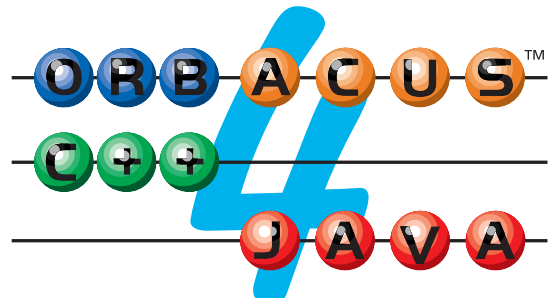




CORBA/C++ Programming with ORBacus

Student Workbook

Version 1.0.7
Printed in USA
September 2001
IONA Technologies, Inc.



Copyright © 2000–2001 IONA Technologies

Parts of this material are adapted from M. Henning/S. Vinoski, *Advanced CORBA Programming with C++*. © 1999 Addison Wesley Longman, Inc. Reprinted by permission of Addison Wesley Longman.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in these notes and Object Oriented Concepts was aware of the trademark claim, the designations have been printed in initial caps or all caps.

Object Oriented Concepts, Inc. has taken care in the preparation of this material, but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained therein.

ROYALTY-FREE PUBLIC LICENSE AGREEMENT FOR ORBACUS TRAINING COURSE MATERIALS

READ CAREFULLY: This License Agreement for ORBacus Training Course Materials (“License”) is a legal agreement between you, the Licensee, (either an individual or a single entity) and IONA Technologies (“IONA”) for non-commercial, royalty-free use of the Materials. Any commercial use is subject to a different license. By using the Materials, Licensee indicates acceptance of this License, and agrees to be bound by all its terms and conditions for using the Materials.

No rights are granted to the Materials except as expressly set forth herein. Nothing other than this License grants Licensee permission to use the Materials. Licensee may not use the Materials except as expressly provided under this License. If Licensee does not accept the terms and conditions of this License, do not use the Materials.

In consideration of Licensee's forbearance of commercial use of the Materials, IONA grants Licensee non-exclusive, royalty-free rights as expressly provided herein.

DEFINITIONS.

The “Materials” are IONA’s training course materials for its ORBacus software.

To “distribute” means to broadcast, publish, transfer, post, upload, download or otherwise disseminate in any medium to any third party.

To “modify” means to create a work derived from the Materials.

A “commercial use” is the use of the Materials in connection with, for or in aid of the generation of revenue, such as in the conduct of Licensee's daily business operations.

LICENSE TO USE.

Licensee may use the Materials provided that such use does not constitute a commercial use. Licensee shall not copy or distribute the Materials. Licensee shall not modify the Materials.

Notwithstanding the restrictions on commercial use and copying, if Licensee is an accredited academic institution, Licensee may use the Materials in courses provided at Licensee, and may provide copies of the Materials to the attendees at such courses, provided that there is no separate charge for such Materials other than general tuition for attendance at such institution.

RESTRICTIONS.

Licensee acknowledges that the Materials are protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The Materials are licensed, not sold. All title and copyrights in and to the Materials and any copies of the Materials are owned exclusively by IONA. The Materials incorporate, with the permission of the publisher, certain material from “Advanced CORBA Programming with C++” published by Addison Wesley Longman. Licensee may not sublicense, assign or transfer this License or the Materials.

NO WARRANTIES.

IONA EXPRESSLY DISCLAIMS ANY WARRANTY FOR THE MATERIALS. THE MATERIALS ARE PROVIDED TO LICENSEE “AS IS,” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE ENTIRE RISK AS

TO THE USE, QUALITY AND PERFORMANCE OF THE MATERIALS IS WITH THE LICENSEE. SHOULD THE MATERIALS PROVE DEFECTIVE, LICENSEE ASSUMES THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

NO LIABILITY FOR DAMAGES.

IN NO EVENT WILL IONA BE LIABLE FOR ANY GENERAL, DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, INACCURATE INFORMATION, LOSS OF INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OR INABILITY TO USE THE MATERIALS, EVEN IF IONA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT RESTRICTED RIGHTS.

The Materials are provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the commercial Computer Software-Restricted Rights 48 C.F.R. paragraph 52.227-19, as applicable. Manufacturer is IONA Technologies, Inc., 200 West Street, Waltham, MA 02451.

TERMINATION.

Any violation or any attempt to violate any of the terms and conditions of this License will automatically terminate Licensee's rights under this License. Upon such termination Licensee will cease any and all use of the Materials and will destroy any and all of Licensee's copies of the Materials.

LICENSE SCOPE AND MODIFICATION.

This License sets forth the entire agreement between Licensee and IONA and supersedes all prior agreements and understandings between the parties related to the subject matter hereof. None of the terms of this License may be waived or modified except as expressly agreed in writing by both Licensee and IONA.

SEVERABILITY.

Should any provision of this License be declared void or unenforceable, the validity of the remaining provisions shall not be affected thereby.

GOVERNING LAW.

