that are **ServantLocator**s. Because the POA knows that the servant returned by this servant manager will be used only for a single request, it can supply extra information to the servant manager's operations and the servant manager's pair of operations may be able to cooperate to do something different than a **ServantActivator**.

When the POA uses the **ServantLocator** interface, immediately after performing the operation invocation on the servant returned by **preinvoke**, the POA will invoke **postinvoke** on the servant manager, passing the **ObjectId** value and the **Servant** value as parameters (among others). The next request with this **ObjectId** value will then cause **preinvoke** to be invoked again. This feature may be used to force every request for objects associated with a POA to be mediated by the servant manager.

When using such a **ServantLocator**, the following statements apply for a given **ObjectId** used in the **preinvoke** and **postinvoke** operations:

- The servant returned by **preinvoke** is used only to process the single request that caused **preinvoke** to be invoked.
- No servant incarnated by the servant manager will be placed in the Active Object Map.
- When the invocation of the request on the servant is complete, **postinvoke** will be invoked for the object.
- No serialization of invocations of **preinvoke** or **postinvoke** may be assumed; there may be multiple concurrent invocations of **preinvoke** for the same **ObjectId**.
- The same thread will be used to **preinvoke** the object, process the request, and **postinvoke** the object.

3.3.6.1 preinvoke

```
Servant preinvoke(
        in ObjectId             oid,
        in POA                  adapter,
        in CORBA::Identifier    operation,
        out Cookie              the_cookie )
    raises (ForwardRequest);
```

This operation is invoked by the POA whenever the POA receives a request for an object that is not currently active, assuming the POA has the **USE_SERVANT_MANAGER** and **NON_RETAIN** policies.

The **oid** parameter contains the **ObjectId** value associated with the incoming request. The **adapter** is an object reference for the POA in which the object is being activated.

The user-supplied servant manager implementation is responsible for locating or creating an appropriate servant that corresponds to the **ObjectId** value if possible. **preinvoke** returns a value of type **Servant**, which is the servant that will be used to process the incoming request.

The **Cookie** is a type opaque to the POA that can be set by the servant manager for use later by **postinvoke**. The **operation** is the name of the operation that will be called by the POA when the servant is returned.

3.3.6.2 postinvoke

```
void postinvoke(
        in ObjectId oid,
        in POA adapter,
        in CORBA::Identifier operation,
        in Cookie the_cookie,
        in Servant the_servant);
```

This operation is invoked whenever a servant completes a request, assuming the POA has the **USE_SERVANT_MANAGER** and **NON_RETAIN** policies.

The **oid** parameter contains the Object Id value of the object on which the request was made. The **adapter** parameter is an object reference for the POA in whose scope the object was active. The **serv** parameter contains a reference to the servant that is associated with the object.

The **Cookie** is a type opaque to the POA; it contains any value that was set by the **preinvoke** operation. The **operation** is the name of the operation that was called by the POA for the request.

Destroying a servant that is known to the POA can lead to undefined results.

3.3.7 POA Policy Objects

Interfaces derived from **CORBA::Policy** are used with the **POA::create_POA** operation to specify policies that apply to a POA. Policy objects are created using factory operations on any pre-existing POA, such as the root POA. Policy objects are specified when a POA is created. Policies may not be changed on an existing POA. Policies are *not* inherited from the parent POA.

3.3.7.1 Thread Policy

Objects with the **ThreadPolicy** interface are obtained using the **POA::create_thread_policy** operation and passed to the **POA::create_POA** operation to specify the threading model used with the created POA. The value attribute of **ThreadPolicy** contains the value supplied to the **POA::create_thread_policy** operation from which it was obtained. The following values can be supplied.

- **ORB_CTRL_MODEL** - The ORB is responsible for assigning requests for an ORB controlled POA to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads.
- **SINGLE_THREAD_MODEL** - Requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all upcalls made by this POA to implementation code (servants and servant managers) are made in a manner that is safe for code that multi-thread-unaware.

If no **ThreadPolicy** object is passed to **create_POA**, the thread policy defaults to **ORB_CTRL_MODEL**.

Note – In some environments, calling multi-thread-unaware code safely (that is, using the **SINGLE_THREAD_MODEL)** may mean that the POA will use only the main thread, in which case the application programmer is responsible to ensure that the main thread is given to the ORB, using **ORB::perform_work** or **ORB::run**.

POAs using the **SINGLE_THREAD_MODEL** may need to cooperate to ensure that calls are safe even when implementation code (such as a servant manager) is shared by multiple single-threaded POAs.

These models presume that the ORB and the application are using compatible threading primitives in a multi-threaded environment.

3.3.7.2 Lifespan Policy

Objects with the **LifespanPolicy** interface are obtained using the **POA::create_lifespan_policy** operation and passed to the **POA::create_POA** operation to specify the lifespan of the objects implemented in the created POA. The following values can be supplied.

- **TRANSIENT** - The objects implemented in the POA cannot outlive the process in which they are first created. Once the POA is deactivated, use of any object references generated from it will result in an **OBJECT_NOT_EXIST** exception.
- **PERSISTENT** - The objects implemented in the POA can outlive the process in which they are first created.
  - Persistent objects have a POA associated with them (the POA which created them). When the ORB receives a request on a persistent object, it first searches for the matching POA, based on the names of the POA and all of its ancestors.
  - Administrative action beyond the scope of this specification may be necessary to inform the ORB's location service of the creation and eventual termination of existence of this POA, and optionally to arrange for on-demand activation of a process implementing this POA.
  - POA names must be unique within their enclosing scope (the parent POA). A portable program can assume that POA names used in other processes will not conflict with its own POA names. A conforming CORBA implementation will provide a method for ensuring this property.

If no **LifespanPolicy** object is passed to **create_POA**, the lifespan policy defaults to **TRANSIENT**.

3.3.7.3 Object Id Uniqueness Policy

Objects with the **IdUniquenessPolicy** interface are obtained using the **POA::create_id_uniqueness_policy** operation and passed to the **POA::create_POA** operation to specify whether the servants activated in the created POA must have unique object identities. The following values can be supplied.

- **UNIQUE_ID** - Servants activated with that POA support exactly one Object Id.
- **MULTIPLE_ID** - a servant activated with that POA may support one or more Object Ids.

If no **IdUniquenessPolicy** is specified at POA creation, the default is **UNIQUE_ID**.

3.3.7.4 Id Assignment Policy

Objects with the **IdAssignmentPolicy** interface are obtained using the **POA::create_id_assignment_policy** operation and passed to the **POA::create_POA** operation to specify whehter Object Ids in the created POA are generated by the application or by the ORB. The following values can be supplied.

- **USER_ID -** Objects created with that POA are assigned Object Ids only by the application.
- **SYSTEM_ID** - Objects created with that POA are assigned Object Ids only by the POA. If the POA also has the **PERSISTENT** policy, assigned Object Ids must be unique across all instantiations of the same POA.

If no **IdAssignmentPolicy** is specified at POA creation, the default is **SYSTEM_ID**.

3.3.7.5 Servant Retention Policy

Objects with the **ServantRetentionPolicy** interface are obtained using the **POA::create_servant_retention_policy** operation and passed to the **POA::create_POA** operation to specify whether the created POA retains active servants in an Active Object Map. The following values can be supplied.

- **RETAIN -** The POA will retain active servants in its Active Object Map.
- **NON_RETAIN -** Servants are not retained by the POA.

If not **ServantRetentionPolicy** is specified at POA creation, the default is **RETAIN**.

Note – The **NON_RETAIN** policy requires either the **USE_DEFAULT_SERVANT** or **USE_SERVANT_MANAGER** policies.

3.3.7.6 Request Processing Policy

Objects with the **RequestProcessingPolicy** interface are obtained using the **POA::create_request_processing_policy** operation and passed to the **POA::create_POA** operation to specify how requests are processed by the created POA. The following values can be supplied.

- **USE_ACTIVE_OBJECT_MAP_ONLY** - If the Object Id is not found in the Active Object Map, an **OBJECT_NOT_EXIST** exception is returned to the client. The **RETAIN** policy is also required.
- **USE_DEFAULT_SERVANT** - If the Object Id is not found in the Active Object Map or the **NON_RETAIN** policy is present, and a default servant has been been registered with the POA using the **set_servant** operation, the request is dispatched to the default servant. If no default servant has been registered, a **OBJ_ADAPTER** exception is returned to the client. The **MULTIPLE_ID** policy is also required.
- **USE_SERVANT_MANAGER -** If the Object Id is not found in the Active Object Map or the **NON_RETAIN** policy is present, and a servant manager has been registered with the POA using the **set_servant_manager** operation, the servant manager is given the opportunity to locate a servant or raise an exception. If no servant manager has been registered, a **OBJECT_ADAPTER** exception is returned to the client.

If no RequestProcessingPolicy is specified at POA creation, the default is **USE_ACTIVE_OBJECT_MAP_ONLY**.

By means of combining the **USE_ACTIVE_OBJECT_MAP_ONLY** / **USE_DEFAULT_SERVANT** / **USE_SERVANT_MANAGER** policies and the **RETAIN** / **NON_RETAIN** policies, the programmer is able to define a rich number of possible behaviors.

3.3.7.6.1 RETAIN and USE_ACTIVE_OBJECT_MAP_ONLY

This combination represents the situation where the POA does no automatic object activation (that is, the POA searches only the Active Object Map). The server must activate all objects served by the POA explicitly, using either the **activate_object** or **activate_object_with_id** operation.

3.3.7.6.2 RETAIN and USE_SERVANT_MANAGER

This combination represents a very common situation, where there is an Active Object Map and a **ServantManager**.

Because **RETAIN** is in effect, the application can call **activate_object** or **activate_object_with_id** to establish known servants in the Active Object Map for use in later requests.

If the POA doesn't find a servant in the Active Object Map for a given object, it tries to determine the servant by means of invoking **incarnate** in the **ServantManager** (specifically a **ServantActivator**) registered with the POA. If no **ServantManager** is available, the POA raises the **OBJECT_ADAPTER** system exception.

3.3.7.6.3 RETAIN and USE_DEFAULT_SERVANT

This combination represents the situation where there is a default servant defined for all requests involving unknown objects.

Because **RETAIN** is in effect, the application can call **activate_object** or **activate_object_with_id** to establish known servants in the Active Object Map for use in later requests.

The POA first tries to find a servant in the Active Object Map for a given object. If it does not find such a servant, it uses the default servant. If no default servant is available, the POA raises the **OBJECT_ADAPTER** system exception.

3.3.7.6.4 NON-RETAIN and USE_SERVANT_MANAGER:

This combination represents the situation where one servant is used per method call.

The POA doesn't try to find a servant in the Active Object Map because the ActiveObjectMap does not exist. In every request, it will call **preinvoke** on the **ServantManager** (specifically a **ServantLocator**) registered with the POA. If no **ServantManager** is available, the POA will raise the **OBJECT_ADAPTER** system exception.

3.3.7.6.5 NON-RETAIN and USE_DEFAULT_SERVANT:

This combination represent the situation where there is one single servant defined for all CORBA objects.

The POA does not try to find a servant in the Active Object Map because the ActiveObjectMap doesn't exist. In every request, the POA will invoke the appropriate operation on the default servant registered with the POA. If no default servant is available, the POA will raise the **OBJECT_ADAPTER** system exception.

3.3.7.7 Implicit Activation Policy

Objects with the **ImplicitActivationPolicy** interface are obtained using the **POA::create_implicit_activation_policy** operation and passed to the **POA::create_POA** operation to specify whether implicit activation of servants is supported in the created POA. The following values can be supplied.

- **IMPLICIT_ACTIVATION** - the POA will support implicit activation of servants. **IMPLICIT_ACTIVATION** also requires the **SYSTEM_ID** and **RETAIN** policies.
- **NO_IMPLICIT_ACTIVATION** - the POA will not support implicit activation of servants.

If no **ImplicitActivationPolicy** is specified at POA creation, the default is **NO_IMPLICIT_ACTIVATION**.

3.3.8 POA Interface

A POA object manages the implementation of a collection of objects. The POA supports a name space for the objects, which are identified by Object Ids.

A POA also provides a name space for POAs. A POA is created as a child of an existing POA, which forms a hierarchy starting with the root POA.

3.3.8.1 Locality Constraints

A **POA** object must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a **MARSHAL** system exception. An attempt to use a **POA** object with the DII may raise the **NO_IMPLEMENT** exception.

3.3.8.2 create_POA

```
POA create_POA(in string adapter_name,
               in POAManager a_POAManager,
               in CORBA::PolicyList policies)
    raises (AdapterAlreadyExists, InvalidPolicy);
```

This operaton creates a new POA as a child of the target POA. The specified name identifies the new POA with respect to other POAs with the same parent POA. If the target POA already has a child POA with the specified name, the **AdapterAlreadyExists** exception is raised.

If the **a_POAManager** parameter is null, a new **POAManager** object is created and associated with the new POA. Otherwise, the specified **POAManager** object is associated with the new POA. The **POAManager** object can be obtained using the attribute name **the_POAManager**.

The specified policy objects are associated with the POA and used to control its behavior. The policy objects are effectively copied before this operation returns, so the application is free to destroy them while the POA is in use. Policies are *not* inherited from the parent POA.

If any of the policy objects specified are not valid for the ORB implementation, if conflicting policy objects are specified, or if any of the specified policy objects require prior administrative action that has not been performed, an **InvalidPolicy** exception is raised containing the index in the policies parameter value of the first offending policy object.

Note – Creating a POA using a POA manager that is in the active state can lead to race conditions if the POA supports preexisting objects, because the new POA may receive a request before its adapter activator, servant manager, or default servant have been initialized. These problems do not occur if the POA is created by an adapter activator registered with a parent of the new POA, because requests are queued until the adapter activator returns. To avoid these problems when a POA must be explicitly initialized, the application can initialize the POA by invoking **find_POA** with a TRUE activate parameter.

3.3.8.3 find_POA

```
POA find_POA(in string adapter_name,  in boolean activate_it)
    raises (AdapterNonExistent);
```

If the target POA is the parent of a child POA with the specified name (relative to the target POA), that child POA is returned. If a child POA with the specified name does not exist and the value of the **activate_it** parameter is TRUE, the target POA's **AdapterActivator**, if one exists, is invoked, and, if it successfully activates the child POA, that child POA is returned. Otherwise, the **AdapterNonExistent** exception is raised.

3.3.8.4 destroy

```
void destroy(   in boolean etherealize_objects,
                in boolean wait_for_completion);
```

This operation destroys the POA and all descendant POAs. The POA so destroyed (that is, the POA with its name) may be re-created later in the same process. (This differs from the **POAManager::deactivate** operation that does not allow a re-creation of its associated POA in the same process.)

When a POA is destroyed, any requests that have started execution continue to completion. Any requests that have not started execution are processed as if they were newly arrived, that is, the POA will attempt to cause recreation of the POA by invoking one or more adapter activators as described in Section 3.3.3.

If the **etherealize_objects** parameter is TRUE, the POA has the **RETAIN** policy, and a servant manager is registered with the POA, the **etherealize** operation on the servant manager will be called for each active object in the Active Object Map. The apparent destruction of the POA occurs before any calls to **etherealize** are made. Thus, for example, an etherealize method that attempts to invoke operations on the POA will receive the **OBJECT_NOT_EXIST** exception.

If the **wait_for_completion** parameter is TRUE, the **destroy** operation will return only after all requests in process have completed and all invocations of **etherealize** have completed. Otherwise, the **destroy** operation returns after destroying the POAs.

3.3.8.5 Policy Creation Operations

```
ThreadPolicy
    create_thread_policy(in ThreadPolicyValue value);
LifespanPolicy
    create_lifespan_policy(in LifespanPolicyValue value);
IdUniquenessPolicy
    create_id_uniqueness_policy(in IdUniquenessPolicyValue value);
IdAssignmentPolicy
    create_id_assignment_policy(in IdAssignmentPolicyValue value);
ImplicitActivationPolicy
    create_implicit_activation_policy
        (in ImplicitActivationPolicyValue value);
ServantRetentionPolicy
    create_servant_retention_policy(in ServantRetentionPolicyValue value);
RequestProcessingPolicy
    create_request_processing_policy
        (in RequestProcessingPolicyValue value);
```

These operations each return a reference to a policy object with the specified value. The application is responsible for calling the inherited destroy operation on the returned reference when it is no longer needed.

3.3.8.6 the_name

```
readonly attribute string the_name;
```

This attribute identifies the POA relative to its parent. This name is assigned when the POA is created. The name of the root POA is system-dependent and should not be relied upon by the application.

3.3.8.7 *the_parent*

readonly attribute POA the_parent;

This attribute identifies the parent of the POA. The parent of the root POA is null.

3.3.8.8 *the_POAManager*

readonly attribute POAManager the_POAManager;

This attribute identifies the POA manager associated with the POA.

3.3.8.9 *the_activator*

attribute AdapterActivator the_activator;

This attribute identifies the adapter activator associated with the POA. A newly created POA has no adapter activator (the attribute is null). It is system-dependent whether the root POA initially has an adapter activator; the application is free to assign its own adapter activator to the root POA.

3.3.8.10 *get_servant_manager*

ServantManager get_servant_manager()
raises(WrongPolicy);

This operation requires the **USE_SERVANT_MANAGER** policy; if not present, the **WrongPolicy** exception is raised.

This operation returns the servant manager associated with the POA. If no servant manager has been associated with the POA, it returns a null reference. It is system-dependent whether the root POA initially has a servant manager; the application is free to assign its own servant manager to the root POA.

3.3.8.11 *set_servant_manager*

void set_servant_manager(in ServantManager imgr)
raises(WrongPolicy);

This operation requires the **USE_SERVANT_MANAGER** policy; if not present, the **WrongPolicy** exception is raised.

This operation sets the default servant manager associated with the POA.

3.3.8.12 get_servant

```
Servant get_servant()
    raises(NoServant, WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy; if not present, the **WrongPolicy** exception is raised.

This operation returns the default servant associated with the POA. If no servant has been associated with the POA, the **NoServant** exception is raised.

3.3.8.13 set_servant

```
void set_servant(in Servant p_servant)
    raises(WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy; if not present, the **WrongPolicy** exception is raised.

This operation registers the specified servant with the POA as the default servant. This servant will be used for all requests for which no servant is found in the Active Object Map.

3.3.8.14 activate_object

```
ObjectId activate_object( in Servant p_servant)
    raises (ServantAlreadyActive, WrongPolicy);
```

This operation requires the **SYSTEM_ID** and **RETAIN** policy; if not present, the **WrongPolicy** exception is raised.

If the POA has the **UNIQUE_ID** policy and the specified servant is already in the Active Object Map, the **ServantAlreadyActive** exception is raised. Otherwise, the **activate_object** operation generates an Object Id and enters the Object Id and the specified servant in the Active Object Map. The Object Id is returned.

3.3.8.15 activate_object_with_id

```
void activate_object_with_id( in ObjectId oid,
                              in Servant p_servant)
    raises (ObjectAlreadyActive, ServantAlreadyActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the **WrongPolicy** exception is raised.

If the CORBA object denoted by the Object Id value is already active in this POA (there is a servant bound to it in the Active Object Map), the **ObjectAlreadyActive** exception is raised. If the POA has the **UNIQUE_ID** policy and the servant is already in the Active Object Map, the **ServantAlreadyActive** exception is raised. Otherwise, the **activate_object_with_id** operation enters an association between the specified Object Id and the specified servant in the Active Object Map.

If the POA has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this POA, the **activate_object_with_id** operation may raise the **BAD_PARAM** system exception. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke **activate_object_with_id** on a POA that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that POA, or, if the POA also has the **PERSISTENT** policy, for a previous instantiation of the same POA.

3.3.8.16 deactivate_object

```
void deactivate_object(in ObjectId oid)
    raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the **WrongPolicy** exception is raised.

This operation causes the association of the Object Id specified by the **oid** parameter and its servant to be removed from the Active Object Map. If a servant manager is associated with the POA, **ServantLocator::etherealize** will be invoked with the **oid** and the servant. (The **deactivate_object** operation does not wait for the **etherealize** operation to complete before **deactivate_object** returns.) If there is no active object associated with the specified Object Id, the operation raises an **ObjectNotActive** exception.

Note – If the servant associated with the **oid** is serving multiple Object Ids, **ServantLocator::etherealize** may be invoked multiple times with the same servant when the other objects are deactivated. It is the responsibility of the object implementation to refrain from destroying the servant while it is active with any Id.

3.3.8.17 create_reference

```
Object create_reference (in CORBA::RepositoryId intf)
    raises (WrongPolicy);
```

This operation requires the **SYSTEM_ID** policy; if not present, the **WrongPolicy** exception is raised.

This operation creates an object reference that encapsulates a POA-generated Object Id value and the specified interface repository id. This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the appropriate servant manager to be invoked, if one is available. The generated Object Id value may be obtained by invoking **POA::reference_to_id** with the created reference.

3.3.8.18 create_reference_with_id

```
Object create_reference_with_id (
        in ObjectId oid,
        in CORBA::RepositoryId intf );
```

This operation creates an object reference that encapsulates the specified Object Id and interface repository Id values. This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the object to be activated if necessary, or the default servant used, depending on the applicable policies.

If the POA has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this POA, the **create_reference_with_id** operation may raise the **BAD_PARAM** system exception. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke this operation on a POA that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that POA, or, if the POA also has the **PERSISTENT** policy, for a previous instantiation of the same POA.

3.3.8.19 servant_to_id

```
ObjectId servant_to_id(in Servant p_servant)
    raises (ServantNotActive, WrongPolicy);
```

This operation requires the **RETAIN** and either the **UNIQUE_ID** or **IMPLICIT_ACTIVATION** policies; if not present, the **WrongPolicy** exception is raised.

This operation has three possible behaviors.

- If the POA has the **UNIQUE_ID** policy and the specified servant is active, the Object Id associated with that servant is returned.
- If the POA has the **IMPLICIT_ACTIVATION** policy and either the POA has the **MULTIPLE_ID** policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and that Object Id is returned.
- Otherwise, the **ServantNotActive** exception is raised.

3.3.8.20 servant_to_reference

```
Object servant_to_reference (in Servant p_servant)
    raises (ServantNotActive, WrongPolicy);
```

This operation requires the **RETAIN** and either the **UNIQUE_ID** or **IMPLICIT_ACTIVATION** policies; if not present, the **WrongPolicy** exception is raised.

This operation has three possible behaviors.

- If the POA has the **UNIQUE_ID** policy and the specified servant is active, an object reference encapsulating the information used to activate the servant is returned.
- If the POA has the **IMPLICIT_ACTIVATION** policy and either the POA has the **MULTIPLE_ID** policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and a corresponding object reference is returned.
- Otherwise, the **ServantNotActive** exception is raised.

Note – The allocation of an Object Id value and installation in the Active Object Map caused by implicit activation may actually be deferred until an attempt is made to externalize the reference. The real requirement here is that a reference is produced that will behave appropriately (that is, yield a consistent Object Id value when asked politely).

3.3.8.21 reference_to_servant

```
Servant reference_to_servant (Object reference)
    raises (ObjectNotActive, WrongAdapter, WrongPolicy);
```

This operation requires the **RETAIN** policy or the **USE_DEFAULT_SERVANT** policy. If neither policy is present, the **WrongPolicy** exception is raised.

If the POA has the **RETAIN** policy and the specified object is present in the Active Object Map, this operation returns the servant associated with that object in the Active Object Map. Otherwise, if the POA has the **USE_DEFAULT_SERVANT** policy and a default servant has been registered with the POA, this operation returns the default servant. Otherwise, the **ObjectNotActive** exception is raised.

If the object reference was not created by this POA, the **WrongAdapter** exception is raised.

3.3.8.22 reference_to_id

```
ObjectId reference_to_id(in Object reference)
    raises (WrongAdapter, WrongPolicy);
```

The **WrongPolicy** exception is declared to allow future extensions.

This operation returns the Object Id value encapsulated by the specified **reference**. This operation is valid only if the reference was created by the POA on which the operation is being performed. If the reference was not created by that POA, a **WrongAdapter** exception is raised. The object denoted by the reference does not have to be active for this operation to succeed.

3.3.8.23 id_to_servant

```
Servant id_to_servant(in ObjectId oid)
    raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the **WrongPolicy** exception is raised.

This operation returns the active servant associated with the specified Object Id value. If the Object Id value is not active in the POA, an **ObjectNotActive** exception is raised.

3.3.8.24 id_to_reference

```
Object id_to_reference(in ObjectId oid)
    raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the **WrongPolicy** exception is raised.

If an object with the specified Object Id value is currently active, a reference encapsulating the information used to activate the object is returned. If the Object Id value is not active in the POA, an **ObjectNotActive** exception is raised.

3.3.9 Current operations

The **PortableServer::Current** interface, derived from **CORBA::Current**, provides method implementations with access to the identity of the object on which the method was invoked. The **Current** interface is provided to support servants that implement multiple objects, but can be used within the context of POA-dispatched method invocations on any servant. To provide location transparency, ORBs are required to support use of **Current** in the context of both locally-invoked and remotely-invoked operations.

An instance of **Current** can be obtained by the application by issuing the **CORBA::ORB::resolve_initial_references("POACurrent")** operation. Thereafter, it can be used within the context of a method dispatched by the POA to obtain the POA and ObjectId that identify the object on which that operation was invoked.

3.3.9.1 get_POA

```
POA get_POA() raises (NoContext);
```

This operation returns a reference to the POA implementing the object in whose context it is called. If called outside the context of a POA-dispatched operation, a **NoContext** exception is raised.

3.3.9.2 get_object_id

ObjectId get_object_id() raises (NoContext);

This operation returns the ObjectId identifying the object in whose context it is called. If called outside the context of a POA-dispatched operation, a **NoContext** exception is raised.

3.4 Consolidated IDL Related to PortableServer

3.4.1 IDL in CORBA module concerning the POA

```
// IDL
module CORBA
{
    …

    // basic Policy definition
    interface Policy
    {
        Policy copy( );
        void destroy( );
    };

    typedef sequence <Policy>  PolicyList;
    ...
    interface ORB {
        ...
        boolean work_pending(  );
        void perform_work( );
        void shutdown( in boolean wait_for_completion );
        void run();
        ...
    };

    // deprecate get_current in the next major printing of CORBA
};
```

3.4.2 IDL for POA and Related Interfaces

```
#pragma prefix "omg.org"
module PortableServer
{
    // forward reference
    interface POA;

    native Servant;

    typedef sequence<octet> ObjectId;

    exception ForwardRequest
    {
        Object forward_reference;
    };

    // ***********************************************
    //
    // Policy interfaces
    //
    // ***********************************************
    enum ThreadPolicyValue {
        ORB_CTRL_MODEL,
        SINGLE_THREAD_MODEL
    };
    interface ThreadPolicy : CORBA::Policy
    {
        readonly attribute ThreadPolicyValue value;
    };

    enum LifespanPolicyValue {
        TRANSIENT,
        PERSISTENT
    };
    interface LifespanPolicy : CORBA::Policy
    {
        readonly attribute LifespanPolicyValue value;
    };

    enum IdUniquenessPolicyValue {
        UNIQUE_ID,
        MULTIPLE_ID
    };
    interface IdUniquenessPolicy : CORBA::Policy
    {
        readonly attribute IdUniquenessPolicyValue value;
    };

    enum IdAssignmentPolicyValue {
```

```
    USER_ID,
    SYSTEM_ID
};
interface IdAssignmentPolicy : CORBA::Policy
{
    readonly attribute IdAssignmentPolicyValue value;
};

enum ImplicitActivationPolicyValue {
    IMPLICIT_ACTIVATION,
    NO_IMPLICIT_ACTIVATION
};
interface ImplicitActivationPolicy : CORBA::Policy
{
    readonly attribute ImplicitActivationPolicyValue value;
};

enum ServantRetentionPolicyValue {
    RETAIN,
    NON_RETAIN
};
interface ServantRetentionPolicy : CORBA::Policy
{
    readonly attribute ServantRetentionPolicyValue value;
};

enum RequestProcessingPolicyValue {
    USE_ACTIVE_OBJECT_MAP_ONLY,
    USE_DEFAULT_SERVANT,
    USE_SERVANT_MANAGER
};
interface RequestProcessingPolicy : CORBA::Policy
{
    readonly attribute RequestProcessingPolicyValue value;
};

// ***************************************************
//
// POAManager interface
//
// ***************************************************

interface POAManager
{
    exception AdapterInactive{};

    void activate()
        raises(AdapterInactive);
    void hold_requests(in boolean wait_for_completion)
        raises(AdapterInactive);
    void discard_requests(in boolean wait_for_completion)
```

```
            raises(AdapterInactive);
        void deactivate(    in boolean etherealize_objects,
                            in boolean wait_for_completion)
            raises(AdapterInactive);
};

// ************************************************
//
// AdapterActivator interface
//
// ************************************************

interface AdapterActivator
{
    boolean unknown_adapter(in POA parent, in string name);
};

// ************************************************
//
// ServantManager interface
//
// ************************************************

interface ServantManager
{ };

interface ServantActivator : ServantManager {
     Servant incarnate (
                in ObjectId             oid,
                in POA                  adapter )
          raises (ForwardRequest);

    void etherealize (
                in ObjectId             oid,
                in POA                  adapter,
                in Servant              serv,
                in boolean              cleanup_in_progress,
                in boolean              remaining_activations );
};

interface ServantLocator : ServantManager {
          native Cookie;
          Servant preinvoke(
                in ObjectId             oid,
                in POA                  adapter,
                in CORBA::Identifier    operation,
                out Cookie              the_cookie )
              raises (ForwardRequest);

          void postinvoke(
                in ObjectId             oid,
```

```
                    in POA                  adapter,
                    in CORBA::Identifier    operation,
                    in Cookie               the_cookie,
                    in Servant              the_servant);
};


// ***************************************************
//
// POA interface
//
// ***************************************************

interface POA
{
    exception AdapterAlreadyExists {};
    exception AdapterInactive {};
    exception AdapterNonExistent {};
    exception InvalidPolicy { unsigned short index; };
    exception NoServant {};
    exception ObjectAlreadyActive {};
    exception ObjectNotActive {};
    exception ServantAlreadyActive {};
    exception ServantNotActive {};
    exception WrongAdapter {};
    exception WrongPolicy {};

    //--------------------------------------------------
    //
    // POA creation and destruction
    //
    //--------------------------------------------------

    POA create_POA(in string adapter_name,
                        in POAManager a_POAManager,
                        in CORBA::PolicyList policies)
        raises (AdapterAlreadyExists, InvalidPolicy);

    POA find_POA(in string adapter_name,  in boolean activate_it)
        raises (AdapterNonExistent);

    void destroy(   in boolean etherealize_objects,
                    in boolean wait_for_completion);

    // ***************************************************
    //
    // Factories for Policy objects
    //
    // ***************************************************

    ThreadPolicy
```

```
    create_thread_policy(in ThreadPolicyValue value);
LifespanPolicy
    create_lifespan_policy(in LifespanPolicyValue value);
IdUniquenessPolicy
    create_id_uniqueness_policy
        (in IdUniquenessPolicyValue value);
IdAssignmentPolicy
    create_id_assignment_policy
        (in IdAssignmentPolicyValue value);
ImplicitActivationPolicy
    create_implicit_activation_policy
        (in ImplicitActivationPolicyValue value);
ServantRetentionPolicy
    create_servant_retention_policy
        (in ServantRetentionPolicyValue value);
RequestProcessingPolicy
    create_request_processing_policy
        (in RequestProcessingPolicyValue value);

//--------------------------------------------------
//
// POA attributes
//
//--------------------------------------------------

readonly attribute string the_name;
readonly attribute POA the_parent;
readonly attribute POAManager the_POAManager;
attribute AdapterActivator the_activator;


//--------------------------------------------------
//
// Servant Manager registration:
//
//--------------------------------------------------

ServantManager get_servant_manager()
    raises (WrongPolicy);

void set_servant_manager( in ServantManager imgr)
    raises (WrongPolicy);

//--------------------------------------------------
//
// operations for the USE_DEFAULT_SERVANT policy
//
//--------------------------------------------------

Servant get_servant()
    raises (NoServant, WrongPolicy);
```

```
void set_servant(in Servant p_servant)
    raises (WrongPolicy);

// ***************************************************
//
// object activation and deactivation
//
// ***************************************************

ObjectId activate_object( in Servant p_servant )
    raises (ServantAlreadyActive, WrongPolicy);

    void activate_object_with_id(
                            in ObjectId id,
                            in Servant p_servant)
    raises (ServantAlreadyActive, ObjectAlreadyActive,
            WrongPolicy);

void deactivate_object(in ObjectId oid)
    raises (ObjectNotActive, WrongPolicy);

// ***************************************************
//
// reference creation operations
//
// ***************************************************

Object create_reference (
        in CORBA::RepositoryId intf )
    raises (WrongPolicy);

Object create_reference_with_id (
        in ObjectId oid,
        in CORBA::RepositoryId intf )
    raises (WrongPolicy);


//-------------------------------------------------
//
// Identity mapping operations:
//
//-------------------------------------------------

ObjectId servant_to_id(in Servant p_servant)
    raises (ServantNotActive, WrongPolicy);

Object servant_to_reference(in Servant p_servant)
    raises (ServantNotActive, WrongPolicy);

Servant reference_to_servant(in Object reference)
```

```
            raises (ObjectNotActive, WrongAdapter, WrongPolicy);

        ObjectId reference_to_id(in Object reference)
            raises (WrongAdapter, WrongPolicy);

        Servant id_to_servant(in ObjectId oid)
            raises (ObjectNotActive, WrongPolicy);

        Object id_to_reference(in ObjectId oid)
            raises (ObjectNotActive, WrongPolicy);

    };


    // ************************************************
    //
    // Current interface
    //
    // ************************************************

    interface Current : CORBA::Current
    {
        exception NoContext { };

        POA get_POA() raises (NoContext);
        ObjectId get_object_id() raises (NoContext);
    };

};
```

3.5 UML Description of PortableServer

The following diagrams were generated by an automated tool and then annotated with the cardinalities of the associations. They are intended to be an aid in comprehension to those who enjoy such representations. They are are not normative.

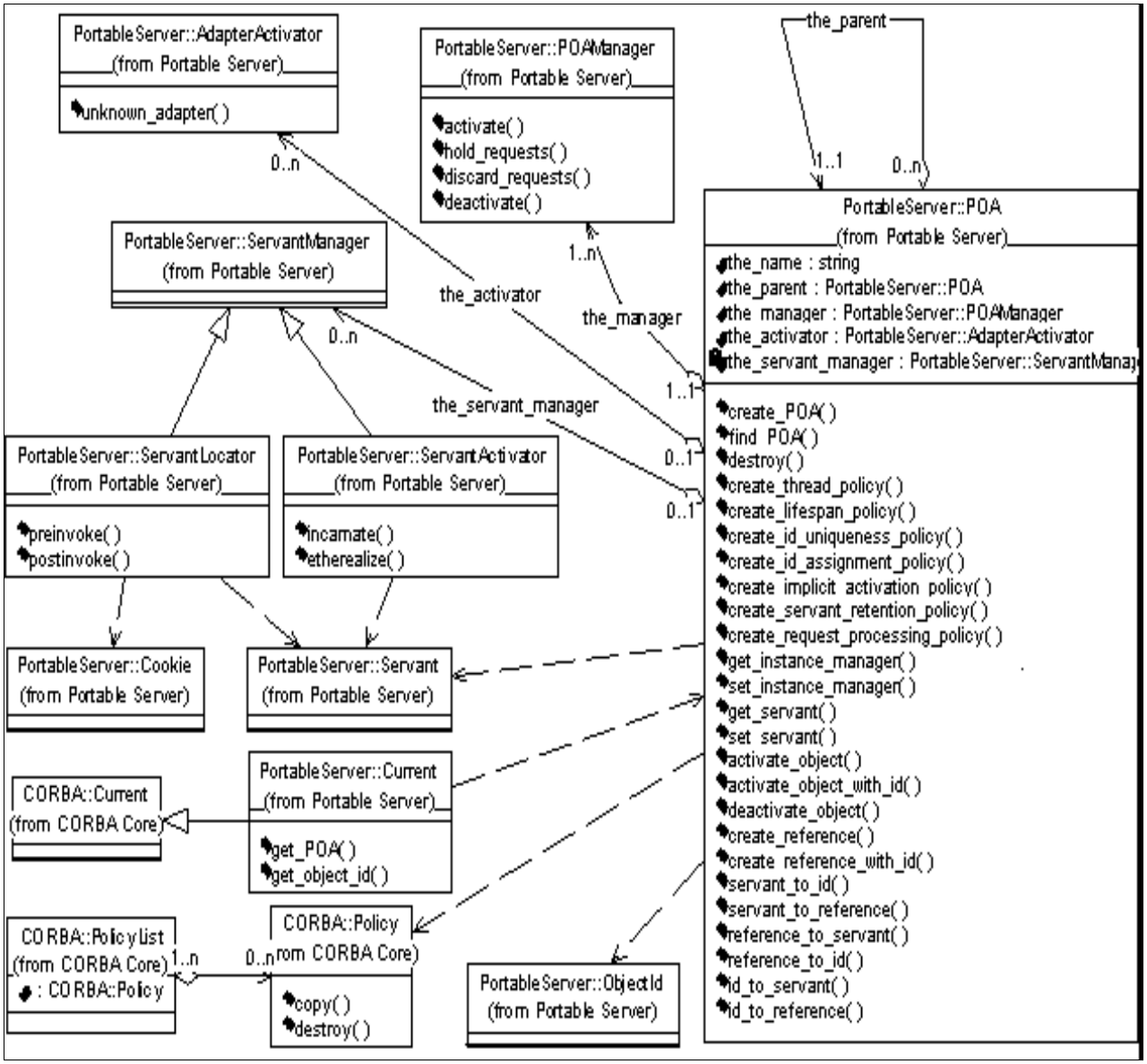

Figure 3-4 UML for main part of PortableServer

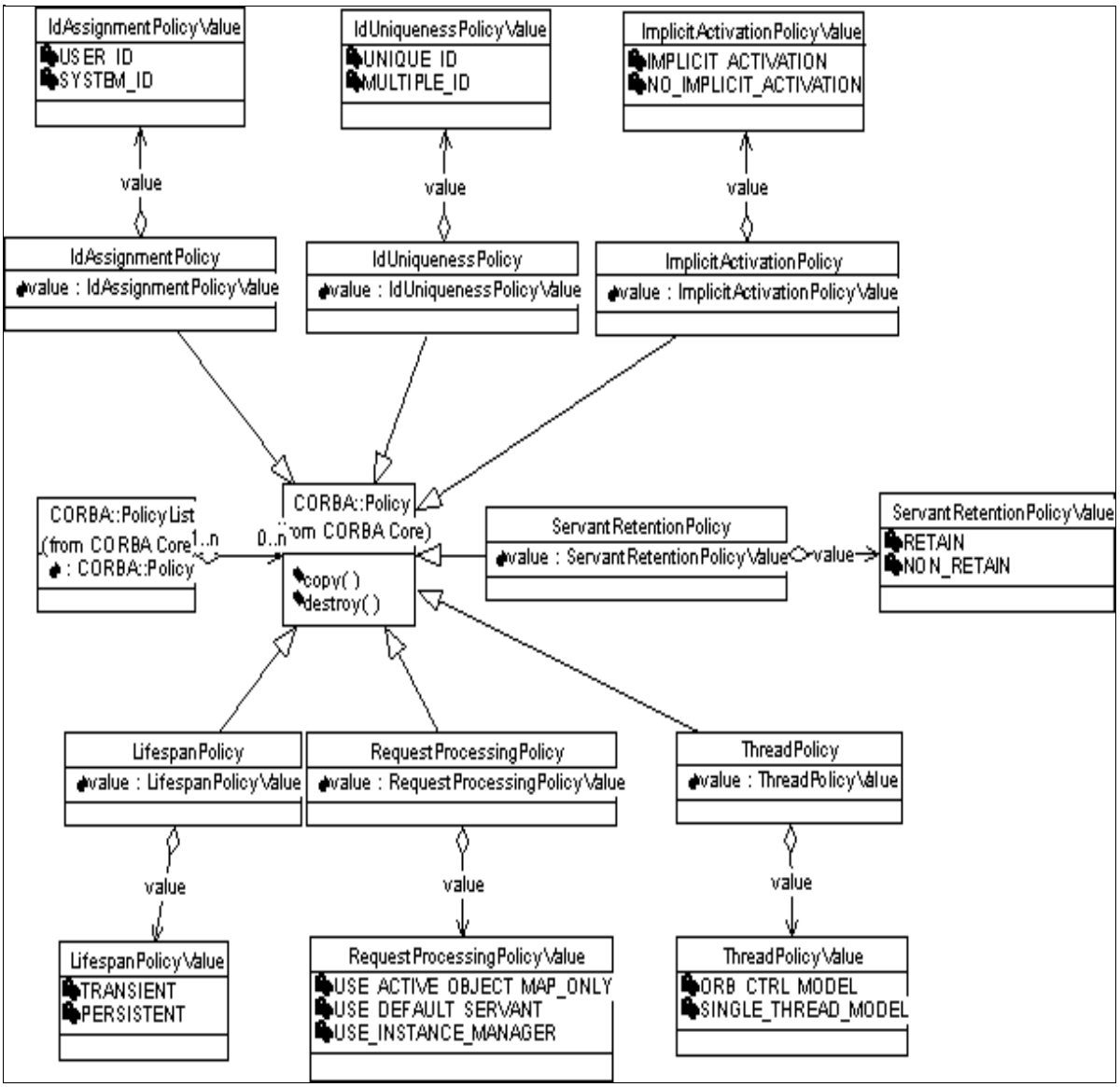

Figure 3-5 UML for PortableServer policies

3.6 Usage Scenarios

This section illustrates how different capabilities of the POA may be used in applications.

Note – In some of the following C++ examples, PortableServer names are not explicitly scoped. It is assumed that all the examples have the C++ statement
`using namespace PortableServer;`

3.6.1 Getting the root POA

All server applications must obtain a reference to the root POA, either to use it directly to manage objects or to create new POA objects. The following example demonstrates how the application server can obtain a reference to the root POA.

```
// C++
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CORBA::Object_ptr pfobj =
   orb->resolve_initial_references("RootPOA");
PortableServer::POA_ptr rootPOA;
rootPOA = PortableServer::POA::narrow(pfobj);
```

3.6.2 Creating a POA

For a variety of reasons, a server application might want to create a new POA. The POA is created as a child of an existing POA. In this example, it is created as a child of the root POA.

```
// C++
CORBA::PolicyList policies(2);
policies[0] = rootPOA->create_thread_policy(
         PortableServer::ThreadPolicy::ORB_CTRL_MODEL);
policies[1] = rootPOA->create_lifespan_policy(
         PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
   rootPOA->create_POA("my_little_poa",
         PortableServer::POAManager::_nil(), policies);
```

3.6.3 Explicit Activation with POA-assigned Object Ids

By specifying the **SYSTEM_ID** policy on a POA, objects may be explicitly activated through the POA without providing a user-specified identity value. Using this approach, objects are activated by performing the **activate_object** operation on the POA with the object in question. For this operation, the POA allocates, assigns, and returns a unique identity value for the object.

Generally this capability is most useful for transient objects, where the Object Id needs to be valid only as long as the servant is active in the server. The Object Ids can remain completely hidden and no servant manager need be provided. When this is the case, the identity and lifetime of the servant and the abstract object are essentially equivalent. When POA-assigned Object Ids are used with persistent objects or objects that are activated on demand, the application must be able to associate the generated Object Id value with its corresponding object state.