This License is governed by the laws of the Commonwealth of Massachusetts, U.S.A., and shall be interpreted in accordance with and governed by the laws thereof. Licensee hereby waives any and all right to assert a defense based on jurisdiction and venue for any action stemming from this License brought in U.S. District Court for the District of Massachusetts. Should Licensee have any questions concerning this License, or if Licensee desires to contact IONA for any reason, please contact at:

IONA Technologies, Inc.
200 West Street
Waltham, MA 02451
USA

Contents

Unit 1: Introduction	1-1
1.1 What is CORBA?	1-2
1.2 The Object Management Group (OMG)	1-3
1.3 What is Client/Server Computing?	1-4
1.4 Advantages and Disadvantages of CORBA	1-5
1.5 Heterogeneity	1-6
1.6 The Object Management Architecture (OMA)	1-8
1.7 Core Components of an ORB	1-10
1.8 Request Invocation	1-12
1.9 Object Reference Semantics	1-14
Unit 2: The OMG Interface Definition Language	2-1
2.1 Introduction	2-2
2.2 IDL Compilation (C++)	2-4
2.3 IDL Compilation (Mixed Languages)	2-6
2.4 IDL Source Files	2-7
2.5 Comments and Keywords	2-8
2.6 Identifiers	2-9
2.7 Built-In Types	2-10
2.8 Type Definitions	2-15
2.9 Enumerations	2-16
2.10 Structures	2-18
2.11 Unions	2-20
2.12 Guidelines for Unions	2-22
2.13 Arrays	2-24
2.14 Sequences	2-25
2.15 Sequences or Arrays?	2-26
2.16 Recursive Types	2-28
2.17 Constants and Literals	2-30
2.18 Constant Expressions	2-33

2.19	Interfaces	2-34
2.20	Interface Syntax	2-35
2.21	Interface Semantics	2-36
2.22	Operation Syntax	2-37
2.23	Operation Example	2-38
2.24	User Exceptions	2-40
2.25	Using Exceptions Effectively	2-41
2.26	System Exceptions	2-42
2.27	Oneway Operations	2-44
2.28	Contexts	2-45
2.29	Attributes	2-46
2.30	Modules	2-47
2.31	Forward Declarations	2-48
2.32	Inheritance	2-49
2.33	Inheritance from Object	2-50
2.34	Inheritance Redefinition Rules	2-51
2.35	Inheritance Limitations	2-52
2.36	Multiple Inheritance	2-53
2.37	Scope Rules for Multiple Inheritance	2-54
2.38	IDL Scope Resolution	2-56
2.39	Nesting Restrictions	2-59
2.40	Anonymous Types	2-60
2.41	Repository IDs	2-63
2.42	Controlling Repository ID Prefixes	2-64
2.43	Predefined IDL	2-65
2.44	Using the IDL Compiler	2-66
2.45	Topics Not Covered Here	2-67
Unit 3: Exercise: Writing IDL Definitions		3-1
<hr/>		
3.1	The Climate Control System	3-2
3.2	Thermometers	3-3
3.3	Thermostats	3-4
3.4	The Monitoring Station	3-5
3.5	What You Need to Do	3-6
Unit 4: Solution: Writing IDL Definitions		4-1
<hr/>		
4.1	IDL for the Climate Control System	4-2
Unit 5: Basic C++ Mapping		5-1
<hr/>		
5.1	Introduction	5-2
5.2	Mapping for Identifiers	5-3
5.3	Scoping Rules	5-4
5.4	Mapping for Modules	5-5
5.5	Mapping for Built-In Types	5-6
5.6	Overloading on Built-In Types	5-8

5.7	Memory Allocation for Strings	5-9
5.8	Mapping for Constants	5-10
5.9	Variable-Length Types	5-12
5.10	Example: String Allocation	5-13
5.11	<code>_var</code> Types	5-14
5.12	C++ Mapping Levels	5-15
5.13	The <code>String_var</code> Class	5-16
5.14	Main Rules for Using <code>String_var</code>	5-20
5.15	Mapping for Fixed-Length Structures	5-23
5.16	Mapping for Variable-Length Structures	5-24
5.17	Mapping for Unbounded Sequences	5-26
5.18	Example: Using a String Sequence	5-30
5.19	Using Complex Element Types	5-31
5.20	Mapping for Bounded Sequences	5-32
5.21	Rules for Safe Use of Sequences	5-33
5.22	Mapping for Arrays	5-34
5.23	Array Assignment and Allocation	5-36
5.24	Mapping for Unions	5-38
5.25	Using Unions Safely	5-45
5.26	Mapping for <code>typedef</code>	5-46
5.27	Type <code>any</code> : Concepts	5-47
5.28	Applications of Type <code>any</code>	5-48
5.29	Mapping for Type <code>any</code>	5-49
5.30	Using <code>_var</code> Types	5-58
5.31	Mapping for Variable-Length <code>_var</code> Types	5-60
5.32	Example: Simple Use of <code>_var</code> Types	5-62
5.33	Mapping for Fixed-Length <code>_var</code> Types	5-63
5.34	Dealing with Broken Compilers	5-64

Unit 6: Client-Side C++ Mapping **6-1**

6.1	Introduction	6-2
6.2	Object References	6-3
6.3	Client-Side Proxies	6-4
6.4	Mapping for Interfaces	6-5
6.5	Mapping for Object References	6-6
6.6	Life Cycle of Object References	6-7
6.7	Reference Life Cycle Operations	6-8
6.8	Object Reference Counts	6-9
6.9	Scope of Object References	6-12
6.10	Nil References	6-13
6.11	References and Inheritance	6-14
6.12	Implicit Widening of <code>_ptr</code> References	6-15
6.13	Widening with <code>_duplicate</code>	6-16
6.14	Narrowing Conversion	6-18
6.15	Illegal Uses of References	6-20
6.16	Pseudo Objects and the ORB Interface	6-22
6.17	ORB Initialization	6-24
6.18	Stringified References	6-26