This example illustrates a simple implementation of transient objects using POA-assigned Object Ids. It presumes a POA that has the **SYSTEM_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Assume this interface:

```
// IDL
interface Foo
{
    long doit();
}
```

This might result in the generation of the following skeleton:

```
class POA_Foo : public ServantBase
{
public:
...
   virtual CORBA::Long doit() = 0;
}
```

Derive your implementation:

```
class MyFooServant : public POA_Foo
{
   public:
      MyFooServant(POA_ptr poa, Long value)
         : my_poa(POA::_duplicate(poa)), my_value(value) {}
      ~MyFooServant() { CORBA::release(my_poa); }
      virtual POA_ptr _default_POA()
         { return POA::_duplicate(my_poa); }
      virtual Long doit() { return my_value; }
   protected:
      POA_ptr my_poa;
      Long my_value;
};
```

Now, somewhere in the program during initialization, probably in **main()**:

```
MyFooServant* afoo = new MyFooServant(poa,27);
PortableServer::ObjectId_var oid =
       poa->activate_object(afoo);
Foo_var foo = afoo->_this();
poa->the_POAManager()->activate();
orb->run();
```

This object is activated with a generated Object Id.

3.6.4 Explicit activation with user assigned Object Ids

An object may be explicitly activated by a server using a user-assigned identity. This may be done for several reasons. For example, a programmer may know that certain objects are commonly used, or act as initial points of contact through which clients access other objects (for example, factories). The server could be implemented to create and explicitly activate these objects during initialization, avoiding the need for a servant manager.

If an implementation has a reasonably small number of servants, the server may be designed to keep them all active continuously (as long as the server is executing). If this is the case, the implementation need not provide a servant manager. When the server initializes, it could create all available servants, loading their state and identities from some persistent store. The POA supports an explicit activation operation, **activate_object_with_id**, that associates a servant with an Object Id. This operation would be used to activate all of the existing objects managed by the server during server initialization. Assuming the POA has the **USE_SERVANT_MANAGER** policy and no servant manager is associated with a POA, any request received by the POA for an Object Id value not present in the Active Object Map will result in an **OBJECT_NOT_EXIST** exception.

In simple cases of well-known, long-lived objects, it may be sufficient to activate them with well known Object Id values during server initialization, before activating the POA. This approach ensures that the objects are always available when the POA is active, and doesn't require writing a servant managers. It has severe practical limitations for a large number of objects, though.

This example illustrates the explicit activation of an object using a user chosen Object Id. This example presumes a POA that has the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

The code is like the previous example, but replace the last portion of the example shown above with the following code:

```
// C++
MyFooServant* afoo = new MyFooServant(poa, 27);
PortableServer::ObjectId_var oid =
          PortableServer::string_to_ObjectId("myLittleFoo");
poa->activate_object_with_id(oid.in(), afoo);
Foo_var foo = afoo->_this();
```

3.6.5 Creating references before activation

It is sometimes useful to create references for objects before activating them. This example extends the previous example to illustrate this option:

```
// C++
PortableServer::ObjectId_var oid =
      PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
                  oid.in(), "IDL:Foo:1.0");
Foo_var foo = Foo::_narrow(obj);

// ...later...

MyFooServant* afoo = new MyFooServant(poa, 27);
poa->activate_object_with_id(oid.in(), afoo);
```

3.6.6 Servant Manager Definition and Creation

Servant managers are object implementations, and are required to satisfy all of the requirements of object implementations necessary for their intended function. Because servant managers are local objects, and their use is limited to a single narrow role, some simplifications in their implementation are possible. Note that these simplifications are suggestions, not normative requirements. They are intended as examples of ways to reduce the programming effort required to define servant managers.

A servant manager implementation must provide the following things:

- implementation code for either
  - **incarnate( )** and **etherealize( )**, or
  - **preinvoke( )** and **postinvoke( )**
- implementation code for the servant operations, as for all servants

The first two are obvious; their content is dictated by the requirements of the implementation that the servant manager is managing. For the third point, the default servant manager on the root POA already supplies this implementation code. User written servant managers will have to provide this themselves.

Since servant managers are objects, they themselves must be activated. It is expected that most servant managers can be activated on the root POA with its default set of policies (see Section 3.2.3, “POA Creation”). It is for this reason that the root POA has the **IMPLICIT_ACTIVATION** policy: so that a servant manager can easily be activated. Users may choose to activate a servant manager on other POAs.

The following is an example servant manager to activate objects on demand. This example presumes a POA that has the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Since RETAIN is in effect, the type of servant manager used is a ServantActivator. The ORB supplies a servant activator skeleton class in a library:

```
// C++
namespace POA_PortableServer
{
   class ServantActivator : public virtual ServantManager
      {
         public:
            virtual ~ServantActivator();
            virtual Servant incarnate(
               const ObjectId& POA_ptr poa) = 0;
            virtual void etherealize(
               const ObjectId&, POA_ptr poa,
               Servant, Boolean remaining_activations) = 0;
      };
}
```

A ServantActivator servant manager might then look like:

```
// C++
class MyFooServantActivator : public POA_PortableServer::Ser-
vantActivator
{
   public:
      // ...
      Servant incarnate(
         const ObjectId& oid, POA_ptr poa)
      {
         String_var s = PortableServer::ObjectId_to_string
                           (oid);
         if (strcmp(s, "myLittleFoo") == 0) {
            return new MyFooServant(poa, 27);
         } else {
            throw CORBA::OBJECT_NOT_EXIST();
         }
      }
      void etherealize(
               const ObjectId& oid,
               POA_ptr poa,
               Servant servant,
               Boolean remaining_activations)
      {
         if (remaining_activations == 0)
            delete servant;
      }
      // ...
};
```

3.6.7 Object activation on demand

The precondition for this scenario is the existence of a client with a reference for an object with which no servant is associated at the time the client makes a request on the reference. It is the responsibility of the ORB, in collaboration with the POA and the server application to find or create an appropriate servant and perform the requested operation on it. Such an object is said to be *incarnated* when it has an active servant (or, *incarnation*). Note that the client had to obtain the reference in question previously from some source. From the client's perspective, the abstract object exists as long as it holds a reference, until it receives an **OBJECT_NOT_EXIST** system exception in a reply from an attempted request on the object. Incarnation state does not imply existence or non-existence of the abstract object.

Note – This specification does not address this issues of communication or server process activation, as they are immaterial to the POA interface and operation. It is assumed that the ORB activates the server if necessary, and can deliver the request to the appropriate POA.

To support object activation on demand, the server application must register a servant manager with the appropriate POA. Upon receiving the request, if the POA consults the Active Object Map and discovers that there is no active servant associated with the target Object Id, the POA invokes the **incarnate** operation on the servant manager.

Note – An implication that this model has for GIOP is that the object key in the request message must encapsulate the Object Id value. In addition, it may encapsulate other values as necessitated by the ORB implementation. For example, the server must be able to determine to which POA the request should be directed. It could assign a different communication endpoint to each POA so that the POA identity is implicit in the request, or it could use a single endpoint for the entire server and encapsulate POA identities in object key values. Note that this is not a concrete requirement; the object key may not actually contain any of those values. Whatever the concrete information is, the ORB and POA must be able to use it to find the servant manager, invoke activate if necessary (which requires the actual Object Id value), and/or find the active servant in some map.

The **incarnate** invocation passes the Object Id value to the servant manager. At this point, the servant manager may take any action necessary to produce a servant that it considers to be a valid incarnation of the object in question. The operation returns the servant to the POA, which invokes the operation on it. The **incarnate** operation may alternatively raise an **OBJECT_NOT_EXIST** system exception that will be returned to the invoking client. In this way, the user-supplied implementation is responsible for determining object existence and non-existence.

After activation, the POA maintains the association of the servant and the Object Id in the Active Object Map. (This example presumes the **RETAIN** and **USE_SERVANT_MANAGER** policies.)

As an obvious example of transparent activation, the Object Id value could contain a key for a record in a database that contains the object's state. The servant manager would retrieve the state from the database, construct a servant of the appropriate implementation class (assuming an object-oriented programming language), initialize it with the state from the database, and return it to the POA.

The example servant manager in the last section (Section 3.6.6, "Servant Manager Definition and Creation") could be used for this scenario. Recall that the POA would have the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Given such a ServantActivator, all that remains is initialization code such as the following.

```
PortableServer::ObjectId_var oid =
      PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
                            oid, "IDL:foo:1.0");
MyFooServantActivator* fooIM = new MyFooServantActivator;
ServantActivator_var IMref = fooIM->_this();
poa->set_servant_manager(IMref);
poa->the_POAmanager()->activate();
orb->run();
```

3.6.8 Persistent objects with POA-assigned Ids

It is possible to access the Object Id value assigned to an object by the POA, with the **POA::reference_to_id** operation. If the reference is for an object managed by the POA that is the operation's target, the operation will return the Object Id value, whether it was assigned by the POA or the user. By doing this, an implementation may provide a servant manager that associates the POA-allocated Object Id values with persistently stored state. It may also pass the POA-allocated Object Id values to POA operations such as **activate_object_with_id** and **create_reference_with_id**.

A POA with the **PERSISTENT** policy may be destroyed and later reinstantiated in the same or a different process. A POA with both the **SYSTEM_ID** and **PERSISTENT** policies generates Object Id values are unique across all instantiations of the same POA.

3.6.9 Multiple Object Ids Mapping to a Single Servant

Each POA is created with a policy that indicates whether or not servants are allowed to support multiple object identities simultaneously. If a POA allows multiple identities per servant, the POA's treatment of the servants is affected in the following ways:

- Servants of the type may be explicitly activated multiple times with different identity values without raising an exception.
- A servant cannot be mapped onto or converted to an individual object reference using that POA, since the identity is potentially ambiguous.

3.6.10 One Servant for all Objects

By using the **USE_DEFAULT_SERVANT** policy, the developer can create a POA that will use a single servant to implement all of its objects. This approach is useful when there is very little data associated with each object, so little that the data can be encoded in the Object Id.

The following example illustrates this approach by using a single servant to incarnate all CORBA objects that export a given interface in the context of a server. This example presumes a POA that has the **USER_ID**, **NON_RETAIN,** and **USE_DEFAULT_SERVANT** policies.

Two interfaces are defined in IDL. The **FileDescriptor** interface is supported by objects that will encapsulate access to operations in a file, associated with a file system. Global operations in a file system, such as the ones necessary to create **FileDescriptor** objects, are supported by objects that export the **FileSystem** interface.

```
// IDL
interface FileDescriptor {
    typedef     sequence<octet> DataBuffer;

    long        write (in DataBuffer buffer);
    DataBuffer  read (in long num_bytes);
    void        destroy ();
};

interface FileSystem {
    ...
    FileDescriptor open (in string file_name, in long flags);
    ...
};
```

Implementation of these two IDL interfaces may inherit from static skeleton classes generated by an IDL to C++ compiler as follows:

```
// C++
class FileDescriptorImpl : public POA_FileDescriptor
{
   public:
      FileDescriptorImpl(POA_ptr poa);
      ~FileDescriptorImpl();
      POA_ptr _default_POA();
      CORBA::Long write(
            const FileDescriptor::DataBuffer& buffer);
      FileDescriptor::DataBuffer* read(
            CORBA::Long num_bytes);
      void destroy();
   private:
      POA_ptr my_poa;
};

class FileSystemImpl : public POA_FileSystem
{
   public:
      FileSystemImpl(POA_ptr poa);
      ~FileSystemImpl();
      POA_ptr _default_POA();
      FileDescriptor_ptr open(
         const char* file_name, CORBA::Long flags);
   private:
      POA_ptr my_poa;
      FileDescriptorImpl* fd_servant;
};
```

A single servant may be used to serve all requests issued to all **FileDescriptor** objects created by a **FileSystem** object. The following fragment of code illustrates the steps to perform when a **FileSystem** servant is created.

```
// C++
FileSystemImpl::FileSystemImpl(POA_ptr poa)
   : my_poa(POA::_duplicate(poa))
{
   fd_servant = new FileDescriptorImpl(poa);
   poa->set_servant(fd_servant);
}
```

The following fragment of code illustrates how **FileDescriptor** objects are created as a result of invoking an operation (**open**) exported by a **FileSystem** object. First, a local file descriptor is created using the appropriate operating system call. Then, a CORBA object reference is created and returned to the client. The value of the local file descriptor will be used to distinguish the new **FileDescriptor** object from other **FileDescriptor** objects. Note that FileDescriptor objects in the example are transient, since they use the value of their file descriptors for their ObjectIds, and of course the file descriptors are only valid for the life of a process.

```
// C++
FileDescriptor_ptr
FileSystemImpl::open(
      const char* file_name, CORBA::Long flags)
{
   int fd = ::open(file_name, flags);
   ostrstream ostr;
   ostr << fd;
   PortableServer::ObjectId_var oid =
      PortableServer::string_to_ObjectId(ostr.str());
   Object_var obj = my_poa->create_reference_with_id(
            oid.in(),"IDL:FileDescriptor:1.0");
   return FileDescriptor::_narrow(obj);
}
```

Any request issued to a **FileDescriptor** object is handled by the same servant. In the context of a method invocation, the servant determines which particular object is being incarnated by invoking an operation that returns a reference to the target object and, after that, invoking **POA::reference_to_id**. In C++, the operation used to obtain a reference to the target object is **`_this()`**. Typically, the **ObjectId** value associated with the reference will be used to retrieve the state of the target object. However, in this example, such step is not required since the only thing that is needed is the value for the local file descriptor and that value coincides with the **ObjectId** value associated with the reference.

Implementation of the **read** operation is rather simple. The servant determines which object it is incarnating, obtains the local file descriptor matching its identity, performs the appropriate operating system call, and returns the result in a **DataBuffer** sequence.

```
// C++
FileDescriptor::DataBuffer*
FileDescriptorImpl::read(CORBA::Long num_bytes)
{
   FileDescriptor_var me = _this();
   PortableServer::ObjectId_var oid =
      my_poa->reference_to_id(me.in());
   CORBA::String_var s =
      PortableServer::ObjectId_to_string(oid.in());
   istrstream is(s);
   int fd;
   is >> fd;
   CORBA::Octet* buffer = DataBuffer::alloc_buf(num_bytes);
   int len = ::read(fd, buffer, num_bytes);
   DataBuffer* result = new DataBuffer(len, len, buffer, 1);
   return result;
}
```

Using a single servant per interface is useful in at least two situations.

- In one case, it may be appropriate for encapsulating access to legacy APIs that are not object-oriented (system calls in the Unix environment, as we have shown in the example).
- In another case, this technique is useful in handling scalability issues related to the number of CORBA objects that can be associated with a server. In the example above, there may be a million **FileDescriptor** objects in the same server and this would only require one entry in the ORB. Although there are operating system limitations in this respect (a Unix server is not able to open so many local file descriptors) the important point to take into account is that usage of CORBA doesn't introduce scalability problems but provides mechanisms to handle them.

3.6.11 Single Servant, many objects and types, using DSI

The ability to associate a single DSI servant with many CORBA objects is rather powerful in some scenarios. It can be the basis for development of gateways to legacy systems or software that mediates with external hardware, for example.

Usage of the DSI is illustrated in the following example. This example presumes a POA that supports the **USER_ID**, **USE_DEFAULT_SERVANT**, and **RETAIN** policies.

A single servant will be used to incarnate a huge number of CORBA objects, each of them representing a separate entry in a Database. There may be several types of entries in the Database, representing different entity types in the Database model. Each type of entry in the Database is associated with a separate interface which comprises operations supported by the Database on entries of that type. All these interfaces inherit from the **DatabaseEntry** interface. Finally, an object supporting the **DatabaseAgent** interface supports basic operations in the database such as creating a new entry, destroying an existing entry, etc.

```
// IDL
interface DatabaseEntry {
    readonly attribute string name;
};

interface Employee : DatabaseEntry {
    attribute long id;
    attribute long salary;
};

...

interface DatabaseAgent {
    DatabaseEntry create_entry (
                    in string key,
                    in CORBA::Identifier entry_type,
                    in NVPairSequence initial_attribute_values
    ) ;
    void destroy_entry (in string key);
    ...
};
```

Implementation of the **DatabaseEntry** interface may inherit from the standard dynamic skeleton class as follows:

```
// C++
class DatabaseEntryImpl :
      public POA_PortableServer::DynamicImplementation
{
public:
   DatabaseEntryImpl (DatabaseAccessPoint db);
   virtual void invoke (ServerRequest_ptr request);
   ~DatabaseEntryImpl ();

   virtual POA_ptr _defaultPOA()
   {
      return poa;
   }

};
```

On the other hand, implementation of the **DatabaseAgent** interface may inherit from a static skeleton class generated by an IDL to C++ compiler as follows:

```
// C++
class DatabaseAgentImpl :
      public DatabaseAgentImplBase
{
protected:
   DatabaseAccessPoint mydb;
   DatabaseEntryImpl * common_servant;
public:
   DatabaseAgentImpl ();
   virtual DatabaseEntry_ptr create_entry (
            const char * key,
            const char * entry_type,
            const NVPairSequence& initial_attribute_values
   );
   virtual void destroy_entry (const char * key);
   ~DatabaseAgentImpl ();
};
```

A single servant may be used to serve all requests issued to all **DatabaseEntry** objects created by a **DatabaseAgent** object. The following fragment of code illustrates the steps to perform when a **DatabaseAgent** servant is created. First, access to the database is initialized. As a result, some kind of descriptor (a DatabaseAccessPoint local object) used to operate on the database is obtained. Finally, a servant will be created and associated with the POA.

```
// C++
void DatabaseAgentImpl::DatabaseAgentImpl ()
{
   mydb = ...;
   common_servant = new DatabaseEntryImpl(mydb);
   poa->set_servant(common_servant);
};
```

The code used to create **DatabaseEntry** objects representing entries in the database is similar to the one used for creating **FileDescriptor** objects in the example of the previous section. In this case, a new entry is created in the database and the key associated with that entry will be used to represent the identity for the corresponding **DatabaseEntry** object. All requests issued to a **DatabaseEntry** object are handled by the same servant because references to this type of object are associated with a common POA created with the **USE_DEFAULT_SERVANT** policy.

```
// C++
DatabaseEntry_ptr DatabaseAgentImpl::create_entry (
            const char * key,
            const char * entry_type,
            const NVPairSequence& initial_attribute_values)

   // creates a new entry in the database:
   mydb->new_entry (key, ...);

   // creates a reference to the CORBA object used to
   // encapsulate access to the new entry in the database.
   // There is an interface for each entry type:
   CORBA::Object_ptr obj = poa->create_reference_with_id(
      string_to_ObjectId (key),
      identifierToRepositoryId (entry_type),
   );

   DatabaseEntry_ptr entry = DatabaseEntry::_narrow (obj);
   CORBA::release (obj);
   return entry;
};
```

Any request issued to a **DatabaseEntry** object is handled by the same servant. In the context of a method invocation, the servant determines which particular object it is incarnating, obtains the database key matching its identity, invokes the appropriate operation in the database and returns the result as an output parameter in the **ServerRequest** object.

Sometimes, a program may need to determine the type of an entry in the database in order to invoke operations on the entry. If that is the case, the servant may obtain the type of an entry based on the interface supported by the **DatabaseEntry** object encapsulating access to that entry. This interface may be obtained by means of invoking the **get_interface** operation exported by the reference to the **DatabaseEntry** object.

```
// C++
void DatabaseEntryImpl::invoke (ServerRequest_ptr request)
{
   CORBA::Object_ptr current_obj = _this ();

   // The servant determines the key associated with
   // the database entry represented by current_obj:
   PortableServer::ObjectId oid =
      poa->reference_to_id (current_obj);
   char * key = ObjectId_to_string (oid);

   // The servant handles the incoming CORBA request.  This
   // typically involves the following steps:
   // 1. mapping the CORBA request into a database request
   //    using the key obtained previously
   // 2. constructing output parameters to the CORBA request
   //    from the response to the database request
   ...
};
```

Note that in this example, we may have a billion **DatabaseEntry** objects in a server requiring only a single entry in map tables supported by the POA (that is, the ORB at the server). No permanent storage is required for references to DatabaseEntry objects at the server. Actually, references to DatabaseEntry objects will only occupy space:

- at clients, as long as those references are used; or
- at the server, only while a request is being served

Scalability problems can be handled using this technique. There are many scenarios where this scalability causes no penalty in terms of performance (basically, when there is no need to restore the state of an object, each time a request to it is being served).

Dynamic management of Any values 4

4.1 Rationale

An **any** can be passed to a program that doesn't have any static information for the type of the **any** (code generated for the type by an IDL compiler has not been compiled with the object implementation). As a result, the object receiving the **any** does not have a portable method of using it.

The facility presented here enables traversal of the data value associated with an **any** at runtime and extraction of the primitive constituents of the data value. This is especially helpful for writing powerful generic servers (bridges, event channels supporting filtering, etc).

Similarly, this facility enables the construction of an **any** at runtime, without having static knowlege of its type. This is especially helpful for writing generic clients (bridges, browsers, debuggers, user interface tools, etc).

4.2 Overview

Any values can be dynamically interpreted (traversed) and constructed through **DynAny** objects. A **DynAny** object is associated with a data value which may correspond to a copy of the value inserted into an **any**. The **DynAny** object may be seen as owning a pointer to a external buffer which holds some representation of the data value.

A constructed **DynAny** object is a **DynAny** object associated with a constructed type. There is a different interface, inheriting from the **DynAny** interface, associated with each kind of contructed type in IDL (struct, sequence, union, or array). A constructed **DynAny** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a component of the constructed data value.

As an example, a **DynStruct** is associated with a struct value. This means that the object may be seen as owning a pointer to a external buffer which holds a representation of struct. The **DynStruct** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a member of the struct.

If a **DynAny** object has been created from another (a constructed) **DynAny** object then the buffer pointed to by the first **DynAny** object is logically contained within the buffer pointed by the second **DynAny** object.

Destroying a **DynAny** object implies deleting the buffer it points to and also destroying all **DynAny** objects obtained from it. Invoking operations using references to descendants of a destroyed **DynAny** object leads to unpredictable results. Note that releasing a reference to a **DynAny** object will not delete the buffer pointed by the object, since the object indeed exists (it has not been explicitly destroyed).

If the programmer wants to destroy a **DynAny** object but still wants to manipulate some component of the data value associated with it, then he or she should first create a **DynAny** for the component and, after that, make a copy of the created **DynAny** object.

The behaviour of **DynAny** objects has been defined in order to enable efficient implementations in terms of allocated memory space and speed of access. **DynAny** objects are intended to be used for traversing values extracted from **any**s or constructing values of **any**s at runtime. Their use for other purposes is not recommended.

4.3 DynAny API

The **DynAny** API comprises the following IDL definitions to be included in the CORBA module:

```
// IDL
interface DynAny {
    exception Invalid {};
    exception InvalidValue {};
    exception TypeMismatch {};
    exception InvalidSeq {};

    typedef sequence<octet> OctetSeq;
    TypeCode type ();

    void assign (in DynAny dyn_any) raises (Invalid);
    void from_any (in any value) raises (Invalid);
    any to_any() raises (Invalid);

    void destroy();

    DynAny copy();
```

```
    void insert_boolean(in boolean value) raises (InvalidValue);
    void insert_octet(in octet value) raises (InvalidValue);
    void insert_char(in char value) raises (InvalidValue);
    void insert_short(in short value) raises (InvalidValue);
    void insert_ushort(in unsigned short value) raises (InvalidValue);
    void insert_long(in long value) raises (InvalidValue);
    void insert_ulong(in unsigned long value) raises (InvalidValue);
    void insert_float(in float value) raises (InvalidValue);
    void insert_double(in double value) raises (InvalidValue);
    void insert_string(in string value) raises (InvalidValue);
    void insert_reference (in Object value) raises (InvalidValue);
    void insert_typecode (in TypeCode value) raises (InvalidValue);
    void insert_longlong(in long long value) raises(InvalidValue);
    void insert_ulonglong(in unsigned long long value) raises(InvalidValue);
    void insert_longdouble(in long double value) raises(InvalidValue);
    void insert_wchar(in wchar value) raises(InvalidValue);
    void insert_wstring(in wstring value) raises(InvalidValue);
    void insert_any(in any value) raises(InvalidValue);

    boolean get_boolean() raises (TypeMismatch);
    octet get_octet() raises (TypeMismatch);
    char get_char() raises (TypeMismatch);
    short get_short() raises (TypeMismatch);
    unsigned short get_ushort () raises (TypeMismatch);
    long get_long() raises (TypeMismatch);
    unsigned long get_ulong() raises (TypeMismatch);
    float get_float() raises (TypeMismatch);
    double get_double() raises (TypeMismatch);
    string get_string() raises (TypeMismatch);
    Object get_reference() raises (TypeMismatch);
    TypeCode get_typecode () raises (TypeMismatch);
    long long get_longlong() raises(TypeMismatch);
    unsigned long long get_ulonglong() raises(TypeMismatch);
    long double get_longdouble() raises(TypeMismatch);
    wchar get_wchar() raises(TypeMismatch);
    wstring get_wstring() raises(TypeMismatch);
    any get_any() raises (TypeMismatch);

    DynAny current_component ();
    boolean next ();
    boolean seek (in long index);
    void rewind ();
};

interface DynFixed : DynAny {
    OctetSeq get_value ();
    void set_value (in OctetSeq val) raises (InvalidValue);
};
```

```
interface DynEnum: DynAny {
    attribute string value_as_string;
    attribute unsigned long value_as_ulong;
};

typedef string FieldName;

struct NameValuePair {
    FieldName id;
    any value;
};

typedef sequence<NameValuePair> NameValuePairSeq;

interface DynStruct: DynAny {
    FieldName current_member_name ();
    TCKind current_member_kind ();
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
            raises (InvalidSeq);
};

interface DynUnion: DynAny {
    attribute boolean set_as_default;
    DynAny discriminator ();
    TCKind discriminator_kind ();
    DynAny member ();
    attribute FieldName member_name;
    TCKind member_kind ();
};

typedef sequence<any> AnySeq;

interface DynSequence: DynAny {
    attribute unsigned long length;
    AnySeq get_elements ();
    void set_elements (in AnySeq value)
            raises (InvalidSeq);
};

interface DynArray: DynAny {
    AnySeq get_elements();
    void set_elements(in AnySeq value)
            raises (InvalidSeq);
};
```

4.3.1 Locality and usage constraints

DynAny objects are intended to be local to the process in which they are created and used. This means that references to **DynAny** objects cannot be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a **MARSHAL** system exception.

Since their interfaces are specified in IDL, **DynAny** objects export operations defined in the standard **CORBA::Object** interface. However, any attempt to invoke operations exported through the **Object** interface may raise the standard **NO_IMPLEMENT** exception.

An attempt to use a **DynAny** object with the DII may raise the **NO_IMPLEMENT** exception.

4.3.2 Creating a DynAny object

A **DynAny** object can be created as a result of:

- invoking an operation on an existing **DynAny** object
- invoking an operation exported by the ORB

Actually, a constructed **DynAny** object support operations that enable the creation of new **DynAny** objects encapsulating access to the value of some constituent. **DynAny** objects also support the **copy** operation for creating new **DynAny** objects.

In addition, the ORB can act as a factory of **DynAny** objects in the same way as with **TypeCode** objects. Therefore, the standard **ORB** interface includes the following operations:

```
interface ORB {
    ...
    DynAny create_dyn_any (in any value);
    DynAny create_basic_dyn_any(in TypeCode type)
                raises (InconsistentTypeCode);
    DynStruct create_dyn_struct(in TypeCode type)
                raises (InconsistentTypeCode);
    DynSequence create_dyn_sequence(in TypeCode type)
                raises (InconsistentTypeCode);
    DynArray create_dyn_array(in TypeCode type)
                raises (InconsistentTypeCode);
    DynUnion create_dyn_union(in TypeCode type)
                raises (InconsistentTypeCode);
    DynEnum create_dyn_enum(in TypeCode type)
                raises (InconsistentTypeCode);
    DynFixed create_dyn_fixed(in TypeCode type)
                raises (InconsistentTypeCode);
    ...
};
```

The **create_dyn_any** operation creates a new **DynAny** object from an **any** value. A duplicate of the **TypeCode** associated with the **any** value is assigned to the resulting **DynAny** object. The value associated with the **DynAny** object is a copy of the value in the original any.

The rest of the operations used to create **DynAny** objects receive a **TypeCode** input parameter and throw the **InconsistentTypeCode** exception if the **TypeCode** passed as a parameter is not consistent with the operation.

Dynamic interpretation of an **any** usually involves creating a **DynAny** object using **create_dyn_any** as the first step. Depending on the type of the **any**, the resulting **DynAny** object reference can be narrowed to a **DynStruct**, **DynSequence**, **DynArray**, **DynUnion** or **DynEnum** object reference.

Dynamic creation of an **any** containing a struct data value typically involves creating a **DynStruct** object using **create_dyn_struct**, passing the **TypeCode** associated with the struct data value to be created. Then, components of the struct can be initialized by means of invoking operations on the resulting **DynStruct** object or **DynAny** objects generated for each member of the struct. Finally, once the data value pointed by the **DynStruct** object has been properly initialized, the **to_any** operation can be invoked. The same approach would be followed for dynamic creation of sequences, unions, etc.

Dynamic creation of an **any** containing a value of a basic data type typically involves creating a **DynAny** object using **create_basic_dyn_any**, passing the **TypeCode** associated with the basic data type value to be created. Then, the value can be initialized by means of invoking operations on the resulting **DynAny** object (**insert_boolean** if the **DynAny** is of type **boolean**, etc). Finally, the **to_any** operation can be invoked.

4.3.3 The DynAny interface

The following operations can be applied to a **DynAny** object:

- Obtaining the **TypeCode** associated with the **DynAny** object.
- Generating an **any** value from the **DynAny** object.
- Destroying the **DynAny** object.
- Creating a **DynAny** object as copy of the **DynAny** object.
- Inserting/getting a value of some basic type into/from the **DynAny** object.
- Iterating through the components of a **DynAny**.
- Obtaining the TypeCode associated to the DynAny object.
- Initializing a **DynAny** object from another **DynAny** object
- Initializing a **DynAny** object from an **any** value
- Generating an **any** value from the **DynAny** object

- Destroying the **DynAny** object
- Creating a **DynAny** object as copy of the **DynAny** object
- Inserting/Getting a value of some basic type into/from the **DynAny** object
- Iterating through the components of a **DynAny**

4.3.3.1 Obtaining the TypeCode associated with a DynAny object

A **DynAny** object is created with a **TypeCode** value assigned to it. This **TypeCode** value determines the type of the value handled through the **DynAny** object. The **type** operation returns the **TypeCode** associated with a **DynAny** object:

```
TypeCode type();
```

Note that the **TypeCode** associated with a **DynAny** object is initialized at the time the **DynAny** is created and cannot be changed during lifetime of the **DynAny** object.

4.3.3.2 Initializing a DynAny object from another DynAny object

The **assign** operation initializes the value associated to a **DynAny** object with the value associated to another **DynAny** object:

```
void assign(in DynAny dyn_any) raises(Invalid);
```

If an invalid **DynAny** object is passed (it has a different typecode or doesn't contain a meaningful value) the **Invalid** exception is returned.

4.3.3.3 Initializing a DynAny object from an any value

The **from_any** operation initializes the value associated to a **DynAny** object with the value contained in an **any**:

```
void from_any(in any value) raises(Invalid);
```

If an invalid **any** is passed (it has a different typecode or hasn't been assigned a value) the **Invalid** exception is returned.

4.3.3.4 Generating an any value from a DynAny object

The **to_any** operation creates an **any** value from a **DynAny** object:

```
any to_any() raises(Invalid);
```

If the **DynAny** object has not been correctly created or doesn't contain a meaningful value (it hasn't been properly initialized, for example) the **Invalid** exception is returned.

A duplicate of the **TypeCode** associated with the **DynAny** object is assigned to the resulting **any**. The value associated with the **DynAny** object is copied into the **any**.

4.3.3.5 Destroying a DynAny object.

The **destroy** operation destroys a **DynAny** object. This operation frees any resources used to represent the data value associated with a **DynAny** object.

```
void destroy();
```

Destruction of a **DynAny** object implies destruction of all **DynAny** objects obtained from it.

Destruction of **DynAny** objects should be handle with care taking into account issues dealing with representation of data values associated with **DynAny** objects.

If the programmer wants to destroy a **DynAny** object but still wants to manipulate some component of the data value associated with it, he or she should first create a **DynAny** for the component and then make a copy of the created **DynAny** object.

4.3.3.6 Creating a copy of a DynAny

The **copy** operation enables the creation of a new **DynAny** object whose value is a deep copy of the value pointed by the **DynAny** object:

```
DynAny copy();
```

4.3.3.7 Accessing a value of some basic type in a DynAny object

The insert and get operations have been defined to enable insertion/extraction of basic data type values into/from a **DynAny** object.

Insert operations raise the **InvalidValue** exception if the value inserted is not consistent with the type of the accessed component in the **DynAny** object.

Get operations raise the **TypeMismatch** exception if the accessed component in the **DynAny** is of a type that is not consistent with the requested type.

These operations are necessary to handle basic **DynAny** objects but are also helpful to handle constructed **DynAny** objects. Inserting a basic data type value into a constructed **DynAny** object implies initializing the next component of the constructed data value associated with the **DynAny** object. For example, invoking **insert_boolean** in a **DynStruct** implies inserting a boolean data value as the next member of the associated struct data value.

In addition, availability of these operations enable the traversal of **any**s associated with sequences of basic data types without the need to generate a **DynAny** object for each element in the sequence.

4.3.3.8 Iterating through components of a DynAny

The **DynAny** interface allows a client to iterate through the components of the struct data value pointed by a **DynStruct** object.

As mentioned above, a **DynAny** object may be seen as owning a pointer to a external buffer that holds some representation of a data value. In addition, the **DynAny** object holds a pointer to a buffer offset where the current component is being represented.

The buffer pointer effectively points to the space used to represent the first component of the data value when the programmer creates the **DynAny** object. It also points to the first component each time **rewind** is invoked.

void rewind();

The **next** operation logically advances the pointer and returns TRUE if the resulting pointer points to a component, or FALSE if there are no more components. Invoking **next** on a **DynAny** associated with a basic data type value is allowed, but it always returns FALSE.

boolean next();

The programmer is able to inspect/initialize the a component of the data value associated with the **DynAny** object by means of invoking **current_component** at each step during the iteration.

DynAny current_component();

The resulting **DynAny** object reference would be used to get/set the value of the component currently accessed. In order to get access to specific operations, the resulting **DynAny** object reference may be narrowed based on its **TypeCode**.

In order to construct an **any** associated with a sequence data value, for example, the programmer may first create the **DynAny** object invoking **create_dyn_sequence**. After doing so, the programmer may iterate through the elements of the sequence. At each step, an element in the sequence would be initialized by means of invoking **current_component** and using the returned **DynAny**. After that, **next** will be invoked. The end of the initialization process would be detected when **next** returns FALSE. At that point, the programmer would invoke **to_any** to create an **any**.

Operation **seek** logically sets a new offset for this pointer, returning TRUE if the resulting pointer points to a component or FALSE if there is no component at the designated offset. Invoking **seek** on a **DynAny** associated to a basic data type value is allowed but it only returns TRUE if the value passed as argument equals to zero.

boolean seek(in long index);

4.3.4 The DynFixed interface

DynFixed objects are associated with values of the IDL **fixed** type.

```
typedef sequence<octet> OctetSeq;
interface DynFixed : DynAny {
    OctetSeq get_value ();
    void set_value (in OctetSeq val) raises (InvalidValue);
};
```

The **get_value** operation returns the value of the **DynFixed** as a sequence of octet. Each octet contains either one or two decimal digits. If the fixed type has an odd number of decimal digits (which can be determined from the **TypeCode::fixed_digits** operation), then the representation begins with the first (most significant) digit. Otherwise, the first half-octet is all zero, and the first digit is in the second half-octet. The sign of the value, which is stored in the last half-octet of the sequence, shall be 0xD for negative numbers and 0xC for positive and zero values.

The **set_value** operation sets the value of the **DynFixed** with an **OctetSeq** having the same format as that descibed above. If the **OctetSeq** does not conform to the expected number of digits as determined by the **TypeCode**, the **InvalidValue** exception is raised.

4.3.5 The DynEnum interface

DynEnum objects are associated with enumerated values.

```
interface DynEnum: DynAny {
    attribute string value_as_string;
    attribute unsigned long value_as_ulong;
};
```

The **DynEnum** interface consists of two attributes: the **value_as_string** attribute which contains the value of the enum value as a string and the **value_as_ulong** which contains the value of the enum value as a unsigned long:

```
attribute string value_as_string;
attribute unsigned long value_as_ulong;
```

4.3.6 The DynStruct interface

DynStruct objects are associated with struct values and exception values.

```
typedef string FieldName;

struct NameValuePair {
    FieldName id;
    any value;
};

typedef sequence<NameValuePair> NameValuePairSeq;

interface DynStruct: DynAny {
    FieldName current_member_name ();
    TCKind current_member_kind ();
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
            raises (InvalidSeq);
};
```

The **current_member_name** operation returns the name of the member currently being accessed.

```
FieldName current_member_name ();
```

This operation may return an empty string since the **TypeCode** of the struct being manipulated may not contain the names of members in the struct.

current_member_kind returns the TCKind associated with the current member being accessed.

```
TCKind current_member_kind ();
```

It is possible to obtain a sequence of name/value pairs describing the name and the value of each member in the struct associated with a **DynStruct** object using the **get_members** operation:

```
NameValuePairSeq get_members();
```

The **set_members** operation initializes the struct data value associated with a **DynStruct** object from a sequence of name value pairs:

```
void set_members(in NameValuePairSeq value)
        raises (InvalidSeq);
```

Members must appear in the **NameValuePairSeq** in the order in which they appear in the IDL specification of the struct. This operation raises the **InvalidSeq** exception if an inconsistent name or value is passed as argument (for example, the **NameValuePairSeq** does not match the members of the struct, it's too long/short, or member values are passed in the wrong order).

DynStruct objects can also be used for handling exception values. In that case, members of the exceptions are handled in the same way as members of a struct.

4.3.7 The DynUnion interface

DynUnion objects are associated with unions.

```
interface DynUnion: DynAny {
    attribute boolean set_as_default;
    DynAny discriminator ();
    TCKind discriminator_kind ();
    DynAny member ();
    attribute FieldName member_name;
    TCKind member_kind ();
};
```

The **DynUnion** interface allows for the insertion/extraction of an OMG IDL union type into/from a **DynUnion** object.

The discriminator operation returns a **DynAny** object reference that must be narrowed to the type of the discriminator in order to insert/get the discriminator value:

```
DynAny discriminator ();
```

Note that the type of the discriminator is contained in the **TypeCode** of the union.

The **member** operation returns a **DynAny** object reference that is used in order to insert/get the member of the union:

```
DynAny member ();
```

discriminator_kind and **member_kind** return the TCKind associated with the discriminator and member of the union, respectively:

```
    TCKind discriminator_kind ();
    TCKind member_kind ();
```

The **member_name** attribute allows for the inspection/assignment of the name of the union member without checking the value of the discriminator.

The **set_as_default** attribute determines whether the discriminator associated with the union has been assigned a valid default value.

Union values can be traversed using the operations defined in section 4.3.3.8. In that case, the first component in the union corresponds to the discriminator while the second corresponds to the actual value of the union. Operation **next** should then be called twice.

4.3.8 The DynSequence interface

DynSequence objects are associated with sequences.

```
typedef sequence<any> AnySeq;

interface DynSequence: DynAny {
    attribute unsigned long length;
    AnySeq get_elements();
    void set_elements(in AnySeq value)
            raises (InvalidSeq);
};
```

The **length** attribute contains the number of elements contained in (or to be contained in) the sequence; its value is initalized to zero for unbounded sequences:

```
attribute unsigned long length;
```

The **get_elements** and **set_elements** operations return and receive respectively a sequence of **any**s containing each of the elements of the sequence:

```
AnySeq get_elements();
void set_elements(in AnySeq value);
```

The **set_elements** operation raises the **InvalidSeq** exception if an inconsistent value is passed in the sequence of **any** values passed as argument (for example, the **AnySeq** is too long/short).

4.3.9 The DynArray interface

DynArray objects are associated with arrays.

```
interface DynArray: DynAny {
    AnySeq get_elements();
    void set_elements(in AnySeq value)
            raises (InvalidSeq);
};
```

The **get_elements** and **set_elements** operations return and receive respectively a sequence of **any**s containing each of the elements of the array:

```
AnySeq get_elements();
void set_elements(in AnySeq value);
```

The **set_elements** operation raises the **InvalidSeq** exception if an inconsistent value is passed in the sequence of **any** values passed as argument (for example, the **AnySeq** is too long/short).

Note that the dimension of the array is contained in the **TypeCode** which is accessible through the **type** attribute.

4.4 Usage in C++ language

4.4.1 Dynamic creation of CORBA::Any values

4.4.1.1 Creating an any which contains a struct

Consider the following IDL definition:

```
// IDL
struct MyStruct {
    long member1;
    boolean member2;
};
```

The following example illustrates how a **`CORBA::Any`** value may be constructed on the fly containing a value of type **MyStruct**:

```
// C++
CORBA::ORB_var orb;
CORBA::StructMemberSeq mems(2);
CORBA::Any result;
long value1;
boolean value2;

mems[0].name = CORBA::string_dup("member1");
mems[1].type = CORBA::TypeCode::_duplicate(CORBA::_tc_long);
mems[0].name = CORBA::string_dup("member2");
mems[1].type =
   CORBA::TypeCode::_duplicate(CORBA::_tc_boolean);

CORBA::TypeCode_var new_tc = orb->create_struct_tc (
                               "IDL:MyStruct:1.0",
                               "MyStruct",
                               mems
                             );

// construct the DynStruct object. Values for members have
// read in the value1 and value2 variables

DynStruct_ptr dyn_struct = orb->create_dyn_struct (new_tc);
dyn_struct->insert_long(value1);
dyn_struct->insert_boolean(value2);
result = dyn_struct->to_any();
dyn_struct->destroy ();
CORBA::release(dyn_struct);
```

4.4.2 Dynamic interpretation of CORBA::Any values

4.4.2.1 Filtering of events

Suppose there is a CORBA object which receives events and print all those events which correspond to a data structure containing a member called **is_urgent** whose value is TRUE.

The following fragment of code corresponds to a method which determines if an event should be printed or not. Note that the program allows several struct events to be filtered with respect to some common member.

```
// C++
CORBA::Boolean Tester::eval_filter(const CORBA::Any &event)
{
   CORBA::Boolean success = FALSE;

   // First, typecode is extracted from the event.  This
   // is necessary to get struct member names:
   CORBA::TypeCode_var event_type = event->type();

   // The filter only returns true if the event is a struct:
   if (event_type->kind() == CORBA::tk_struct)
   {
      DynAny_ptr dyn_any = orb->create_dyn_any(event);
      DynStruct_ptr dyn_struct= DynStruct::_narrow(dyn_any);
      CORBA::release(dyn_any);

      CORBA::Boolean found = FALSE;

      do
      {
         CORBA::String_var member_name =
                     dyn_struct->current_member_name();

         found = (strcmp(member_name, "is_urgent") == 0);
      } while (!found && !dyn_struct->next());


      if (found)
      {
         // We only create a DynAny object for the member
         // we were looking for:
         CORBA::DynAny_var dyn_member =
                     dyn_struct->current_component ();
         success = dyn_member->get_boolean();
      };

      dyn_struct->destroy();
      CORBA::release(dyn_struct);
   };

   return success;
};
```

Cobol Mapping 5

This chapter is in a separate document because of different document preparation systems used. That separate chapter should replace the previously approved Cobol Mapping. The changes include only those to handle the POA and other portability issues.

C Language Mapping 14

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

14.1 Requirements for a Language Mapping

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

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

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

14.1.1 Basic Data Types

A language mapping must define the means of expressing all of the data types defined in Section 3.8.1, “Basic Types,” on page 3-20. The ORB defines the range of values supported, but the language mapping defines how a programmer sees those values. For example, the C mapping might define TRUE as 1 and FALSE as 0, whereas the LISP mapping might define TRUE as T and FALSE as NIL. The mapping must specify the means to construct and operate on these data types in the programming language.