6.19	The Object Interface	6-32
6.20	Object Reference Equivalence	6-34
6.21	Providing Object Equivalence Testing	6-35
6.22	_var References	6-36
6.23	_var References and Widening	6-39
6.24	References Nested in Complex Types	6-40
6.25	Mapping for Operations	6-41
6.26	Mapping for Attributes	6-42
6.27	Parameter Passing	6-43
6.28	Parameter Passing: Pitfalls	6-58
6.29	Mapping for Exceptions	6-61
6.30	Mapping for System Exceptions	6-64
6.31	Semantics of System Exceptions	6-66
6.32	Mapping for User Exceptions	6-68
6.33	Compiling and Linking	6-69
Unit 7: Exercise: Writing a Client		7-1
<hr/>		
7.1	Source Files and Build Environment	7-2
7.2	Server Operation	7-2
7.3	Client Operation	7-3
7.4	What You Need to Do	7-3
Unit 8: Solution: Writing a Client		8-1
<hr/>		
8.1	Communicating with the Thermostat	8-2
8.2	Communicating with the Controller	8-4
8.3	The Complete Client Code	8-6
Unit 9: Server-Side C++ Mapping		9-1
<hr/>		
9.1	Introduction	9-2
9.2	Mapping for Interfaces	9-3
9.3	Skeleton Classes	9-4
9.4	Servant Classes	9-6
9.5	Operation Implementation	9-8
9.6	Attribute Implementation	9-9
9.7	Servant Activation and Reference Creation	9-10
9.8	Server Initialization	9-12
9.9	Parameter Passing	9-15
9.10	Throwing Exceptions	9-28
9.11	Exception Pitfalls	9-30
9.12	Tie Classes	9-33
9.13	Clean Server Shutdown	9-34
9.14	Handling Signals (UNIX)	9-38
9.15	Handling Signals (Windows)	9-40
9.16	Implementation Inheritance	9-41
9.17	Interface Inheritance	9-42

9.18 Compiling and Linking	9-43
----------------------------	------

Unit 10: Exercise: Writing a Server **10-1**

10.1 Source Files and Build Environment	10-2
10.2 Server Operation	10-2
10.3 What You Need to Do	10-3

Unit 11: Solution: Writing a Server **11-1**

11.1 Solution	11-2
11.2 The <code>server.h</code> File	11-6
11.3 The <code>server.cpp</code> File	11-9

Unit 12: The Portable Object Adapter (POA) **12-1**

12.1 Interface Overview	12-2
12.2 Functions of a POA	12-4
12.3 Functions of a POA Manager	12-5
12.4 POA Manager State Transitions	12-6
12.5 Request Flow	12-8
12.6 Contents of an Object Reference	12-9
12.7 Policies	12-10
12.8 POA Policies	12-12
12.9 POA Creation	12-14
12.10 POA-to-POA Manager Relationship	12-17
12.11 The Life Span Policy	12-18
12.12 The ID Assignment Policy	12-19
12.13 The Active Object Map (AOM)	12-20
12.14 The ID Uniqueness Policy	12-21
12.15 The Servant Retention Policy	12-22
12.16 The Request Processing Policy	12-23
12.17 The Implicit Activation Policy	12-24
12.18 The Thread Policy	12-25
12.19 The Root POA Policies	12-26
12.20 Policy Creation	12-28
12.21 Creating Persistent Objects	12-30
12.22 Creating a Simple Persistent Server	12-32
12.23 Explicit Servant Activation	12-36
12.24 Object Creation	12-40
12.25 Destroying CORBA Objects	12-42
12.26 Deactivation and Servant Destruction	12-48

Unit 13: Exercise: Writing a Persistent Server **13-1**

13.1 Source Files and Build Environment	13-2
13.2 Server Operation	13-2
13.3 What You Need to Do	13-3

Unit 14: Solution: Writing a Persistent Server **14-1**

14.1 Solution 14-2

Unit 15: Advanced Uses of the POA **15-1**

15.1 Pre-Loading of Objects 15-2
15.2 Servant Managers 15-3
15.3 Servant Activators 15-4
15.4 Implementing a Servant Activator 15-6
15.5 Use Cases for Servant Activators 15-8
15.6 Servant Manager Registration 15-9
15.7 Type Issues with Servant Managers 15-10
15.8 Servant Locators 15-11
15.9 Implementing Servant Locators 15-12
15.10 Use Cases for Servant Locators 15-14
15.11 Servant Managers and Collections 15-16
15.12 One Servant for Many Objects 15-18
15.13 The Current Object 15-20
15.14 Default Servants 15-22
15.15 Trade-Offs for Default Servants 15-24
15.16 POA Activators 15-25
15.17 Implementing POA Activators 15-26
15.18 Registering POA Activators 15-28
15.19 Finding POAs 15-30
15.20 Identity Mapping Operations 15-32

Unit 16: Exercise: Writing Servant Locators **16-1**

16.1 Source Files and Build Environment 16-2
16.2 Server Operation 16-2
16.3 What You Need to Do 16-2

Unit 17: Solution: Writing Servant Locators **17-1**

17.1 Solution 17-2

Unit 18: ORBacus Configuration **18-1**

18.1 Introduction 18-2
18.2 Defining Properties 18-3
18.3 Setting Properties in the Registry 18-4
18.4 Setting Properties in a Configuration File 18-5
18.5 Setting Properties Programmatically 18-6
18.6 Setting Properties from the Command Line 18-7
18.7 Commonly Used Properties 18-8

Unit 19: The Naming Service	19-1
19.1 Introduction	19-2
19.2 Terminology	19-3
19.3 Example Naming Graph	19-4
19.4 Naming IDL Structure	19-6
19.5 Name Representation	19-7
19.6 Stringified Names	19-8
19.7 Pathnames and Name Resolution	19-9
19.8 Obtaining an Initial Naming Context	19-10
19.9 Naming Service Exceptions	19-12
19.10 Creating and Destroying Contexts	19-14
19.11 Creating Bindings	19-16
19.12 Context Creation Example	19-18
19.13 Rebinding	19-21
19.14 Resolving Bindings	19-22
19.15 Removing Bindings	19-24
19.16 Listing Name Bindings	19-26
19.17 Pitfalls in the Naming Service	19-29
19.18 Stringified Name Syntax	19-30
19.19 Using Stringified Names	19-31
19.20 URL-Style IORs	19-32
19.21 URL Escape Sequences	19-35
19.22 Resolving URL-Style IORs	19-36
19.23 Creating URL-Style IORs	19-37
19.24 What to Advertise	19-38
19.25 Federated Naming	19-39
19.26 Running the Naming Service	19-40
19.27 The <code>nsadmin</code> Tool	19-41
19.28 Compiling and Linking	19-42
Unit 20: Exercise: Using the Naming Service	20-1
20.1 Source Files and Build Environment	20-2
20.2 Server Operation	20-2
20.3 What You Need to Do	20-2
Unit 21: Solution: Using the Naming Service	21-1
21.1 Solution	21-2
Unit 22: The Implementation Repository (IMR)	22-1
22.1 Purpose of an Implementation Repository	22-2
22.2 Binding	22-4
22.3 Indirect Binding	22-6
22.4 Automatic Server Start-Up	22-8
22.5 IMR Process Structure	22-9

22.6	Location Domains	22-10
22.7	The imradmin Tool	22-11
22.8	Server Execution Environment	22-12
22.9	Server Attributes	22-14
22.10	Getting IMR Status	22-16
22.11	IMR Configuration	22-18
22.12	IMR Properties	22-20
22.13	The Boot Manager	22-22
22.14	The mkref Tool	22-23
Unit 23: Exercise: Using the Implementation Repository		23-1
<hr/>		
23.1	Source Files and Build Environment	23-2
23.2	Server Operation	23-2
23.3	What You Need to Do	23-2
Unit 24: Solution: Using the Implementation Repository		24-1
<hr/>		
24.1	Solution	24-2
Unit 25: Threaded Clients and Servers		25-1
<hr/>		
25.1	Overview	25-2
25.2	The Blocking Concurrency Model	25-3
25.3	The Reactive Concurrency Model	25-4
25.4	The Threaded Concurrency Model	25-7
25.5	The Thread-per-Client Concurrency Model	25-8
25.6	The Thread-per-Request Concurrency Model	25-9
25.7	The Thread-Pool Concurrency Model	25-10
25.8	Selecting a Concurrency Model	25-12
25.9	Overview of JThreads/C++	25-14
25.10	JTC Initialization	25-15
25.11	Simple Mutexes	25-16
25.12	Recursive Mutexes	25-17
25.13	Automatic Unlocking	25-18
25.14	Monitors	25-19
25.15	Simple Producer/Consumer Example	25-20
25.16	Rules for Using Monitors	25-22
25.17	Static Monitors	25-26
25.18	The <code>JTCThread</code> Class	25-28
25.19	Joining with Threads	25-30
25.20	Other JThreads/C++ Functionality	25-31
25.21	Synchronization Strategies for Servers	25-32
25.22	Basic Per-Servant Synchronization	25-33
25.23	Life Cycle Considerations	25-34
25.24	Threading Guarantees for the POA	25-39

1. Introduction

Summary

This unit presents the motivation for using CORBA, the basics of the CORBA architecture, and the fundamentals of the CORBA object model, including the semantics of object references.

Objectives

By the completion of this unit, you will have a basic understanding of CORBA's advantages and disadvantages, how an ORB helps you to develop distributed applications, the basic functions of an ORB, and the semantics of request dispatch.

What is CORBA?

CORBA (Common Object Request Broker Architecture) is a distributed object-oriented client/server platform.

It includes:

- an object-oriented Remote Procedure Call (RPC) mechanism
- object services (such as the Naming or Trading Service)
- language mappings for different programming languages
- interoperability protocols
- programming guidelines and patterns

CORBA replaces ad-hoc special-purpose mechanisms (such as socket communication) with an open, standardized, scalable, and portable platform.



1
Introduction
Copyright 2000–2001 IONA Technologies



1.1 What is CORBA?

Fundamentally, the Common Object Request Broker Architecture (CORBA) is a distributed client/server platform with an object-oriented spin. The idea is to provide an object-oriented programming model for distributed computing to programmers that is as close as possible to programming with ordinary local objects. The job of CORBA is take all the grunt work out of distributed programming, so you can focus on your business logic instead of having to worry about distribution infrastructure.

CORBA consists of a large set of specifications that run to thousands of pages. However, the fundamental services it provides can be summarized as above. Apart from an RPC mechanism, CORBA offers a number of services that take care of common chores, provides language bindings for a number of popular programming languages, defines an interoperability protocol so implementations from different vendors can interoperate, and it defines a number of programming guidelines and patterns (often enshrined in specific APIs) that you use when you develop applications.

CORBA enables you to get away from having to worry about infrastructure and ad-hoc and home-grown communication mechanism and replaces them with a an open and standardized platform that is both portable and scalable.

The Object Management Group (OMG)

The OMG was formed in 1989 to create specifications for open distributed computing.

Its mission is to

"... establish industry guidelines and object management specifications to provide a common framework for distributed application development."

The OMG is the world's largest software consortium with more than 800 member organizations.

Specifications published by the OMG are free of charge. Vendors of CORBA technology do not pay a royalty to the OMG.

Specifications are developed by consensus of interested submitters.



2
Introduction
Copyright 2000–2001 IONA Technologies



1.2 The Object Management Group (OMG)

The Object Management Group (OMG) publishes specifications of technology that are meant to enable the development of distributed object-oriented applications. (By and large, the specifications actually achieve this goal, although, as with any other large effort of this kind, there are a few bad apples among the bunch.)