14.1.2 Constructed Data Types

A language mapping must define the means of expressing the constructed data types defined in Section 3.8.2, “Constructed Types,” on page 3-22. The ORB defines aggregates of basic data types that are supported, but the language mapping defines how a programmer sees those aggregates. For example, the C mapping might define an OMG IDL struct as a C struct, whereas the LISP mapping might define an OMG IDL struct as a list. The mapping must specify the means to construct and operate on these data types in the programming language.

14.1.3 Constants

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

14.1.4 Objects

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

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

14.1.5 Invocation of Operations

An operation invocation requires the specification of the object to be invoked, the operation to be performed, and the parameters to be supplied. There are a variety of possible mappings, depending to a large extent on the procedure mechanism in the particular language. Some possible choices for language mapping of invocation include:

interface-specific stub routines, a single general-purpose routine, a set of calls to construct a parameter list and initiate the operation, or mapping ORB operations to operations on objects defined in an object-oriented programming language.

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

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

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

14.1.6 Exceptions

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

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

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

14.1.7 Attributes

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

14.1.8 ORB Interfaces

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

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

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

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

14.2 Scoped Names

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

Consider the following example:

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

Code to use this interface would look as follows:

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

filename_t FN;
example0_color C = example0_red;
example0_bar myUnion;

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

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

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

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

14.3 Mapping for Interfaces

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

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

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

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

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

1. Section 14.15, "Implicit Arguments to Operations," on page 14-17 describes the additional arguments added to an operation in the C mapping.

name of the interface is defined to be a **CORBA_Object**. The literal

CORBA_OBJECT_NIL is legal wherever a **CORBA_Object** may be used; it is guaranteed to pass the **is_nil** operation defined in Section 7.2.3, “Nil Object References,” on page 7-4.

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

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

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

This is equivalent to the following C declaration:

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

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

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

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

example1 ex1;
example2 ex2;
CORBA_Environment ev;

/* code for binding ex2 */

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

14.4 Inheritance and Operation Names

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

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

This is equivalent to the following C declarations:

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

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

14.5 Mapping for Attributes

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

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

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

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

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

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

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

The language mapping for attributes then becomes the language mapping for these equiv-

alent operations. More specifically, the function signatures generated for the above operations are as follows:

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

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

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

If the "**set**" accessor function fails to set the attribute value, the method should return one of the standard exceptions defined in Section 3.15, "Standard Exceptions," on page 3-33.

14.6 Mapping for Constants

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

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

14.7 Mapping for Basic Data Types

The basic data types have the mappings shown in Table 1 on page 14-8. Implementations are responsible for providing typedefs for CORBA_short, CORBA_long, and so forth. consistent with OMG IDL requirements for the corresponding data types.

Table 1: Data Type Mappings

OMG IDL	C
short	CORBA_short
long	CORBA_long
unsigned short	CORBA_unsigned_short
unsigned long	CORBA_unsigned_long
float	CORBA_float
double	CORBA_double

Table 1: Data Type Mappings *(Continued)*

OMG IDL	C
char	CORBA_char
boolean	CORBA_boolean
octet	CORBA_octet
enum	CORBA_enum
any	typedef struct CORBA_any { CORBA_TypeCode _type; void *_value; } CORBA_any;

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

The C mapping of OMG IDL **enum** types is an unsigned integer type capable of representing 2^{32} enumerations. Each enumerator in an **enum** is **#define**d with an appropriate unsigned integer value conforming to the ordering constraints described in "Enumerations" on page 3-24.

TypeCodes are described in Section 6.7, "TypeCodes," on page 6-33. The **_value** member for an **any** is a pointer to the actual value of the datum.

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

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

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

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

14.8 Mapping Considerations for Constructed Types

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

- The type **any**
- A bounded or unbounded string
- A bounded or unbounded sequence
- An object reference or reference to a transmissible pseudo-object[2]
- A struct or union that contains a member whose type is variable-length
- An array with a variable-length element type
- A typedef to a variable-length type

The reason for treating fixed- and variable-length data structures differently is to allow more flexibility in the allocation of **out** parameters and return values from an operation. This flexibility allows a client-side stub for an operation that returns a sequence of strings, for example, to allocate all the string storage in one area that is deallocated in a single call. The mapping of a variable-length type as an **out** parameter or operation return value is a pointer to the associated class or array, as shown in Table 2 on page 14-20.

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

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

The functions are defined at global scope using the fully-scoped name of T converted into a C language name (as described in Section 14.2) followed by the suffix "__alloc" (note the double underscore). For **any** and **string**, the allocation functions are:

```
/* C */
CORBA_any *CORBA_any_alloc();
char *CORBA_string_alloc();
```

respectively.

14.9 Mapping for Structure Types

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

2.Transmissible pseudo-objects are listed as "general arguments" in Table 14 on page A-2 .

14.10 Mapping for Union Types

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

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

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

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

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

Reference to union elements is as in normal C:

```
/* C */
Foo *v;

/* make a call that returns a pointer to a Foo in v */

switch(v->_d) {
   case 1:  printf("x = %ld\n", v->_u.x); break;
   case 2:  printf("y = %f\n", v->_u.y); break;
   default: printf("z = %c\n", v->_u.z); break;
}
```

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

14.11 Mapping for Sequence Types

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

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

In C, this is converted to:

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

An instance of this type is declared as follows:

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

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

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

Prior to passing **&x** as an **inout** parameter, the programmer must set the **_buffer** member to point to a **CORBA_long** array of 10 elements. The **_length** member must be set to the actual number of elements to transmit. Upon successful return from the invocation, the **_length** member will contain the number of values that were copied into the buffer pointed to by the **_buffer** member. If more data must be returned than the original buffer can hold, the callee can deallocate the original **_buffer** member using **CORBA_free()** (honoring the release flag) and assign **_buffer** to point to new storage.

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

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

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

CORBA_sequence_set_release can be used to set the state of the release flag.

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

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

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

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

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

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

The OMG IDL type

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

maps to

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

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

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

the sequence is mapped onto **`struct { ... } CORBA_sequence_long;`**

If the **type** in

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

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

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

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

generates a type of

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

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

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

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

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

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

for example, the sequence buffer allocation function is named

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

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

14.12 Mapping for Strings

OMG IDL strings are mapped to 0-byte terminated character arrays; i.e. the length of the string is encoded in the character array itself through the placement of the 0-byte. Note that the storage for C strings is one byte longer than the stated OMG IDL bound. Consider the following OMG IDL declarations:

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

In C, this is converted to:

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

Instances of these types are declared as follows:

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

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

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

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

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

Prior to passing **s1** or **s2** as an **in** parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated string to the variable. The caller cannot pass a null pointer as the string argument.

Prior to passing **&s1** or **&s2** as an **out** parameter (or receiving an **sten** or **sinf** as the return result), the programmer does nothing. The client stub will allocate storage for the returned buffer; for bounded strings, it allocates a buffer of the specified size, while for unbounded strings, it allocates a buffer big enough to hold the returned string. Upon successful return from the invocation, the character pointer will contain the address of the allocated buffer. The client is responsible for freeing the allocated storage using **CORBA_free()**.

Prior to passing **&s1** or **&s2** as an **inout** parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated array to the variable. If the returned string is larger than the original buffer, the client stub will call **CORBA_free()** on the original string and allocate a new buffer for the new string. The client should therefore never pass an **inout** string parameter that was not allocated using **CORBA_string_alloc**. The client is responsible for freeing the allocated storage using **CORBA_free()**, regardless of whether or not a reallocation was necessary.

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

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

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

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

14.13 Mapping for Arrays

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

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

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

The C mapping provides the following definitions:

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

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

If the return result, or an **out** parameter for an array holding a variable-length type, of an operation is an array, the array storage is dynamically allocated by the stub; a pointer to the array slice of the dynamically allocated array is returned as the value of the client stub function. When the data is no longer needed, it is the programmer's responsibility to return the dynamically allocated storage by calling **CORBA_free()**.

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

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

This function is identical to the allocation functions described in Section 14.8, "Mapping Considerations for Constructed Types," on page 14-10, except that the return type is pointer to array slice, not pointer to array.

14.14 Mapping for Exception Types

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

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

yields the following C declarations:

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

The identifier for the exception uniquely identifies this exception type. For example, it could be the Interface Repository identifier for the exception (see Section 6.5.19, "ExceptionDef," on page 6-23).

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

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

14.15 Implicit Arguments to Operations

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

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

As described above, the **`CORBA_Object`** type is an opaque type. The **`CORBA_Environment`** type is partially opaque; Section 14.20 provides a description of the non-opaque portion of the exception structure and an example of how to handle exceptions in client code. The **`CORBA_Context`** type is opaque; see the Dynamic Invocation Interface chapter for more information on how to create and manipulate context objects.

14.16 Interpretation of Functions with Empty Argument Lists

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

14.17 Argument Passing Considerations

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

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

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

Consider the following OMG IDL specification:

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

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

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

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

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

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

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

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

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

14.18 Return Result Passing Considerations

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

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

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

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

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

Consider the following interface:

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

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

The following C declarations ensue from processing the specification:

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

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

For operation results of type variable-length **struct**, variable-length **union**, **string**, **sequence**, **array**, or **any**, the ORB will allocate storage for the return value using the appropriate type-specific allocation function. The client may use and retain that storage indefinitely, and must indicate when the value is no longer needed by calling the procedure **`CORBA_free()`** described in Section 14.17, "Argument Passing Considerations.

14.19 Summary of Argument/Result Passing

Table 2 on page 14-20 summarizes what a client passes as an argument to a stub and receives as a result.[3]

Table 2: Basic Argument and Result Passing

Data Type	In	Inout	Out	Return
short	short	short*	short*	short
long	long	long*	long*	long
unsigned short	unsigned_short	unsigned_short*	unsigned_short*	unsigned_short
unsigned long	unsigned_long	unsigned_long*	unsigned_long*	unsigned_long
float	float	float*	float*	float
double	double	double*	double*	double
boolean	boolean	boolean*	boolean*	boolean
char	char	char*	char*	char
octet	octet	octet*	octet*	octet
enum	enum	enum*	enum*	enum
object reference ptr[1]	objref_ptr	objref_ptr*	objref_ptr*	objref_ptr
struct, fixed	struct*	struct*	struct*	struct
struct, variable	struct*	struct*	struct**	struct*
union, fixed	union*	union*	union*	union
union, variable	union*	union*	union**	union*
string	char*	char**	char**	char*
sequence	sequence*	sequence*	sequence**	sequence*
array, fixed	array	array	array	array slice*[2]
array, variable	array	array	array slice**[2]	array slice*[2]
any	any*	any*	any**	any*

1. Including pseudo-object references.

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

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

Table 3: Client Argument Storage Responsibilities

Type	Inout Param	Out Param	Return Result
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enum	1	1	1
object reference ptr	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3

Table 4: Argument Passing Cases

Case[1]	
1	Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller provides the initial value, and the callee may change that value. For out parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.
2	Caller allocates storage for the object reference. For inout parameters, the caller provides an initial value; if the callee wants to reassign the inout parameter, it will first call CORBA_Object_release on the original input value. To continue to use an object reference passed in as an inout, the caller must first duplicate the reference. The client is responsible for the release of all out and return object references. Release of all object references embedded in other out and return structures is performed automatically as a result of calling CORBA_free.

3.For brevity, the “CORBA_” prefix is omitted from type names in the tables shown here.

Table 4: Argument Passing Cases *(Continued)*

Case[1]	
3	For out parameters, the caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned instance into a new instance, then modify the new instance.
4	For inout strings, the caller provides storage for both the input string and the char* pointing to it. The callee may deallocate the input string and reassign the char* to point to new storage to hold the output value. The size of the out string is therefore not limited by the size of the in string. The caller is responsible for freeing the storage for the out.The callee is not allowed to return a null pointer for an inout, out, or return value.
5	For inout sequences and anys, assignment or modification of the sequence or any may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the boolean release in the sequence or any.
6	For out parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance.

1. As listed in Table 3 on page 14-21

14.20 Handling Exceptions

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

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

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

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

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

CORBA_exception_set() allows a method implementation to raise an exception. The **ev** parameter is the environment parameter passed into the method. The caller must supply a value for the major parameter. The value of the major parameter constrains the other parameters in the call as follows:

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

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

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

CORBA_exception_value() returns a pointer to the structure corresponding to

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

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

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

Consider the following example:

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

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

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

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

CORBA_Environment ev;
exampleX obj;
exampleX_BadCall *bc;

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

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

14.21 Method Routine Signatures

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

Most object adapters are likely to provide method signatures that are similar in most

respects to those of the client stubs. In particular, the mapping for the operation parameters expressed in OMG IDL should be the same as for the client side.

See Section 14.24, “Mapping for Object Implementations,” on page 14-27 for the description of method signatures for implementations using the Portable Object Adapter.

14.22 Include Files

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

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

14.23 Pseudo-objects

In the C language mapping, there are several interfaces that are defined as pseudo-objects; Table 14 on page A-2 lists the pseudo-objects. A client makes calls on a pseudo-object in the same way as an ordinary ORB object. However, the ORB may implement the pseudo-object directly, and there are restrictions on what a client may do with a pseudo-object.

The ORB itself is a pseudo-object with the following partial definition (see Chapter 7, “ORB Interface" for the complete definition):

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

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

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

just as if the ORB were an ordinary object. The C library contains the routine **`CORBA_ORB_object_to_string`**, and it does not do a real invocation. The **`orbobj`** is an object reference that specifies which ORB is of interest, since it is possible to choose which ORB should be used to convert an object reference to a string (see Chapter 7, “ORB Interface" for details on this specific operation).

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

face repository.

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

14.23.1 ORB Operations

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

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

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

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

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

14.24 Mapping for Object Implementations

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

14.24.1 Operation-specific Details

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

14.24.2 PortableServer Functions

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

```
/* C */
extern char* PortableServer_ObjectId_to_string(
                    PortableServer_ObjectId* id,
                    CORBA_Environment* env
                );
extern wchar_t* PortableServer_ObjectId_to_wstring(
                    PortableServer_ObjectId* id
                    CORBA_Environment* env
                );

extern PortableServer_ObjectId*
                PortableServer_string_to_ObjectId(
                        char* str,
                        CORBA_Environment* env
                );
extern PortableServer_ObjectId*
                PortableServer_wstring_to_ObjectId(
                        wchar_t* str,
                        CORBA_Environment* env
                );
```

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

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

14.24.3 Mapping for PortableServer::ServantLocator::Cookie

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

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

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

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

14.24.4 Servant Mapping

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

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

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

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

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

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

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

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

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

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

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

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

14.24.5 Interface Skeletons

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

For example, consider the following IDL interface:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14.24.6 Servant Structure Initialization

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

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

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

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

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

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

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

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

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

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

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

14.24.7 Application Servants

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

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

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

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

The application could initialize the servant statically as follows:

```
/* C */
PortableServer_ServantBase__epv base_epv = {
   NULL,          /* ignore ORB private data */
   NULL,          /* no servant-specific finalize
                     function needed */
   NULL,          /* use base default_POA function */
};
POA_Counter__epv counter_epv = {
   NULL,             /* ignore ORB private data */
   app_servant_add   /* point to our add function */
};
/* Vector of EPVs */
POA_Counter__vepv counter_vepv = {
   &base_epv,
   &counter_epv
};
AppServant my_servant = {
   /* initialize POA_Counter */
   {
      NULL,          /* ignore ORB private data */
      &counter_vepv  /* Counter vector of EPVs */
   },
   0           /* initialize counter value */
};
```

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

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

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

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

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

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

14.24.8 Method Signatures

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

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

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

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

The **_servant** parameter is the pointer to the servant incarnating the CORBA object on which the request was invoked. The method can obtain the object reference for the target CORBA object by using the **POA_Current** object. The **_env** parameter is used for raising exceptions. Note that the names of the **_servant** and **_env** parameters are standardized to allow the bodies of method functions to refer to them portably.

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

The method terminates with an error by executing the **CORBA_exception_set** operation (described in "Handling Exceptions" on page 14-22) prior to executing a **return** statement. When raising an exception, the method code is *not* required to assign legal values to any **out** or **inout** parameters. Due to restrictions in C, however, it must return a legal function value.

14.25 Mapping of the Dynamic Skeleton Interface to C

For general information about mapping of the Dynamic Skeleton Interface to programming languages, refer to Section 5.3, "Dynamic Skeleton Interface: Language Mapping," on page 5-3.

This section contains

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

14.25.1 Mapping of ServerRequest to C

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14.25.2 Mapping of Dynamic Implementation Routine to C

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

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

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

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

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

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

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

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

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

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

As for other servants, initialization and finalization functions for **PortableServer_DynamicImpl** are also provided, and must be invoked as

described in “Servant Structure Initialization” on page 14-32.

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

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

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

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

```
/* C */

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

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

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

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

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

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

14.26 ORB Initialization Operations

ORB Initialization

The following PIDL specifies initialization operations for an ORB; this PIDL is part of the CORBA module (not the ORB interface) and is described in Section 7.4, "ORB Initialization," on page 7-6.

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

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

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

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

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

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

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

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

For C, the order of consumption of **`argv`** parameters may be significant to an application. In order to ensure that applications are not required to handle **`argv`** parameters they do not recognize the ORB initialization function must be called before the remainder of the parameters are consumed. Therefore, after the **`ORB_init`** call the **`argv`** and **`argc`** parameters will have been modified to remove the ORB understood arguments. It is important to note that the ORB_init call can only reorder or remove references to parameters from the argv list; this restriction is made in order to

avoid potential memory management problems caused by trying to free parts of the argv list or extending the argv list of parameters. This is why **`argv`** is passed as a **`char**`** and not a **`char***`**.

14.27 Operations for Obtaining Initial Object References

The following PIDL specifies the operations (in the ORB interface) that allow applications to get pseudo object references for the Interface Repository and Object Services. It is described in detail in Section 7.6, "Obtaining Initial Object References," on page 7-10.

```
// PIDL
module CORBA {
    interface ORB {
        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;

        exception InvalidName {};

        ObjectIdList list_initial_services ();

        Object resolve_initial_references (in ObjectId identifier)
            raises (InvalidName);
    };
};
```

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

```
/* C */
typedef char* CORBA_ORB_ObjectId;
typedef CORBA_sequence_CORBA_ORB_ObjectId
                        CORBA_ORB_ObjectIdList;
typedef struct CORBA_ORB_InvalidName {
    ...
} CORBA_ORB_InvalidName;

CORBA_ORB_ObjectIdList CORBA_ORB_list_initial_services (
                              CORBA_ORB orb,
                              CORBA_Environment *env);

CORBA_Object CORBA_ORB_resolve_initial_references (
                        CORBA_ORB orb,
                        CORBA_ORB_ObjectId identifier,
                        CORBA_Environment *env);
```

Mapping of OMG IDL to C++ 16

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

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

16.1 Preliminary Information

16.1.1 Scoped Names

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

- OMG IDL modules are mapped to C++ namespaces.
- OMG IDL interfaces are mapped to C++ classes (as described in Section 16.3, “Mapping for Interfaces,” on page 16-3).
- All OMG IDL constructs scoped to an interface are accessed via C++ scoped names. For example, if a type **mode** were defined in interface **printer** then the type would be referred to as **`printer::mode`**.

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

```
// IDL
module M
{
    struct E {
```

```
        long L;
    };
};
```

is mapped into:

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

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

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

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

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

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

16.1.2 C++ Type Size Requirements

1 The sizes of the C++ types used to represent OMG IDL types are implementation-dependent. That is, this mapping makes no requirements as to the **sizeof(T)** for anything except basic types (see Section 16.5, "Mapping for Basic Data Types," on page 16-11) and string (see Section 16.7, "Mapping For String Types," on page 16-13).

16.1.3 CORBA Module

1 The mapping relies on some predefined types, classes, and functions that are logically defined in a module named **CORBA.** The module is automatically accessible from a C++ compilation unit that includes a header file generated from an OMG IDL specification. In the examples presented in this document, CORBA definitions are

referenced without explicit qualification for simplicity. In practice, fully scoped names or C++ **using** statements for the **CORBA** namespace would be required in the application source. Appendix A contains standard OMG IDL types.

16.2 Mapping for Modules

1 A shown in Section 16.1.1, "Scoped Names," on page 16-1, a module defines a scope, and as such is mapped to a C++ **namespace** with the same name:

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

// C++
namespace M
{
     // definitions
}
```

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

16.3 Mapping for Interfaces

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

2 A CORBA–C++–compliant program cannot

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

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

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

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

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

16.3.1 Object Reference Types

4 The use of an interface type in OMG IDL denotes an object reference. Because of the different ways an object reference can be used and the different possible implementations in C++, an object reference maps to two C++ types. For an interface **A**, these types are named **A_var** and **A_ptr**. For historical reasons, the type **ARef** is defined as a synonym for **A_ptr**, but usage of the **Ref** names is not portable and is thus deprecated. These types need not be distinct—**A_var** may be identical to **A_ptr**, for example—so a compliant program cannot overload operations using these types solely.

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

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

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

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

16.3.2 Widening Object References

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

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

9 Implicit widening from a **B_var** to **A_var** or **Object_var** need not be supported; instead, widening between **_var** types for object references requires a call to **_duplicate** (described in Section 16.3.3, "Object Reference Operations," on page 16-5).[1] An attempt to implicitly widen from one **_var** type to another must cause a compile-time error.[2] Assignment between two **_var** objects of the same type is supported, but widening assignments are not and must cause a compile-time error. Widening assignments may be done using **_duplicate**.

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

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

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

16.3.3 Object Reference Operations

10 Conceptually, the **Object** class in the **CORBA** module is the base interface type for all CORBA objects. Any object reference can therefore be widened to the type

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

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

Object_ptr. As with other interfaces, the CORBA namespace also defines the type **Object_var**.