The specifications are submitted for technology adoption by members of the OMG who have an interest in a specific technology. The specifications are developed and refined by a process of consensus decision making. Of the more than 800 member organizations, none can dominate or otherwise unduly influence the process. Once published, specifications are made available free of charge, and anyone can develop products based on these specifications free of charge.

What is Client/Server Computing?

A client/server computing system has the following characteristics:

- A number of clients and servers cooperate to carry out a computational task.
- Servers are passive entities that offer a service and wait for requests from clients to perform that service.
- Clients are active entities that obtain service from servers.
- Clients and servers usually run as processes on different machines (but may run on a single machine or even within a single process).
- Object-oriented client/server computing adds OO features to the basic distribution idea: interfaces, messages, inheritance, and polymorphism.



3
Introduction
Copyright 2000–2001 IONA Technologies



1.3 What is Client/Server Computing?

Fundamentally, client/server computing is about different and distinct computational entities (tasks, threads, processes, computers, systems) that cooperate to get some work done. The entities are servers (which are passive and offer service), and clients (which are active and obtain service). The same entity can act as both a client and a server at different times or even simultaneously.

A very simple example of a client/server system is the UNIX print spooler. The **lpsched** (or **lpd**) process is a server that runs permanently and waits for instructions from clients to print something; the **lp** (or **lpr**) command is a client that contacts the server with a print request. Even though the communication between the two is very simple, the print spooler exhibits all the fundamental characteristics of a client/server system.

Adding object-oriented features to client/server computing means to transparently support the fundamental principles of object orientation: the separation of interfaces from implementations, inheritance, and polymorphism. These features mean that you get object-orientation that "extends across the wire", which means that you can access a remote object much as if it were local. (This is in sharp contrast to client/server platforms such as DCE, which has no real notion of objects or polymorphism and makes it very hard to naturally extend an OO programming model to distributed systems.)

Advantages and Disadvantages of CORBA

Some advantages:

- vendor-neutral and open standard, portable, wide variety of implementations, hardware platforms, operating systems, languages
- takes the grunt work out of distributed programming

Some disadvantages:

- no reference implementation
- specified by consensus and compromise
- not perfect
- can shoot yourself in the foot and blow the whole leg off...

Still, it's the best thing going!



4
Introduction
Copyright 2000–2001 IONA Technologies



1.4 Advantages and Disadvantages of CORBA

CORBA offers quite a few advantages. Among them is the fact that CORBA is not proprietary technology, so you get implementations from a large number of vendors for almost every imaginable combination of hardware, operating system, and programming language. Quite fierce competition among vendors ensures that you have a choice, while the specifications ensure interoperability and portability. You can even get Open Source implementations free of charge.

CORBA makes distributed programming easier than any other platform in existence. Most of the low-level and difficult work required for distribution is taken care of for you, so you can concentrate on your application instead of distributed programming. (However, that doesn't mean that you can forget that a network is somewhere between the client and server, only that you don't have to deal with that network directly.)

On the down side, CORBA suffers from a few problems too. Because the OMG publishes specifications, not source code, there is no reference implementation that would definitively state what CORBA is. This means that specifications are sometimes too loose or ambiguous and permit implementation behavior to diverge until the OMG catches up with the problem and fixes the specification. Consensus decision making is also not necessarily the best way to establish a specification. While the specifications usually do what most parties want, they are typically not as elegant or tight as they could be, due to the need to accommodate everyone's needs. And, as with any powerful and complex tool, it is easy to build something that doesn't work very well. You still need to know what you are doing and CORBA cannot do your thinking for you.

Still, CORBA is by far the most successful and widely-used distributed client/server technology in existence. Once you know it, you will enjoy it!

Heterogeneity

CORBA can deal with homogeneous and heterogeneous environments. The main characteristics to support heterogeneous systems are:

- location transparency
- server transparency
- language independence
- implementation independence
- architecture independence
- operating system independence
- protocol independence
- transport independence



5
Introduction
Copyright 2000–2001 IONA Technologies



1.5 Heterogeneity

CORBA hides most of the differences that are present in heterogeneous systems, which are composed of components from different vendors that use different technologies. Specifically, CORBA provides the following features:

- Location transparency

The client does not know or care whether the target object is local to its own address space, is implemented in a different process on the same machine, or is implemented in a process on a different machine. Server processes are not obliged to remain on the same machine forever; they can be moved around from machine to machine without clients becoming aware of it (with some constraints, which we discuss in Unit 22).

- Server transparency

The client does not need to know which server implements which objects.

- Language independence

The client does not care what language is used by the server. For example, a C++ client can call a Java implementation without being aware of it. The implementation language for objects can be changed for existing objects without affecting clients.

- Implementation independence

The client does not know how the implementation works. For example, the server may implement its objects as proper C++ objects, or the server may actually implement its objects using non-OO techniques (such as implementing objects as lumps of data). The client sees the

same consistent object-oriented semantics regardless of how objects are implemented in the server.

- Architecture independence

The client is unaware of the CPU architecture that is used by the server and is shielded from such details as byte ordering and structure padding.

- Operating system independence

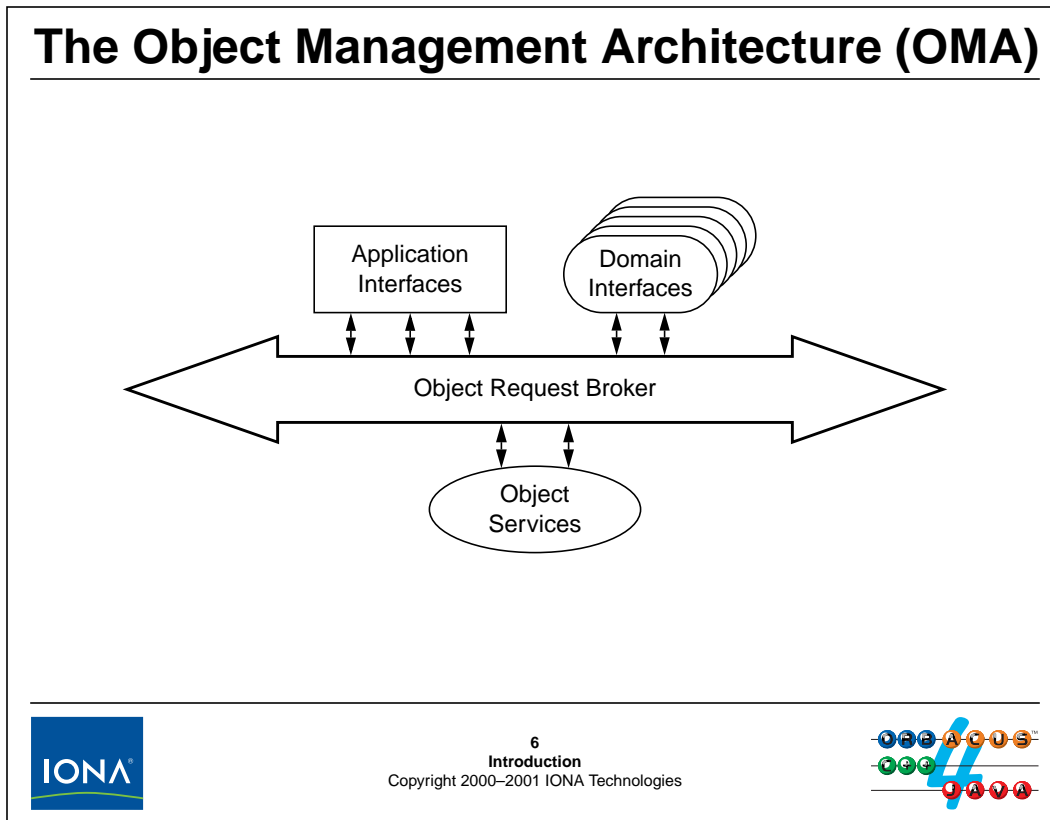
The client does not care what operating system is used by the server. The server may even be implemented without the support of an operating system—for example, as a real-mode embedded program.

- Protocol independence

The client does not know what communication protocol is used to send messages. If several protocols are available to communicate with the server, the ORB transparently selects a protocol at run time.

- Transport independence

The client is ignorant of the transport and data link layer used to transmit messages. ORBs can transparently use various networking technologies such as Ethernet, ATM, token ring, or serial lines.



1.6 The Object Management Architecture (OMA)

The Object Management Architecture (OMA) and its core, the CORBA specification—provide a complete architectural framework that is both rich enough and flexible enough to accommodate a wide variety of distributed systems.

The OMA uses two related models to describe how distributed objects and the interactions between them can be specified in platform-independent ways. The Object Model defines how the interfaces of objects distributed across a heterogeneous environment are described using an Interface Definition Language (IDL), and the Reference Model characterizes interactions between such objects.

The Object Model defines an object as an *encapsulated entity* with an *immutable distinct identity* whose services are accessed only through well-defined *interfaces*. Clients use an object's services by issuing *requests* to the object. The implementation details of the object and its location are kept hidden from clients.

The Reference Model provides *interface categories* that are general groupings for object interfaces. As the above diagram shows, all interface categories are conceptually linked by an Object Request Broker (ORB). Generally, an ORB enables communication between clients and objects, transparently activating those objects that are not running when requests are delivered to them. The ORB also provides an interface that can be used directly by clients as well as objects.

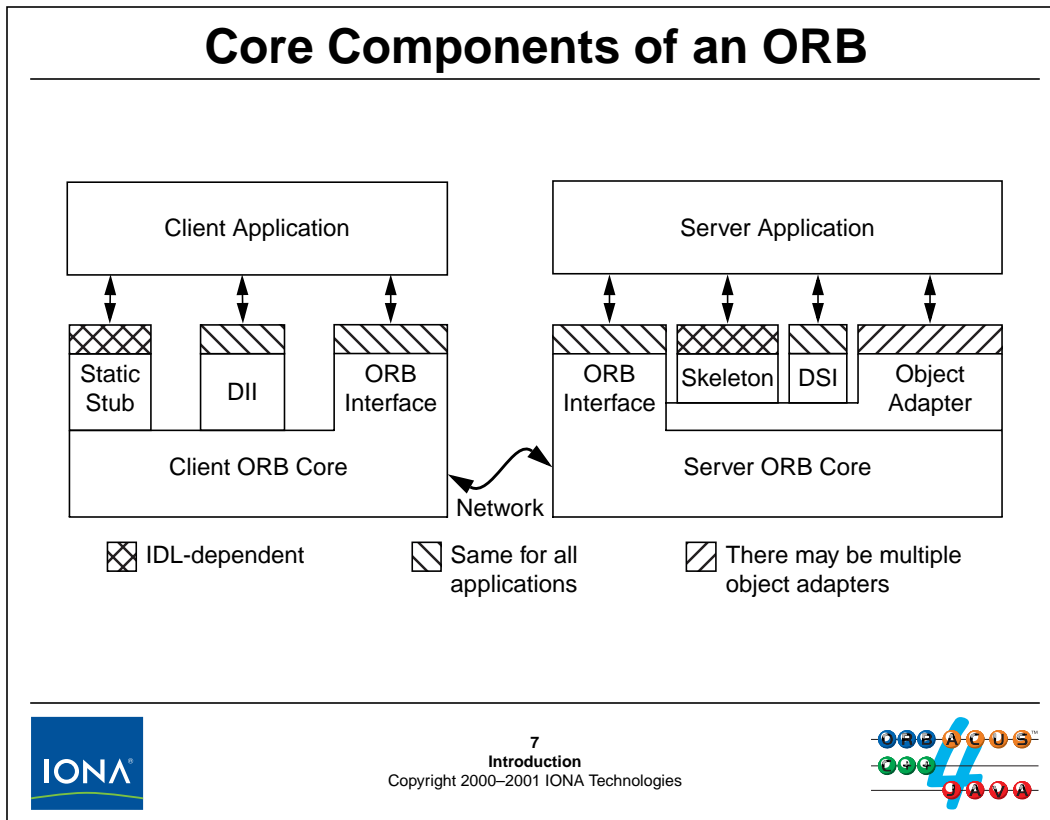
The above diagram shows the interface categories that use the ORB's activation and communication facilities.