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

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

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

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

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

```
// IDL
interface A { };

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

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

16.3.4 Narrowing Object References

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

inherits from A. If an object reference to a C object is widened to an **A_ptr** variable called **ap**, the

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

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

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

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

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

16.3.5 Nil Object Reference

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

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

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

22 As described in Section 16.3.1 on page 4, object references may not be compared using **operator==**, so **is_nil** is the only compliant way an object reference can be checked to see if it is nil.

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

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

16.3.6 Object Reference Out Parameter

25 When a **_var** is passed as an **out** parameter, any previous value it refers to must be implicitly released. To give C++ mapping implementations enough hooks to meet this requirement, each object reference type results in the generation of an **_out** type which is used solely as the **out** parameter type. For example, interface **A** results in the object

reference type **A_ptr**, the helper type **A_var**, and the **out** parameter type **A_out**. The general form for object reference **_out** types is shown below.

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

  private:
     A_ptr& ptr_;
};
```

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

16.3.7 Interface Mapping Example

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

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

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

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

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

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

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

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

     A_ptr in() const { return ptr_; }
     A_ptr& inout() { return ptr_; }
```

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

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

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

  private:
    // hidden assignment operators for var types to
    // fulfill the rules specified in
    // Section 16.3.2 on page 5
    void operator=(const _var &);
};
```

28 The definition for the **A_out** type is the same as the one shown in Section 16.3.6 on page 7.

16.4 Mapping for Constants

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

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

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

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

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

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

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

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

16.5 Mapping for Basic Data Types

1 The basic data types have the mappings shown in Table 1 on page 16-11. Note that the mapping of the OMG IDL **boolean** type defines only the values 1 (TRUE) and 0 (FALSE); other values produce undefined behavior.

TABLE 1. Basic Data Type Mappings

OMG IDL	C++	C++ Out Type
short	CORBA::Short	CORBA::Short_out
long	CORBA::Long	CORBA::Long_out
unsigned short	CORBA::UShort	CORBA::UShort_out
unsigned long	CORBA::ULong	CORBA::ULong_out
float	CORBA::Float	CORBA::Float_out
double	CORBA::Double	CORBA::Double_out
char	CORBA::Char	CORBA::Char_out

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

TABLE 1. Basic Data Type Mappings *(Continued)*

OMG IDL	C++	C++ Out Type
boolean	CORBA::Boolean	CORBA::Boolean_out
octet	CORBA::Octet	CORBA::Octet

2 Each OMG IDL basic type is mapped to a typedef in the CORBA module. This is because some types, such as **short** and **long**, may have different representations on different platforms, and the CORBA definitions will reflect the appropriate representation. For example, on a 64-bit machine where a long integer is 64 bits, the definition of **CORBA::Long** would still refer to a 32-bit integer. Requirements for the sizes of basic types are shown in Section 3.8.1, "Basic Types," on page 3-20.

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

4 The **_out** types for the basic types are used to type **out** parameters within operation signatures, as described in Section 16.18.1, "Operation Parameters and Signatures," on page 16-54. For the basic types, each **_out** type shall be a **typedef** to a reference to the corresponding C++ type. For example, the **Short_out** shall be defined in the **CORBA** namespace as follows:

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

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

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

7 For the **Boolean** type, only the values 1 (representing **TRUE**) and 0 (representing **FALSE**) are defined; other values produce undefined behavior. Since many existing C++ software packages and libraries already provide their own preprocessor macro definitions of **TRUE** and **FALSE**, this mapping does not require that such definitions be provided by a compliant implementation. Requiring definitions for **TRUE** and **FALSE** could cause compilation problems for CORBA applications that make use of such packages and libraries. Instead, we recommend that compliant applications simply use the values 1 and 0 directly.[4] Alternatively, for those C++ compilers that support the new **bool** type, the keywords **true** and **false** may be used.

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

16.6 Mapping for Enums

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

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

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

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

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

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

16.7 Mapping For String Types

1 As in the C mapping, the OMG IDL string type, whether bounded or unbounded, is mapped to **char*** in C++. String data is null-terminated. In addition, the **CORBA** module defines a class **String_var** that contains a **char*** value and automatically frees the pointer when a **String_var** object is deallocated. When a **String_var** is constructed or assigned from a **char***, the **char*** is consumed and thus the string data may no longer be accessed through it by the caller. Assignment or construction from a **const char*** or from another **String_var** causes a copy. The **String_var** class also provides operations to convert to and from **char*** values, as well as subscripting operations to access characters within the string. The full definition of the **String_var** interface is given in “String_var Class” on page C-2.

2 Because its mapping is **char***, the OMG IDL string type is the only non-basic type for which this mapping makes size requirements.

3 For dynamic allocation of strings, compliant programs must use the following functions from the **CORBA** namespace:

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

4 The **string_alloc** function dynamically allocates a string, or returns a null pointer if it cannot perform the allocation. It allocates **len+1** characters so that the resulting string has enough space to hold a trailing NUL character. The **string_dup** function dynamically allocates enough space to hold a copy of its string argument, including the NUL character, copies its string argument into that memory, and returns a pointer to the new string. If allocation fails, a null pointer is returned. The **string_free** function deallocates a string that was allocated with **string_alloc** or **string_dup**. Passing a null pointer to **string_free** is acceptable and results in no action being performed. These functions allow ORB implementations to use special memory management mechanisms for strings if necessary, without forcing them to replace global **operator new** and **operator new[]**.

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

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

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

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

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

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

```
    }
    String_out& operator=(const char* p) {
        ptr_ = string_dup(p); return *this;
    }
    operator char*&() { return ptr_; }
    char*& ptr() { return ptr_; }

  private:
    char*& ptr_;

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

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

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

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

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

16.8 Mapping for Structured Types

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

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

- The type **any**
- A bounded or unbounded string
- A bounded or unbounded sequence
- An object reference or reference to a transmissible pseudo-object[5]
- A struct or union that contains a member whose type is variable-length
- An array with a variable-length element type
- A typedef to a variable-length type

3 The reason for treating fixed- and variable-length data structures differently is to allow more flexibility in the allocation of **out** parameters and return values from an operation. This flexibility allows a client-side stub for an operation that returns a sequence of strings, for example, to allocate all the string storage in one area that is deallocated in a single call.

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

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

5. Transmissible pseudo-objects are listed as "general arguments" in Section TBL. 14, "Pseudo-objects," on page A-2.

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

16.8.1 T_var Types

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

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

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

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

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

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

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

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

8 The copy constructor deep-copies any data pointed to by the **T_var** constructor parameter. This copy will be destroyed when the **T_var** is destroyed or when a new value

is assigned to it. Compliant implementations may, but are not required to, utilize some form of reference counting to avoid such copies.

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

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

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

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

13 In addition to the member functions described above, the **T_var** types must support conversion functions that allow them to fully support the parameter passing modes shown in Table 2 on page 16-54. The form of these conversion functions is not specified so as to allow different implementations, but the conversions must be automatic (i.e., they must require no explicit application code to invoke them).

14 Because implicit conversions can sometimes cause problems with some C++ compilers and with code readability, the **T_var** types also support member functions that allow them to be explicitly converted for purposes of parameter passing. To pass a **T_var** as an **in** parameter, an application can call the **in()** member function of the **T_var**; for **inout** parameters, the **inout()** member function; for **out** parameters, the **out()** member function; and to obtain a return value from the **T_var**, the **_retn()** function.[6] For each **T_var** type, the return types of each of these functions shall match the types shown in Table 2 on page 16-54 for the **in**, **inout**, **out**, and return modes for underlying type **T** respectively.

15 For **T_var** types that return **T*&** from the **out()** member function, the **out()** member function calls **delete** on the **T*** owned by the **T_var**, sets it equal to the null pointer, and then returns a reference to it. This is to allow for proper management of the **T*** owned by a **T_var** when passed as an **out** parameter, as described in Section 16.18 on page 51. An example implementation of such an **out()** function is shown below:

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

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

16 Similarly, for **T_var** types whose corresponding type **T** is returned from IDL operations as **T*** (see Table 2 on page 16-54), the **_retn()** function stores the value of the **T*** owned by the **T_var** into a temporary pointer, sets the **T*** to the null pointer value, and then returns the temporary. The **T_var** thus yields ownership of its **T*** to the caller of **_retn()** without calling **delete** on it, and the caller becomes responsible for eventually deleting the returned **T***. An example implementation of such a **_retn()** function is shown below:

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

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

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

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

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

20 **T_var** types do not work with a pointer to constant **T**, since they provide no constructor nor **operator=** taking a **const T*** parameter. Since C++ does not allow **delete** to be called on a **const T***, the **T_var** object would normally have to copy the const

object; instead, the absence of the **const T*** constructor and assignment operators will result in a compile-time error if such an initialization or assignment is attempted. This allows the application developer to decide if a copy is really wanted or not. Explicit copying of **const T*** objects into **T_var** types can be achieved via the copy constructor for **T**:

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

16.8.2 T_out Types

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

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

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

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

  private:
     T*& ptr_;

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

22 The first constructor binds the reference data member with the **T*&** argument and sets the pointer to the null pointer value. The second constructor binds the reference data member with the pointer held by the **T_var** argument, and then calls **delete** on the pointer (or **string_free()** in the case of the **String_out** type or **T_free()** in the case of a

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

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

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

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

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

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

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

16.9 Mapping for Struct Types

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

2 For a string or object reference member, the name of the C++ member's type is not specified by the mapping—a compliant program therefore cannot create an object of that type. The behavior[7] of the type is the same as the normal mapping (**char*** for string,

A_ptr for an interface A) except the type's copy constructor copies the member's storage and its assignment operator releases the member's old storage. These types must also provide the **in()**, **inout()**, **out()**, and **_retn()** functions that their corresponding **T_var** types provide to allow them to support the parameter passing modes specified in Table 2 on page 16-54.

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

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

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

// C++
Fixed x1 = {1.2, 2.4, 3.6};
Fixed_var x2 = new Fixed;
x2->y = x1.z;
```

5 The example above shows usage of the **T** and **T_var** types for a fixed-length struct. When it goes out of scope, **x2** will automatically free the heap-allocated **Fixed**[8] object using **delete**.

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

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

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

8. This type is not to be confused with the IDL **fixed** type.

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

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

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

8 Continuing with the example:

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

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

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

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

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

11 On line 7, assignment is performed to a member from another member, so the original value is of the left-hand member is freed and the new value is copied. Similarly, lines 8

and 9 involve assignment to or from a **String_var**, so in both cases the original value of the left-hand side is freed and the new value is copied.

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

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

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

16.10 Mapping for Union Types

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

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

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

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

5 The following example helps illustrate the mapping for union types:

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

// C++
typedef Octet Bytes[64];
typedef Octet Bytes_slice;
class Bytes_forany { ... };
struct S { Long len; };
typedef ... A_ptr;
class U
{
  public:
     U();
     U(const U&);
     ~U();
     U &operator=(const U&);

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

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

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

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

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

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

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

7 For an array union member, the accessor returns a pointer to the array slice, where the slice is an array with all dimensions of the original except the first (array slices are described in detail in Section 16.12 on page 33). The array slice return type allows for read-write access for array members via regular subscript operators. For members of an anonymous array type, supporting typedefs for the array must be generated directly into the union. For example:

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

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

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

9 For string union members, the **`char*`** modifier results in the freeing of old storage before ownership of the pointer parameter is assumed, while the **`const char*`** modifier and the **`String_var`** modifier[9] both result in the freeing of old storage before the parameter's storage is copied. The accessor for a string member returns a **`const char*`** to allow examination but not modification of the string storage.[10]

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

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

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

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

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

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

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

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

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

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

16.11 Mapping for Sequence Types

1 A sequence is mapped to a C++ class that behaves like an array with a current length and a maximum length. For a bounded sequence, the maximum length is implicit in the sequence's type and cannot be explicitly controlled by the programmer. For an unbounded sequence, the initial value of the maximum length can be specified in the sequence constructor to allow control over the size of the initial buffer allocation. The programmer may always explicitly modify the current length of any sequence.

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

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

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

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

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

class LongSeqSeq                // bounded sequence
{
  public:
     LongSeqSeq();              // default constructor
     LongSeqSeq(                // T *data constructor
```

```
            ULong length,
            LongSeq *value,
            Boolean release = FALSE
        );
        LongSeqSeq(const LongSeqSeq&);
        ~LongSeqSeq();
        ...
};
```

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

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

7 The "**T *data**" constructor (as shown in the example above) allows the length and contents of a bounded or unbounded sequence to be set. For unbounded sequences, it also allows the initial value of the maximum length to be set. For this constructor, ownership of the contents vector is determined by the **release** parameter—**FALSE** means the caller owns the storage, while **TRUE** means that the sequence assumes ownership of the storage. If **release** is **TRUE**, the contents vector must have been allocated using the sequence **allocbuf** function, and the sequence will pass it to **freebuf** when finished with it. The **allocbuf** and **freebuf** functions are described on "Additional Memory Management Functions" on page 16-32.

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

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

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

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

12 For an unbounded sequence, the **maximum()** accessor function returns the total amount of buffer space currently available. This allows applications to know how many items they can insert into an unbounded sequence without causing a reallocation to occur. For a

bounded sequence, **maximum()** always returns the bound of the sequence as given in its OMG IDL type declaration.

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

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

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

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

16.11.1 Sequence Example

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

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

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

    ULong maximum() const;

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

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

```
};

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

     ULong maximum() const;

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

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

16.11.2 Using the "release" Constructor Parameter

18 Consider the following example:

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

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

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

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

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

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

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

22 As with other **out** and return values, **out** and return sequences must not be assigned to by the caller without first copying them. This is more fully explained in Section 16.18, "Argument Passing Considerations," on page 16-51.

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

16.11.3 Additional Memory Management Functions

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

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

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

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

16.11.4 Sequence T_var and T_out Types

27 In addition to the regular operations defined for **T_var** and **T_out** types, the **T_var** and **T_out** for a sequence type also supports an overloaded **operator[]** that forwards requests to the **operator[]** of the underlying sequence.[11] This subscript operator should have the same return type as that of the corresponding operator on the underlying sequence type.

16.12 Mapping For Array Types

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

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

```
// C++
F f1; F_var f2;
V v1; V_var v2;
M m1; M_var m2;

f(f2, v2, m2);
f1[0] = f2[1];
v1[1] = v2[1];                  // free old storage, copy
m1[0][1][2] = m2[0][1][2];      // free old storage, copy
```

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

3 As shown in Table 2 on page 16-54, **out** and return arrays are handled via pointer to array *slice*, where a slice is an array with all the dimensions of the original specified except the first one. As a convenience for application declaration of slice types, the mapping also provides a typedef for each array slice type. The name of the slice typedef consists of the name of the array type followed by the suffix "_slice". For example:

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

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

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

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

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

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

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

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

4 Because arrays are mapped into regular C++ arrays, they present special problems for the type-safe **any** mapping described in Section 16.14 on page 36. To facilitate their use with the **any** mapping, a compliant implementation must also provide for each array type a distinct C++ type whose name consists of the array name followed by the suffix _forany. These types must be distinct so as to allow functions to be overloaded on them. Like **Array_var** types, **Array_forany** types allow access to the underlying array type, but unlike **Array_var**, the **Array_forany** type does not **delete** the storage of the underlying array upon its own destruction. This is because the **Any** mapping retains storage ownership, as described in Section 16.14.3 on page 40.

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

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

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

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

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

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

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

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

16.13 Mapping For Typedefs

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

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

```
// C++
typedef Long T;

// ...definitions for A1...

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

// ...definitions for S1...

typedef S1 S2;
typedef S1_var S2_var;
```

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

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

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

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

inline another_array_slice*
another_array_dup(another_array_slice *a) {
     return array_dup(a);
}

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

16.14 Mapping for the Any Type

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

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

2 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.

16.14.1 Handling Typed Values

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

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

- As noted in Section 16.5, "Mapping for Basic Data Types," on page 16-11, the **boolean**, **octet**, and **char** OMG IDL types are not required to map to distinct C++ types, which means that a separate means of distinguishing them from each other for the purpose of function overloading is necessary. The means of distinguishing these types from each other is described in Section 16.14.4, "Distinguishing boolean, octet, char, and bounded string," on page 16-42.
- Since all strings are mapped to **`char*`** regardless of whether they are bounded or unbounded, another means of creating or setting an **any** with a bounded string value is necessary. This is described in Section 16.14.4, "Distinguishing boolean, octet, char, and bounded string," on page 16-42.
- In C++, arrays within a function argument list decay into pointers to their first elements. This means that function overloading cannot be used to distinguish between arrays of different sizes. The means for creating or setting an **any** when dealing with arrays is described below and in Section 16.12, "Mapping For Array Types," on page 16-33.

16.14.2 Insertion into Any

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

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

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

- **`Short`**, **`UShort`**, **`Long`**, **`ULong`**, **`Float`**, **`Double`**
- enumerations
- unbounded strings (**`char*`** passed by value)
- object references (**`T_ptr`**)

7 For values of type T that are too large to be passed by value efficiently, two forms of the insertion function are provided:

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

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

9 These "left-shift-assign" operators are used to insert a typed value into an **any** as follows:

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

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

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

- For the copying version of **operator<<=**, the lifetime of the value in the **Any** is independent of the lifetime of the value passed to **operator<<=**. The implementation of the **Any** may not store its value as a reference or pointer to the value passed to **operator<<=**.
- For the non-copying version of **operator<<=**, the inserted **T*** is consumed by the **Any**. The caller may not use the **T*** to access the pointed-to data after insertion since the **Any** assumes ownership of it, and it may immediately copy the pointed-to data and destroy the original.
- With both the copying and non-copying versions of **operator<<=**, any previous value held by the **Any** is properly deallocated. For example, if the **Any(TypeCode_ptr,void*,TRUE)** constructor (described in Section 16.14.6, "Handling Untyped Values," on page 16-45) was called to create the **Any**, the **Any** is responsible for deallocating the memory pointed to by the **void*** before copying the new value.

12 Copying insertion of a string type causes the following function to be invoked:

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

13 Since all string types are mapped to **char***, this insertion function assumes that the value being inserted is an unbounded string. Section 16.14.4, "Distinguishing boolean, octet, char, and bounded string," on page 16-42 describes how bounded strings may be correctly inserted into an **Any**. Non-copying insertion of both bounded and unbounded strings can be achieved using of the **Any::from_string** helper type described in Section 16.14.4, "Distinguishing boolean, octet, char, and bounded string," on page 16-42.

14 Type-safe insertion of arrays uses the **Array_forany** types described in Section 16.12, "Mapping For Array Types," on page 16-33. Compliant implementations must provide a version of **operator<<=** overloaded for each **Array_forany** type. For example:

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

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

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

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

Because of the type ambiguity between an array of **T** and a **T***, it is highly recommended that portable code explicitly[12] use the appropriate **Array_forany** type when inserting an array into an **Any**:

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

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

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

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

18 The non-copying version of **operator<<=** for object references takes the address of the **T_ptr** type:

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

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

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

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

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

16.14.3 Extraction From Any

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

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

21 This function signature suffices for primitive types that are normally passed by value. For values of type T that are too large to be passed by value efficiently, this function may be prototyped as follows:

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

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

- **Boolean**, **Char**, **Octet**, **Short**, **UShort**, **Long**, **ULong**, **Float**, **Double**
- enumerations
- unbounded strings (**char*** passed by reference, i.e., **char*&**)
- object references (**T_ptr**)

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

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

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

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

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

value of type **Long**, the value of the caller's reference variable is not changed, and **operator>>=** returns **FALSE**.

27 For non-primitive types, extraction is done by pointer. For example, consider the following IDL struct:

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

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

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

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

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

31 Correct extraction of array types relies on the **Array_forany** types described in Section 16.12, "Mapping For Array Types," on page 16-33.