- *Object Services* are domain-independent, or *horizontally oriented*, interfaces used by many distributed object applications. For example, all applications must obtain references to the

objects they intend to use. Both the OMG Naming Service and the OMG Trading Service are object services that allow applications to look up and discover object references. Object services are normally considered part of the core distributed computing infrastructure.

- *Domain Interfaces* play roles similar to those in the Object Services category except that domain interfaces are domain-specific, or *vertically oriented*. For example, there are domain interfaces used in health care applications that are unique to that industry, such as a Person Identification Service. Other interfaces are specific to finance, manufacturing, telecommunications, and other domains. The multiple Domain Interface bubbles in the preceding diagram indicate this multiplicity of domains.
- *Application Interfaces* are developed specifically for a given application. They are not standardized by the OMG. However, if certain application interfaces begin to appear in many different applications, they become candidates for standardization in one of the other interface categories.

Note that none of the interfaces in these categories are privileged in any way. For example, you do not need access to the source code for your ORB to use a naming service other than the OMG-defined one. For example, you can implement a naming service of your own that implements the OMG interfaces for the service, or you can make up an entirely different naming service with different interfaces and use that instead. (Some of the other components that access the ORB may expect a standards-conforming naming service to be present, so changing one component may force you to change others; however, such interdependencies are rare.)



1.7 Core Components of an ORB

The above diagram shows the major components of an ORB:

- ORB Core

The ORB core is proprietary to each implementation and encapsulates basic networking facilities. Application code does not ever access the ORB core directly, and the proprietary interfaces and features of the core are hidden behind a facade of standardized APIs.

- ORB Interface

On top of the core, the ORB interface allows clients and servers to communicate with the core. The ORB interface is standardized and the same for all clients and servers and exists mainly for initialization purposes and a few other basic services.

- Static Stub

The purpose of a static stub is to accept a client request and to pass it to the ORB core, which in turn sees to it that the request makes its way to its (possibly remote) target object. The stub is a piece of code that is generated by an Interface Definition Language (IDL) compiler. The way IDL is compiled into a language-specific API is standardized by language mapping specifications. This means that the static stubs offer the same interface on all ORBs. However, the specific API is dependent on the type of object being accessed. For example, a person object has a different interface than a car object, and the difference is reflected in the generated API.

- Dynamic Invocation Interface (DII)

The Dynamic Invocation Interface (DII) provides an alternate way to access remote objects. Instead of being specific to a specific interface, the DII is general enough to allow you to send a request to any type of object, even an object whose interface was unknown when the client was compiled (no IDL-generated code is involved).

The main advantage of the DII is that it does not require compile-time knowledge of the types of objects a client can communicate with and is therefore very flexible. The DII is usually used to implement dynamic applications, such as debuggers and browsers, that cannot have compile-time knowledge of all the interfaces they need to deal with. The downside of the DII is that it is much more complex than the static stub interface.

The DII interface is identical for all ORBs.

- Skeleton

The skeleton is the server-side equivalent of a stub. It is generated from IDL (and therefore specific to each object type) and provides an up-call interface into the application code. In effect, it provides a callback mechanism so you can implement the behavior of your objects in the server.

- Dynamic Skeleton Interface (DSI)

The Dynamic Skeleton Interface (DSI) is the server-side equivalent of the DII. It permits you to write a server that implements objects whose type is unknown at the time the server is written. This sounds almost like a contradiction in terms, but makes sense when you think about things like protocol bridges. A bridge must accept requests for objects whose types it has never seen before and pass them on via some other protocol, translating these requests on the fly. The type knowledge necessary for this translation is supplied to the bridge in form of dynamic configuration information, or taken from an Interface Repository (IFR), which is a database of IDL definitions.

- Object Adapter

The object adapter mediates calls between the ORB and the server and serves two main purposes:

- It keeps track of how to map incoming requests onto programming language artifacts, such as objects or procedures. To do this, the object adapter must know which objects exist and when objects are created or destroyed. The object adapter therefore offers APIs that allow the application code to keep it informed of the life cycle of objects. The object adapter is also involved in creating and tracking the identity of objects.
- The object adapter straddles the boundary between language-independent requests received from clients and language-dependent up-calls that need to be made to pass control to the application code in the server. For this reason, object adapters have some components that differ for each language mapping.

Only one object adapter, the Portable Object Adapter (POA) is currently standardized. (The Basic Object Adapter (BOA) was deprecated with CORBA 2.2 and is no longer part of the standard.) However, vendors can create other object adapters for special purposes, for example, for real-time systems or object-oriented databases.