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

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

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

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

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

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

16.14.4 Distinguishing boolean, octet, char, and bounded string

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

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

     void operator<<=(from_boolean);
     void operator<<=(from_char);
     void operator<<=(from_octet);
     void operator<<=(from_string);
```

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

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

    // other public Any details omitted

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

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

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

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

38 A bound value of 0 (zero) indicates an unbounded string.

39 For non-copying insertion of a bounded or unbounded string into an **Any**, the nocopy flag on the **from_string** constructor should be set to **TRUE**:

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

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

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

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

16.14.5 Widening to Object

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

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

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

16.14.6 Handling Untyped Values

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

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

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

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

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

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

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

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

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

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

16.14.7 Any Constructors, Destructor, Assignment Operator

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

53 Other constructors are described in Section 16.14.6, "Handling Untyped Values," on page 16-45.

54 ORB implementations concerned with single-process interoperability with the C mapping may overload **operator new()** and **operator delete()** for **Any**s so that

dynamic allocation uses the same mechanism as the C language dynamic allocation functions. Whether these operators are overloaded by the implementation or not, compliant programs use **new** to dynamically allocate anys and **delete** to free them.

16.14.8 The Any Class

55 The full definition of the **Any** class can be found in “Any Class” on page C-3.

16.14.9 The Any_var Class

56 Since **Any**s are returned via pointer as **out** and return parameters (see Table 2 on page 16-54), there exists an **Any_var** class similar to the **T_var** classes for object references. **Any_var** obeys the rules for **T_var** classes described in Section 16.8 on page 16, calling **delete** on its **Any*** when it goes out of scope or is otherwise destroyed. The full interface of the **Any_var** class is shown in Section C.4, “Any_var Class,” on page C-5.

16.15 Mapping for Exception Types

1 An OMG IDL exception is mapped to a C++ class that derives from the standard **UserException** class defined in the **CORBA** module (see Section 16.1.3, “CORBA Module,” on page 16-2). The generated class is like a variable-length struct, regardless of whether or not the exception holds any variable-length members. Just as for variable-length structs, each exception member must be self-managing with respect to its storage.

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

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

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

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

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

For environments that do not support exception handling, please refer to Section D.3, "Without Exception Handling," on page D-1 for information about the **_raise()** function.

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

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

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

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

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

     void _raise();

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

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

7 Each specific system exception (described in Section 14.1.6, "Exceptions," on page 14-3) is derived from **SystemException**:

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

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

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

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

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

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

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

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

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

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

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

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

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

16 For application portability, conforming C++ mapping implementations built using C++ compilers that support the standard C++ Run Time Type Information (RTTI) mechanisms still need to support narrowing for the **Exception** hierarchy. RTTI supports, among other things, determination of the run-time type of a C++ object. In particular, the **dynamic_cast<T*>** operator[13] allows for narrowing from a base pointer to a more derived pointer if the object pointed to really is of the more derived type. This operator is not useful for narrowing object references, since it cannot determine the actual type of remote objects, but it can be used by the C++ mapping implementation to narrow within the exception hierarchy.

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

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

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

16.16 Mapping For Operations and Attributes

1 An operation maps to a C++ function with the same name as the operation. Each read-write attribute maps to a pair of overloaded C++ functions (both with the same name), one to set the attribute's value and one to get the attribute's value. The *set* function takes an **in**

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

parameter with the same type as the attribute, while the *get* function takes no parameters and returns the same type as the attribute. An attribute marked **readonly** maps to only one C++ function, to get the attribute's value. Parameters and return types for attribute functions obey the same parameter passing rules as for regular operations.

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

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

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

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

4 Unlike the C mapping, C++ operations do not require an additional **Environment** parameter for passing exception information—real C++ exceptions are used for this purpose. See Section 16.15, "Mapping for Exception Types," on page 16-47 and "Without Exception Handling" on page D-1 for more details.

16.17 Implicit Arguments to Operations

1 If an operation in an OMG IDL specification has a context specification, then a **`Context_ptr`** input parameter (see Section 17.8.1, "Context Interface," on page 17-12) follows all operation-specific arguments. In an implementation that does not support real C++ exceptions, an output **Environment** parameter is the last argument, following all operation-specific arguments, and following the context argument if present. The parameter passing mode for **Environment** is described in "Without Exception Handling" on page D-1.

16.18 Argument Passing Considerations

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

2 Aggregate types are complicated by the question of when and how parameter memory is allocated and deallocated. Mapping **in** parameters is straightforward because the parameter storage is caller-allocated and read-only. The mapping for **out** and **inout**

parameters is more problematic. For variable-length types, the callee must allocate some if not all of the storage. For fixed-length types, such as a *Point* type represented as a struct containing three floating point members, caller allocation is preferable (to allow stack allocation).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9 To allow the callee the freedom to allocate a single contiguous area of storage for all the data associated with a parameter, we adopt the policy that the callee-allocated storage is not modifiable by the caller. However, trying to enforce this policy by returning a **const** type in C++ is problematic, since the caller is required to release the storage, and calling **delete** on a **const** object is an error[14]. A compliant mapping therefore is not required to detect this error.

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

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

14.It is very likely that the upcoming ANSI/ISO C++ standard will allow **delete** on a **const** object, but many C++ compilers do not yet support this feature.

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

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

11 Since OMG IDL has no concept of pointers in general or null pointers in particular, allowing the passage of null pointers to or from an operation would project C++ semantics onto OMG IDL operations.[15] A compliant implementation is allowed but not required to raise a **BAD_PARAM** exception if it detects such an error.

16.18.1 Operation Parameters and Signatures

12 Table 2 on page 16-54 displays the mapping for the basic OMG IDL parameter passing modes and return type according to the type being passed or returned, while Table 3 on page 16-55 displays the same information for **`T_var`** types. Table 3 is merely for informational purposes; it is expected that operation signatures for both clients and servers will be written in terms of the parameter passing modes shown in Table 2, with the exception that the **`T_out`** types will be used as the actual parameter types for all **out** parameters. It is also expected that **`T_var`** types will support the necessary conversion operators to allow them to be passed directly. Callers should always pass instances of either **`T_var`** types or the base types shown in Table 2, and callees should treat their **`T_out`** parameters as if they were actually the corresponding underlying types shown in Table 2.

13 In Table 2, fixed-length arrays are the only case where the type of an **out** parameter differs from a return value, which is necessary because C++ does not allow a function to return an array. The mapping returns a pointer to a *slice* of the array, where a slice is an array with all the dimensions of the original specified except the first one.

TABLE 2. Basic Argument and Result Passing

Data Type	In	Inout	Out	Return
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
unsigned short	UShort	UShort&	UShort&	UShort
unsigned long	ULong	ULong&	ULong&	ULong
float	Float	Float&	Float&	Float

15. When real C++ exceptions are not available, however, it is important that null pointers are returned whenever an **`Environment`** containing an exception is returned; see "Without Exception Handling" on page D-1 for more details.

TABLE 2. Basic Argument and Result Passing *(Continued)*

Data Type	In	Inout	Out	Return
double	Double	Double&	Double&	Double
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum
object reference ptr[1]	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct, fixed	const struct&	struct&	struct&	struct
struct, variable	const struct&	struct&	struct*&	struct*
union, fixed	const union&	union&	union&	union
union, variable	const union&	union&	union*&	union*
string	const char*	char*&	char*&	char*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, fixed	const array	array	array	array slice*[2]
array, variable	const array	array	array slice*&[2]	array slice*[2]
any	const any&	any&	any*&	any*

1. Including pseudo-object references.

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

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

TABLE 3. T_var Argument and Result Passing

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

1. Including pseudo-object references.

15 Table 26 on page 16-45 and Table 5 on page 16-56 describe the caller's responsibility for storage associated with **inout** and **out** parameters and for return results.

TABLE 4. Caller Argument Storage Responsibilities

Type	Inout Param	Out Param	Return Result
short	1	1	1
long	1	1	1
unsigned short	1	1	1

TABLE 4. Caller Argument Storage Responsibilities *(Continued)*

Type	Inout Param	Out Param	Return Result
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enum	1	1	1
object reference ptr	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3

TABLE 5. Argument Passing Cases

Case[1]	
1	Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller provides the initial value, and the callee may change that value. For out parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.
2	Caller allocates storage for the object reference. For inout parameters, the caller provides an initial value; if the callee wants to reassign the inout parameter, it will first call CORBA::release on the original input value. To continue to use an object reference passed in as an inout, the caller must first duplicate the reference. The caller is responsible for the release of all out and return object references. Release of all object references embedded in other structures is performed automatically by the structures themselves.
3	For out parameters, the caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned instance into a new instance, then modify the new instance.

TABLE 5. Argument Passing Cases *(Continued)*

Case[1]	
4	For inout strings, the caller provides storage for both the input string and the char* pointing to it. Since the callee may deallocate the input string and reassign the char* to point to new storage to hold the output value, the caller should allocate the input string using string_alloc(). The size of the out string is therefore not limited by the size of the in string. The caller is responsible for deleting the storage for the out using string_free(). The callee is not allowed to return a null pointer for an inout, out, or return value.
5	For inout sequences and anys, assignment or modification of the sequence or any may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the Boolean release parameter with which the sequence or any was constructed.
6	For out parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance.

1. As listed in Table 26 on page 16-45

Mapping of Pseudo Objects to C++ 17

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

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

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

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

17.1 Usage

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

- They are prefaced by the keyword **pseudo.**
- Their declarations may refer to other[1] serverless object types that are not otherwise necessarily allowed in OMG IDL.

2 As explained in Section 14.23, “Pseudo-objects,” on page 14-26, the **pseudo** prefix means that the interface may be implemented in either a normal or serverless fashion. That is, apply either the rules described in the following sections or the normal mapping rules described in Chapter 16, “Mapping of OMG IDL to C++".

17.2 Mapping Rules

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

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

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

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

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

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

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

1. The types of references to serverless objects, **T_ptr**, may or may not simply be a typedef of **T***.
2. Each mapped class supports the following static member functions:

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

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

---

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

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

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

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

17.3 Relation to the C PIDL Mapping

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

- C++-PIDL calls for removal of representation dependencies through the use of interfaces rather than structs and typedefs.
- C++-PIDL calls for placement of operations on pseudo objects in their interfaces, including a few cases of redesignated functionality as noted.
- In C++-PIDL, the **`release`** performs the role of the associated **`free`** and **`delete`** operations in the C mapping, unless otherwise noted.

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

17.4 Environment

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

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

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

17.4.1 Environment Interface

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

17.4.2 Environment C++ Class

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

17.4.3 Differences from C-PIDL

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

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

17.4.4 Memory Management

5 **Environment** has the following special memory management rules:

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

17.5 NamedValue

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

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

17.5.1 NamedValue Interface

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

17.5.2 NamedValue C++ Class

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

17.5.3 Differences from C-PIDL

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

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

17.5.4 Memory Management

4 **NamedValue** has the following special memory management rules:

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

17.6 NVList

1 **NVList** is a list of **NamedValue**s. A new **NVList** is constructed using the **ORB::create_list** operation (see Section 17.10, "ORB," on page 17-16). New **NamedValue**s may be constructed as part of an **NVList**, in any of three ways:

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

2 Each of these operations returns the new item.

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

17.6.1 NVList Interface

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

17.6.2 NVList C++ Class

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

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

17.6.3 Differences from C-PIDL

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

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

17.6.4 Memory Management

5 **`NVList`** has the following special memory management rules:

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

17.7 Request

1 **Request** provides the primary support for DII. A new request on a particular target object may be constructed using the short version of the request creation operation shown in Section 17.11, “Object,” on page 17-19:

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

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

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

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

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

4 Alternatively, requests can be constructed using one of the long forms of the creation operation shown in the Object interface in Section 17.11, “Object,” on page 17-19:

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

```
                                const char *operation,
                                NVList_ptr arg_list,
                                NamedValue_ptr result,
                                ExceptionList_ptr,
                                ContextList_ptr,
                                Request_out request,
                                Flags req_flags
);
```

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

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

- a target object reference
- an operation name
- a list of arguments (optional)
- a place to put the result (optional)
- a place to put any returned exceptions
- a **Context** (optional)
- a list of the user-defined exceptions that can be thrown (optional)
- a list of **Context** strings that must be sent with the operation (optional)

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

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

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

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

17.7.1 Request Interface

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

pseudo interface ContextList
{
    readonly attribute unsigned long count;
    void add(in string ctxt);
    string item(in unsigned long index) raises(Bounds);
    Status remove(in unsigned long index) raises(Bounds);
};

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

    attribute context ctx;

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

17.7.2 Request C++ Class

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

```
};

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

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

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

     // argument manipulation helper functions
     Any &add_in_arg();
     Any &add_in_arg(const char* name);
     Any &add_inout_arg();
     Any &add_inout_arg(const char* name);
     Any &add_out_arg();
     Any &add_out_arg(const char* name);
     void set_return_type(TypeCode_ptr tc);
     Any &return_value();

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

17.7.3 Differences from C-PIDL

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

- Replacement of **add_argument**, and so forth, with attribute-based accessors.
- Use of **env** attribute to access exceptions raised in DII calls.

- The **invoke** operation does not take a flag argument, since there are no flag values that are listed as legal in *CORBA V2.0*.
- The **send_oneway** and **send_deferred** operations replace the single **send** operation with flag values, in order to clarify usage.
- The **get_response** operation does not take a flag argument, and an operation **poll_response** is defined to immediately return with an indication of whether the operation has completed. This was done because in *CORBA V2.0,* if the type **Status** is **void**, the version with **RESP_NO_WAIT** does not enable the caller to determine if the operation has completed.
- The **add_*_arg**, **set_return_type**, and **return_value** member functions are added as shortcuts for using the attribute-based accessors.

17.7.4 Memory Management

10 **Request** has the following special memory management rules:

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

11 **ExceptionList** has the following special memory management rules:

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

12 **ContextList** has the following special memory management rules:

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

17.8 Context

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

17.8.1 Context Interface

```
// IDL
pseudo interface Context
{
```

```
    readonly attribute Identifier context_name;
    readonly attribute context parent;

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

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

17.8.2 Context C++ Class

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

     Status create_child(const char *, Context_out);

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

17.8.3 Differences from C-PIDL

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

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

17.8.4 Memory Management

`Context` has the following special memory management rules:

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

17.9 TypeCode

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

2 No constructors for **TypeCode**s are defined. However, in addition to the mapped interface, for each basic and defined OMG IDL type, an implementation provides access to a **TypeCode** pseudo object reference (**`TypeCode_ptr`**) of the form **`_tc_<type>`** that may be used to set types in **`Any`**, as arguments for **equal**, and so on. In the names of these **TypeCode** reference constants, **`<type>`** refer to the local name of the type within its defining scope. Each C++ **`_tc_<type>`** constant must be defined at the same scoping level as its matching type.

3 In all C++ **TypeCode** pseudo object reference constants, the prefix "_tc_" should be used instead of the "TC_" prefix prescribed in Section 6.7, "TypeCodes," on page 6-33. This is to avoid name clashes for CORBA applications that simultaneously use both the C and C++ mappings.

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

17.9.1 TypeCode Interface

```
// IDL
pseudo interface TypeCode
{
    exception Bounds {};
    exception BadKind {};

    // for all TypeCode kinds
    boolean equal(in TypeCode tc);
    TCKind kind();

    // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias, and tk_except
    RepositoryId id() raises(BadKind);
    Identifier name() raises(BadKind);

    // for tk_struct, tk_union, tk_enum, and tk_except
    unsigned long member_count() raises(BadKind);
```

```
    Identifier member_name(in unsigned long index)
                                        raises(BadKind, Bounds);

    // for tk_struct, tk_union, and tk_except
    TypeCode member_type(in unsigned long index)
                                        raises(BadKind, Bounds);

    // for tk_union
    any member_label(in unsigned long index)
                                        raises(BadKind, Bounds);
    TypeCode discriminator_type() raises(BadKind);
    long default_index() raises(BadKind);

    // for tk_string, tk_sequence, and tk_array
    unsigned long length() raises(BadKind);

    // for tk_sequence, tk_array, and tk_alias
    TypeCode content_type() raises(BadKind);

    // deprecated interface
    long param_count();
    any parameter(in long index) raises(bounds);
};
```

17.9.2 TypeCode C++ Class

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

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

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

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

     TypeCode_ptr member_type(ULong index) const;

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

```
        ULong length() const;

        TypeCode_ptr content_type() const;

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

17.9.3 Differences from C-PIDL

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

17.9.4 Memory Management

`TypeCode` has the following special memory management rules:

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

17.10 ORB

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

17.10.1 ORB Interface

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

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

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

```
    Status get_next_response(out Request req);
};
```

17.10.2 ORB C++ Class

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

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

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

17.10.3 Differences from C-PIDL

- Added **create_environment**. Unlike the struct version, **Environment** requires a construction operation. (Since this is overly constraining for implementations that do not support real C++ exceptions, these implementations may allow **`Environment`** to be declared on the stack. See Section D.3, "Without Exception Handling," on page D-1 for details.)
- Assigned multiple request support to ORB, made usage symmetrical with that in **Request**, and used a sequence type rather than otherwise illegal unbounded arrays in signatures.
- Added **create_named_value**, which is required for creating **NamedValue** objects to be used as return value parameters for the **Object::create_request** operation.
- Added **create_exception_list** and **create_context_list** (see Section 17.7, "Request," on page 17-7 for more details).

17.10.4 Mapping of ORB Initialization Operations

2 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 Section 7.4, "ORB Initialization," on page 7-6.

```
// PIDL
module CORBA {
    typedef string ORBid;
    typedef sequence <string> arg_list;
    ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

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

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

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

5 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.

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

7 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.

8 For C++, the order of consumption of **`argv`** parameters may be significant to an application. In order to ensure that applications are not required to handle **`argv`** parameters they do not recognize the ORB initialization function must be called before the remainder of the parameters are consumed. Therefore, after the **`ORB_init`** call the **`argv`** and **`argc`** parameters will have been modified to remove the ORB understood arguments. It is important to note that the ORB_init call can only reorder or remove references to parameters from the argv list, this restriction is made in order to

avoid potential memory management problems caused by trying to free parts of the argv list or extending the argv list of parameters. This is why **argv** is passed as a **char**** and not a **char**&**.

17.10.5 Mapping of Operations to Obtain Initial Object References

9 The following PIDL specifies the operations (in the ORB interface) that allow applications to get pseudo object references for the Interface Repository and Object Services. It is described in detail in Section 7.6, "Obtaining Initial Object References," on page 7-10.

```
// PIDL
module CORBA {
    interface ORB {
        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;

        exception InvalidName {};

        ObjectIdList list_initial_services ();

        Object resolve_initial_references (in ObjectId identifier)
                raises (InvalidName);
    };
};
```

10 The mapping of the preceding PIDL to the C++ language is as follows:

```
// C++
namespace CORBA {
     class ORB {
       public:
          typedef char* ObjectId;
          class ObjectIdList {...};
          class InvalidName {...};
          ObjectIdList *list_initial_services();
          Object_ptr resolve_initial_references(
                    const char *identifier
          );
     };
}
```

17.11 Object

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

17.11.1 Object Interface

```
// IDL
interface Object
{
    boolean is_nil();
    Object duplicate();
    void release();
    ImplementationDef get_implementation();
    InterfaceDef get_interface();
    Status create_request(
                in Context ctx,
                in Identifier operation,
                in NVList arg_list,
                in NamedValue result,
                out Request request,
                in Flags req_flags
    );
    Status create_request2(
                in Context ctx,
                in Identifier operation,
                in NVList arg_list,
                in NamedValue result,
                in ExceptionList exclist,
                in ContextList ctxtlist,
                out Request request,
                in Flags req_flags
    );
};
```

17.11.2 Object C++ Class

2 In addition to other rules, all operation names in interface **Object** have leading underscores in the mapped C++ class. Also, the mapping for **create_request** is split into three forms, corresponding to the usage styles described in Section 4.2.1, “create_request,” on page 4-4 and in Section 17.7, “Request,” on page 17-7 of this document. The **is_nil** and **release** functions are provided in the **CORBA** namespace, as described in Section 16.3.3, “Object Reference Operations,” on page 16-5.

```
// C++
class Object
{
  public:
     static Object_ptr _duplicate(Object_ptr obj);
     static Object_ptr _nil();
     ImplementationDef_ptr _get_implementation();
     InterfaceDef_ptr _get_interface();
     Status _create_request(
          Context_ptr ctx,
          const char *operation,
```

```
                    NVList_ptr arg_list,
                    NamedValue_ptr result,
                    Request_out request,
                    Flags req_flags
                );
                Status _create_request(
                    Context_ptr ctx,
                    const char *operation,
                    NVList_ptr arg_list,
                    NamedValue_ptr result,
                    ExceptionList_ptr,
                    ContextList_ptr,
                    Request_out request,
                    Flags req_flags
                );
                Request_ptr _request(const char* operation);
            };
```

Server-Side Mapping 18

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

18.1 Implementing Interfaces

1 To define an implementation in C++, one defines a C++ class with any valid C++ name. For each operation in the interface, the class defines a non-static member function with the mapped name of the operation (the mapped name is the same as the OMG IDL identifier except when the identifier is a C++ keyword, in which case the string "_cxx_" is prepended to the identifier, as noted in Section 16.1, "Preliminary Information," on page 16-1). Note that the ORB implementation may allow one implementation class to derive from another, so the statement "the class defines a member function" does not mean the class must explicitly define the member function—it could inherit the function.

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

18.1.1 Mapping of PortableServer::Servant

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

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

          ServantBase& operator=(const ServantBase&);

          virtual POA_ptr _default_POA();

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

4 The **ServantBase** destructor is public and virtual to ensure that skeleton classes derived from it can be properly destroyed. The default constructor, along with other implementation-specific constructors, must be protected so that instances of **ServantBase** cannot be created except as sub-objects of instances of derived classes. A default constructor (a constructor that either takes no arguments or takes only arguments with default values) must be provided so that derived servants can be constructed portably. Both copy construction and a public default assignment operator must be supported so that application-specific servants can be copied if necessary. Note that copying a servant that is already registered with the object adapter, either by assignment or by construction, does not mean that the target of the assignment or copy is also registered with the object adapter. Similarly, assigning to a **ServantBase** or a class derived from it that is already registered with the object adapter does not in any way change its registration.

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

18.1.2 Skeleton Operations

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

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

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

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

7 For example, using interface **A** as defined above in Section 18.1.3 on page 4:

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

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

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

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

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

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

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

18.1.3 Inheritance-Based Interface Implementation

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

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

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

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

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

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

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

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

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

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

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

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

Note that the presence of the **_this()** function implies that C++ servants must only be derived directly from a single skeleton class. Direct derivation from multiple skeleton classes could result in ambiguity errors due to multiple definitions of **_this()**. This should not be a limitation, since CORBA objects have only a single most-derived interface. Servants that are intended to support multiple interface types can utilize the delegation-based interface implementation approach, described below in Section 18.1.4 on page 5, or can be registered as DSI-based servants, as described in Section 18.3 on page 11.

18.1.4 Delegation-Based Interface Implementation

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

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

21 For the examples in this section, the OMG IDL interface from Section 18.1.3 on page 4 will again be used:

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

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

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

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

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

```
            CORBA::release(_poa);
            if (_rel) delete _ptr;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

PortableServer::Current::get_object_id(), passing its return value to **PortableServer::POA::id_to_reference()**, and then narrowing the returned object reference appropriately.

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

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

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

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

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

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

18.2 Implementing Operations

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

```
// IDL
interface A
{
    exception B {};
    void f() raises(B);
};

// C++
class MyA : public virtual POA_A
{
  public:
     void f() throw(A::B, CORBA::SystemException);
     ...
};
```

2 Since all operations and attributes may throw CORBA system exceptions, **`CORBA::SystemException`** must appear in all exception specifications, even when an operation has no **raises** clause.

3 Within a member function, the “this” pointer refers to the implementation object’s data as defined by the class. In addition to accessing the data, a member function may implicitly call another member function defined by the same class. For example:

```
// IDL
interface A
{
    void f();
    void g();
};

// C++
class MyA : public virtual POA_A
{
  public:
     void f() throw(SystemException);
     void g() throw(SystemException);
  private:
     long x_;
};

void
MyA::f() throw(SystemException)
{
     this->x_ = 3;
     this->g();
}
```

4 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.

18.2.1 Skeleton Derivation From Object

5 In several existing ORB implementations, each skeleton class derives from the corresponding interface class. For example, for interface **Mod::A**, the skeleton class **POA_Mod::A** is derived from class **Mod::A**. These systems therefore allow an object reference for a servant to be implicitly obtained via normal C++ derived-to-base conversion rules:

```
// C++
MyImplOfA my_a;          // declare impl of A
A_ptr a = &my_a;         // obtain its object reference
                         // by C++ derived-to-base
                         // conversion
```

6 Such code can be supported by a conforming ORB implementation, but it is not required, and is thus not portable. The equivalent portable code invokes **_this()** on the implementation object in order to implicitly register it if it has not yet been registered, and to get its object reference:

```
// C++
MyImplOfA my_a;               // declare impl of A
A_ptr a = my_a._this();       // obtain its object
                              // reference
```

18.3 Mapping of Dynamic Skeleton Interface to C++

1 Section 5.3, “Dynamic Skeleton Interface: Language Mapping,” on page 5-3 contains general information about mapping the Dynamic Skeleton Interface to programming languages.

2 This section contains the following information:

- Mapping of the Dynamic Skeleton Interface’s **ServerRequest** to C++
- Mapping of the Portable Object Adapter’s Dynamic Implementation Routine to C++

18.3.1 Mapping of ServerRequest to C++

3 The **ServerRequest** pseudo object maps to a C++ class in the **CORBA** namespace which supports the following operations and signatures:

```
// C++
class ServerRequest
{
  public:
     const char* operation() const;
     void arguments(NVList_ptr& parameters);
     Context_ptr ctx();
     void set_result(const Any& value);
     void set_exception(const Any& value);
};
```

4 Note that, as with the rest of the C++ mapping, ORB implementations are free to make such operations virtual and modify the inheritance as needed.

5 All of these operations follow the normal memory management rules for data passed into skeletons by the ORB. That is, the DIR is not allowed to modify or change the string returned by **`operation()`**, **in** parameters in the **`NVList`** returned from **`arguments()`**, or the **`Context`** returned by **`ctx()`**. Similarly, data allocated by the DIR and handed to the ORB (the **`NVList`** parameters, the result value, and exception values) are freed by the ORB rather than by the DIR.

18.3.2 Handling Operation Parameters and Results

6 The **`ServerRequest`** provides parameter values when the DIR invokes the **`arguments()`** operation. The **`NVList`** provided by the DIR to the ORB includes the **`TypeCodes`** and direction **`Flags`** (inside **`NamedValues`**) for all parameters, including **out** ones for the operation. This allows the ORB to verify that the correct parameter types have been provided before filling their values in, but does not require it to do so. It also relieves the ORB of all responsibility to consult an Interface Repository, promoting high performance implementations.

7 The **`NVList`** provided to the ORB then becomes owned by the ORB. It will not be deallocated until after the DIR returns. This allows the DIR to pass the **out** values, including the return side of **inout** values, to the ORB by modifying the **`NVList`** after **`arguments()`** has been called. Therefore, if the DIR stores the **`NVList_ptr`** into an **`NVList_var`**, it should pass it to the **`arguments()`** function by invoking the **`_retn()`** function on it, in order to force it to release ownership of its internal **`NVList_ptr`** to the ORB.

18.3.3 Mapping of PortableServer Dynamic Implementation Routine

8 In C++, DSI servants inherit from the standard **`DynamicImplementation`** class. This class inherits from the **`ServantBase`** class and is also defined in the **`PortableServer`** namespace. The Dynamic Skeleton Interface (DSI) is implemented through servants that are members of classes that inherit from dynamic skeleton classes.

```
// C++
namespace PortableServer
{
     class DynamicImplementation : public virtual ServantBase
     {
       public:
          CORBA::Object_ptr _this();
          virtual void invoke(
                         CORBA::ServerRequest_ptr request
                   ) = 0;
          virtual CORBA::RepositoryId
          _primary_interface(
                         const ObjectId& oid,
                         POA_ptr poa
          ) = 0;
     };
}
```

9 The **_this()** function returns a **CORBA::Object_ptr** for the target object. Unlike **_this()** for static skeletons, its return type is not interface-specific because a DSI servant may very well incarnate multiple CORBA objects of different types. If **DynamicImplementation::_this()** is invoked outside of the context of a request invocation on a target object being served by the DSI servant, it raises the **PortableServer::WrongPolicy** exception.

10 The **invoke()** method receives requests issued to any CORBA object incarnated by the DSI servant and performs the processing necessary to execute the request.

11 The **_primary_interface()** method receives an **ObjectId** value and a **POA_ptr** as input parameters and returns a valid **RepositoryId** representing the most-derived interface for that **oid**.

It is expected that the **invoke()** and **_primary_interface()** methods will be only invoked by the POA in the context of serving a CORBA request. Invoking this method in other circumstances may lead to unpredictable results.

18.4 PortableServer Functions

1 Objects registered with POAs use sequences of octet, specifically the **PortableServer::POA::ObjectId** type, as object identifiers. However, because C++ programmers will often want to use strings as object identifiers, the C++ mapping provides several conversion functions that convert strings to **ObjectId** and vice-versa:

```
// C++
namespace PortableServer
{
     char* ObjectId_to_string(const ObjectId&);
     wchar_t* ObjectId_to_wstring(const ObjectId&);

     ObjectId* string_to_ObjectId(const char*);
     ObjectId* wstring_to_ObjectId(const wchar_t*);
}
```

2 These functions follow the normal C++ mapping rules for parameter passing and memory management.

3 If conversion of an **`ObjectId`** to a string would result in illegal characters in the string (such as a NUL), the first two functions throw the **CORBA::BAD_PARAM** exception.

18.5 Mapping for PortableServer::ServantManager

18.5.1 Mapping for Cookie

1 Since **PortableServer::ServantLocator::Cookie** is an IDL **native** type, its type must be specified by each language mapping. In C++, **Cookie** maps to **`void*`**:

```
// C++
namespace PortableServer
{
     typedef void* Cookie;
}
```

2 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`**.

18.5.2 ServantManagers and AdapterActivators

3 Portable servants that implement the **`PortableServer::AdapterActivator`**, the **`PortableServer::ServantActivator`**, or **`PortableServer::ServantLocator`** interfaces are implemented just like any other servant. They may use either the inheritance-based approach or the tie approach.

C++ Definitions for CORBA C

This appendix provides a complete set of C++ definitions for the **CORBA** module. The definitions appear within the C++ namespace named **CORBA**.

```
// C++
namespace CORBA { ... }
```

Any implementations shown here are merely sample implementations: they are not the required definitions for these types.

C.1 Primitive Types

```
typedef unsigned char   Boolean;
typedef unsigned char   Char;
typedef unsigned char   Octet;
typedef short           Short;
typedef unsigned short  UShort;
typedef long            Long;
typedef unsigned long   ULong;
typedef float           Float;
typedef double          Double;

typedef Boolean&        Boolean_out;
typedef Char&           Char_out;
typedef Octet&          Octet_out;
typedef Short&          Short_out;
typedef UShort&         UShort_out;
typedef Long&           Long_out;
typedef ULong&          ULong_out;
typedef Float&          Float_out;
typedef Double&         Double_out;
```

C.2 String_var Class

```
class String_var
{
  public:
   String_var();
   String_var(char *p);
   String_var(const char *p);
   String_var(const String_var &s);
   ~String_var();

   String_var &operator=(char *p);
   String_var &operator=(const char *p);
   String_var &operator=(const String_var &s);

   operator char*();
   operator const char*() const;

   const char* in() const;
   char*& inout();
   char*& out();
   char* _retn();

   char &operator[](ULong index);
   char operator[](ULong index) const;
};

class String_out
{
  public:
   String_out(char*& p);
   String_out(String_var& p);
   String_out(String_out& s);
   String_out& operator=(String_out& s);
   String_out& operator=(char* p);
   String_out& operator=(const char* p)

   operator char*&();
   char*& ptr();

  private:
   // assignment from String_var disallowed
   void operator=(const String_var&);
};
```

C.3 Any Class

```
class Any
{
  public:
   Any();
   Any(const Any&);
   Any(TypeCode_ptr tc, void *value, Boolean release = FALSE);
   ~Any();

   Any &operator=(const Any&);

   void operator<<=(Short);
   void operator<<=(UShort);
   void operator<<=(Long);
   void operator<<=(ULong);
   void operator<<=(Float);
   void operator<<=(Double);
   void operator<<=(const Any&);
   void operator<<=(const char*);

   Boolean operator>>=(Short&) const;
   Boolean operator>>=(UShort&) const;
   Boolean operator>>=(Long&) const;
   Boolean operator>>=(ULong&) const;
   Boolean operator>>=(Float&) const;
   Boolean operator>>=(Double&) const;
   Boolean operator>>=(Any&) const;
   Boolean operator>>=(char*&) const;

   // special types needed for boolean, octet, char,
   // and bounded string insertion
   // these are suggested implementations only
   struct from_boolean {
      from_boolean(Boolean b) : val(b) {}
      Boolean val;
   };
   struct from_octet {
      from_octet(Octet o) : val(o) {}
      Octet val;
   };
   struct from_char {
      from_char(Char c) : val(c) {}
      Char val;
   };
   struct from_string {
      from_string(char* s, ULong b, Boolean nocopy = FALSE) :
         val(s), bound(b) {}
      char *val;
      ULong bound;
   };
```

```
    void operator<<=(from_boolean);
    void operator<<=(from_char);
    void operator<<=(from_octet);
    void operator<<=(from_string);

    // special types needed for boolean, octet, char extraction
    // these are suggested implementations only
    struct to_boolean {
       to_boolean(Boolean &b) : ref(b) {}
       Boolean &ref;
    };
    struct to_char {
       to_char(Char &c) : ref(c) {}
       Char &ref;
    };
    struct to_octet {
       to_octet(Octet &o) : ref(o) {}
       Octet &ref;
    };
    struct to_object {
       to_object(Object_ptr &obj) : ref(obj) {}
       Object_ptr &ref;
    };
    struct to_string {
       to_string(char *&s, ULong b) : val(s), bound(b) {}
       char *&val;
       ULong bound;
    };

    Boolean operator>>=(to_boolean) const;
    Boolean operator>>=(to_char) const;
    Boolean operator>>=(to_octet) const;
    Boolean operator>>=(to_object) const;
    Boolean operator>>=(to_string) const;

    void replace(TypeCode_ptr, void *value, Boolean release = FALSE);

    TypeCode_ptr type() const;
    const void *value() const;

  private:
    // these are hidden and should not be implemented
    // so as to catch erroneous attempts to insert or extract
    // multiple IDL types mapped to unsigned char
    void operator<<=(unsigned char);
    Boolean operator>>=(unsigned char&) const;
};
```

C.4 Any_var Class

```
class Any_var
{
  public:
   Any_var();
   Any_var(Any *a);
   Any_var(const Any_var &a);
   ~Any_var();

   Any_var &operator=(Any *a);
   Any_var &operator=(const Any_var &a);

   Any *operator->();

   const Any& in() const;
   Any& inout();
   Any*& out();
   Any* _retn();

   // other conversion operators for parameter passing
};
```

C.5 Exception Class

```
// C++
class Exception
{
  public:
   Exception(const Exception &);
   virtual ~Exception();
   Exception &operator=(const Exception &);

   virtual void _raise() = 0;

  protected:
   Exception();
};
```

C.6 SystemException Class

```
// C++
enum CompletionStatus { COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE };
class SystemException : public Exception
{
  public:
   SystemException();
   SystemException(const SystemException &);
   SystemException(ULong minor, CompletionStatus status);
```

```
    ~SystemException();
    SystemException &operator=(const SystemException &);

    ULong minor() const;
    void minor(ULong);

    CompletionStatus completed() const;
    void completed(CompletionStatus);

    static SystemException* _narrow(Exception*);
};
```

C.7 UserException Class

```
// C++
class UserException : public Exception
{
  public:
    UserException();
    UserException(const UserException &);
    ~UserException();
    UserException &operator=(const UserException &);

    static UserException* _narrow(Exception*);
};
```

C.8 UnknownUserException Class

```
// C++
class UnknownUserException : public UserException
{
  public:
    Any &exception();

    static UnknownUserException* _narrow(Exception*);
    virtual void raise();
};
```

C.9 release and is_nil

```
// C++
namespace CORBA {
    void release(Object_ptr);
    void release(Environment_ptr);
    void release(NamedValue_ptr);
    void release(NVList_ptr);
    void release(Request_ptr);
    void release(Context_ptr);
```

```
    void release(TypeCode_ptr);
    void release(POA_ptr);
    void release(ORB_ptr);

    Boolean is_nil(Object_ptr);
    Boolean is_nil(Environment_ptr);
    Boolean is_nil(NamedValue_ptr);
    Boolean is_nil(NVList_ptr);
    Boolean is_nil(Request_ptr);
    Boolean is_nil(Context_ptr);
    Boolean is_nil(TypeCode_ptr);
    Boolean is_nil(POA_ptr);
    Boolean is_nil(ORB_ptr);
    ...
}
```

C.10 Object Class

```
// C++
class Object
{
  public:
    static Object_ptr _duplicate(Object_ptr obj);
    static Object_ptr _nil();
    InterfaceDef_ptr _get_interface();
    Status _create_request(
              Context_ptr ctx,
              const char *operation,
              NVList_ptr arg_list,
              NamedValue_ptr result,
              Request_out request,
              Flags req_flags
            );
    Status _create_request(
       Context_ptr ctx,
       const char *operation,
       NVList_ptr arg_list,
       NamedValue_ptr result,
       ExceptionList_ptr,
       ContextList_ptr,
       Request_out request,
       Flags req_flags
    );
    Request_ptr _request(const char* operation);
};
```

C.11 Environment Class

```
// C++
class Environment
{
  public:
   void exception(Exception*);
   Exception *exception() const;
   void clear();

   static Environment_ptr _duplicate();
   static Environment_ptr _nil();
};
```

C.12 NamedValue Class

```
// C++
class NamedValue
{
  public:
   const char *name() const;
   Any *value() const;
   Flags flags() const;

   static NamedValue_ptr _duplicate();
   static NamedValue_ptr _nil();
};
```

C.13 NVList Class

```
// C++
class NVList
{
  public:
   ULong count() const;
   NamedValue_ptr add(Flags);
   NamedValue_ptr add_item(const char*, Flags);
   NamedValue_ptr add_value(const char*, const Any&, Flags);
   NamedValue_ptr add_item_consume(
         char*,
         Flags
   );
   NamedValue_ptr add_value_consume(
         char*,
         Any *,
```

```
        Flags
    );
    NamedValue_ptr item(ULong);
    Status remove(ULong);

    static NVList_ptr _duplicate();
    static NVList_ptr _nil();
};
```

C.14 ExceptionList Class

```
// C++
class ExceptionList
{
  public:
    ULong count();
    void add(TypeCode_ptr tc);
    void add_consume(TypeCode_ptr tc);
    TypeCode_ptr item(ULong index);
    Status remove(ULong index);
};
```

C.15 ContextList Class

```
class ContextList
{
  public:
    ULong count();
    void add(const char* ctxt);
    void add_consume(char* ctxt);
    const char* item(ULong index);
    Status remove(ULong index);
};
```

C.16 Request Class

```
// C++
class Request
{
  public:
    Object_ptr target() const;
    const char *operation() const;
    NVList_ptr arguments();
    NamedValue_ptr result();
    Environment_ptr env();
    ExceptionList_ptr exceptions();
    ContextList_ptr contexts();

    void ctx(Context_ptr);
```

```
    Context_ptr ctx() const;

    Status invoke();
    Status send_oneway();
    Status send_deferred();
    Status get_response();
    Boolean poll_response();

    static Request_ptr _duplicate();
    static Request_ptr _nil();
};
```

C.17 Context Class

```
// C++
class Context
{
  public:
    const char *context_name() const;
    Context_ptr parent() const;

    Status create_child(const char*, Context_out);

    Status set_one_value(const char*, const Any&);
    Status set_values(NVList_ptr);
    Status delete_values(const char*);
    Status get_values(const char*, Flags, const char*, NVList_out);

    static Context_ptr _duplicate();
    static Context_ptr _nil();
};
```

C.18 TypeCode Class

```
// C++
class TypeCode
{
  public:
    class Bounds { ... };
    class BadKind { ... };

    TCKind kind() const;
    Boolean equal(TypeCode_ptr) const;

    const char* id() const;
    const char* name() const;

    ULong member_count() const;
    const char* member_name(ULong index) const;
```

```
    TypeCode_ptr member_type(ULong index) const;

    Any *member_label(ULong index) const;
    TypeCode_ptr discriminator_type() const;
    Long default_index() const;

    ULong length() const;

    TypeCode_ptr content_type() const;

    Long param_count() const;
    Any *parameter(Long) const;

    static TypeCode_ptr _duplicate();
    static TypeCode_ptr _nil();
};
```

C.19 ORB Class

```
// C++
class ORB
{
  public:
    typedef sequence<Request_ptr> RequestSeq;
    char *object_to_string(Object_ptr);
    Object_ptr string_to_object(const char*);
    Status create_list(Long, NVList_out);
    Status create_operation_list(OperationDef_ptr, NVList_out);
    Status create_named_value(NamedValue_out);
    Status create_exception_list(ExceptionList_out);
    Status create_context_list(ContextList_out);

    Status get_default_context(Context_out);
    Status create_environment(Environment_out);

    Status send_multiple_requests_oneway(const RequestSeq&);
    Status send_multiple_requests_deferred(const RequestSeq&);
    Boolean poll_next_response();
    Status get_next_response(Request_out);
```

```
   // Obtaining initial object references
   typedef char* ObjectId;
   class ObjectIdList {...};
   class InvalidName {...};
   ObjectIdList *list_initial_services();
   Object_ptr resolve_initial_references(const char *identifier);

   Boolean work_pending();
   void perform_work();
   void shutdown(Boolean wait_for_completion);
   void run();

   static ORB_ptr _duplicate();
   static ORB_ptr _nil();
};
```

C.20 ORB Initialization

```
// C++
typedef char* ORBid;
static ORB_ptr ORB_init(
                        int& argc,
                        char** argv,
                        const char* orb_identifier = ""
                );
```

C.21 General T_out Types

```
// C++
class T_out
{
  public:
   T_out(T*& p) : ptr_(p) { ptr_ = 0; }
   T_out(T_var& p) : ptr_(p.ptr_) {
      delete ptr_;
      ptr_ = 0;
   }
   T_out(T_out& p) : ptr_(p.ptr_) {}
   T_out& operator=(T_out& p) {
      ptr_ = p.ptr_;
      return *this;
   }
   T_out& operator=(T* p) { ptr_ = p; return *this; }

   operator T*&() { return ptr_; }
   T*& ptr() { return ptr_; }
```

```
    T* operator->() { return ptr_; }

  private:
    T*& ptr_;

    // assignment from T_var not allowed
    void operator=(const T_var&);
};
```

Alternative Mappings For C++ Dialects *D*

This appendix describes alternative mappings for C++ dialects that do not match the assumptions specified in Section 15.1.2, "C++ Implementation Requirements," on page 15-2. Conforming implementations do not have to provide these workarounds if their C++ compiler supports the required features.

D.1 64-bit Integers

IDL translators that support 64-bit integer types should map the signed type to **`LongLong`** and the unsigned type to **`ULongLong`**, where both names are defined in the **`CORBA`** namespace.

D.2 Without Namespaces

If the target environment does not support the **`namespace`** construct but does support nested classes, then a module should be mapped to a C++ class. If the environment does not support nested classes, then the mapping for modules should be the same as for the CORBA C mapping (concatenating identifiers using an underscore ("_") character as the separator).

Note that module constants map to file-scope constants on systems that support namespaces and class-scope constants on systems that map modules to classes.

D.3 Without Exception Handling

For those C++ environments that do not support real C++ exception handling, referred to here as *non-exception handling (non-EH) C++ environments*, an **`Environment`** parameter passed to each operation is used to convey exception information to the caller.

As shown in "Environment" on page 17-3, the **`Environment`** class supports the ability to access and modify the **`Exception`** it holds.

As shown in Section 16.15, "Mapping for Exception Types," on page 16-47, both user-defined and system exceptions form an inheritance hierarchy that normally allow types to be caught either by their actual type or by a more general base type. When used in a non-EH C++ environment, the narrowing functions provided by this hierarchy allow for examination and manipulation of exceptions:

```
// IDL
interface A
{
    exception Broken { ... };
    void op() raises(Broken);
};

// C++
Environment ev;
A_ptr obj = ...
obj->op(ev);
if (Exception *exc = ev.exception()) {
   if (A::Broken *b = A::Broken::_narrow(exc)) {
      // deal with user exception
   } else {
      // must have been a system exception
      SystemException *se = SystemException::_narrow(exc);
      ...
   }
}
```

Section 17.10, "ORB," on page 17-16 specifies that **`Environment`** must be created using **`ORB::create_environment`**, but this is overly constraining for implementations requiring an **`Environment`** to be passed as an argument to each method invocation. For implementations that do not support real C++ exceptions, **`Environment`** may be allocated as a static, automatic, or heap variable. For example, all of the following are legal declarations on a non-EH C++ environment:

```
// C++
Environment global_env;       // global
static Environment static_env;// file static

class MyClass
{
  public:
    ...
  private:
    static Environment class_env;    // class static
};

void func()
{
   Environment auto_env;                       // auto
   Environment *new_env = new Environment;     // heap
   ...
}
```

For ease of use, **Environment** parameters are passed by reference in non-EH environments:

```
// IDL
interface A
{
    exception Broken { ... };
    void op() raises(Broken);
};

// C++
class A ...
{
  public:
   void op(Environment &);
   ...
};
```

For additional ease of use in non-EH environments, **Environment** should support copy construction and assignment from other **Environment** objects. These additional features are helpful for propagating exceptions from one **Environment** to another under non-EH circumstances.

When an exception is "thrown" in a non-EH environment, object implementors and ORB runtimes must ensure that all **out** and return pointers are returned to the caller as null pointers. If non-initialized or "garbage" pointer values are returned, client application code could experience runtime errors due to the assignment of bad pointers to **T_var** types. When a **T_var** goes out of scope, it attempts to **delete** the **T*** given to it; if this pointer value is garbage, a runtime error will almost certainly occur.

Exceptions in non-EH environments need not support the virtual **_raise()** function, since the only useful implementation of it in such an environment would be to abort the program.

C++ Keywords G

Table 1 lists all C++ keywords from the 2 December 1996 Working Paper of the ANSI

TBL. 1 C++ Keywords

and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch
char	class	compl	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	not	not_eq	operator
or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t
while	xor	xor_eq		

(X3J16) C++ Language Standardization Committee.