Note that only two processes are involved in the preceding diagram: one client and one server. In particular, there is no other process via which communications are routed. The run-time support for clients and servers is provided entirely in libraries.

Request Invocation

Clients invoke requests (send messages) to objects via an object reference. The object reference (IOR) identifies the target object.

When a request is sent by a client, the ORB:

- locates the target object
- activates the server if it is not running
- transmits arguments for the request to the server
- activates the target object (servant) in the server if it is not instantiated
- waits for the request to complete
- returns the results of the request to the client or returns an exception if the request failed



8
Introduction
Copyright 2000–2001 IONA Technologies



1.8 Request Invocation

Objects in CORBA are identified by object references. An object reference is a handle that uniquely identifies a target object. For ORBs supporting the Internet Inter-ORB Protocol (IIOP), object references are in a standard format and known as Interoperable Object References (IORs).

For a client to send a request to an object, it must hold an object reference and invoke an operation via the reference (much like you must have a class instance pointer in C++ to invoke a member function on a C++ object). The ORB takes care of the entire request dispatch transparently, such as locating the object, starting its server if it is not running at the time, and making sure that parameters are sent and received correctly (or an exception is raised for a failed request).

The language-specific object that receives the request (that is, a C++ instance for our purposes) is known as the servant for a request.

Object Reference Semantics

An object reference is similar to a C++ class instance pointer, but can denote an object in a remote address space.

- Every object reference identifies exactly one object instance.
- Several different references can denote the same object.
- References can be nil (point nowhere).
- References can dangle (like C++ pointers that point at deleted instances).
- References are opaque.
- References are strongly typed.
- References support late binding.
- References can be persistent.



1.9 Object Reference Semantics

Object references are much like C++ class instance pointers in that they uniquely identify an object. The main difference is that an object reference can denote an object in another address space. The semantics of object references are central to the CORBA object model and must be understood in detail.

- Each reference identifies exactly one object.

Just as a C++ class instance pointer identifies exactly one object instance, an object reference denotes exactly one CORBA object (which may be implemented in a remote address space). A client holding an object reference is entitled to expect that the reference will always denote the same object while the object continues to exist. An object reference is allowed to stop working only when its target object is permanently destroyed. After an object is destroyed, its references become permanently non-functional. This means that a reference to a destroyed object cannot accidentally denote some other object later.

- An object can have several references.

Several different references can denote the same object. In other words, each reference “names” exactly one object, but an object is allowed to have several names.

If you find this strange, remember that the same thing can happen in C++. A C++ class instance pointer denotes exactly one object, and the pointer *value* (such as 0x48bf0) identifies that object. Multiple inheritance can cause a single C++ instance to have several different pointer values, depending on whether pointer points to a base or derived part of the object.

- References can be nil.

CORBA defines a distinguished nil value for object references. A nil reference points nowhere and is analogous to a C++ null pointer. Nil references are useful for conveying “not found” or “not there” semantics. For example, an operation can return a nil reference to indicate that a client’s search for an object did not locate a matching instance.

- References can dangle.

After a server has passed an object reference to a client, that reference is permanently out of the server’s control and can propagate freely via means invisible to the ORB. This means that CORBA has no built-in automatic mechanism for the server to inform a client when the object belonging to a reference is destroyed. Similarly, there is no built-in automatic way for a client to inform a server that it has lost interest in an object reference.

- References are opaque.

Object references contain a number of standardized components that are the same for all ORBs as well as proprietary information that is ORB-specific. To permit source code compatibility across different ORBs, clients and servers are not allowed to see the representation of an object reference. Instead, they must treat an object reference as a black box that can be manipulated only through a standardized interface.

The encapsulation of object references is a key aspect of CORBA. It lets you add new features, such as different communication protocols, over time without breaking existing source code. In addition, vendors can use the proprietary part of object references to provide value-added features, such as performance optimizations, without compromising interoperability with other ORBs.

- References are strongly typed.

Every object reference contains an indication of the interface supported by that reference. This arrangement allows the ORB run time to enforce type safety. For example, an attempt to send a `print` message to an `Employee` object (which does not support that operation) is caught at compile time for statically-typed languages (such as C++) and at run time, otherwise.

- References support late binding.

Clients can treat a reference to a derived object as if it were a reference to a base object. For example, assume that a `Manager` interface is derived from `Employee`. A client may actually hold a reference to a `Manager` but may think of that reference as being of type `Employee`. As in C++, a client cannot invoke `Manager` operations via an `Employee` reference. However, if a client invokes the `person_number` operation via the `Employee` reference, the corresponding message is still sent to the `Manager` servant that implements the `Employee` interface.

This arrangement is exactly analogous to C++ virtual function calls: invoking a method via a base pointer calls the virtual function in the derived instance. One of the major advantages of CORBA, compared with traditional RPC platforms, is that polymorphism and late binding work for remote objects exactly as they do for local C++ objects. This means that there is no artificial wall through your architecture in which you must map an object-oriented design onto a remote procedure call paradigm. Instead, polymorphism works transparently across the wire.

- References can be persistent.

Clients and servers can convert an object reference into a string and write the string to disk. Sometime later, that string can be converted back into an object reference that denotes the same original object.

2. The OMG Interface Definition Language

Summary

This unit presents the syntax and semantics of the OMG Interface Definition Language, including common idioms and design guidelines. The unit also shows how to use the IDL compiler to produce C++ stubs and skeletons.

Objectives

By the completion of this unit, you will be able to write IDL definitions and to compile these definitions into C++ stubs and skeletons.

Introduction

IDL specifications separate language-independent interfaces from language-specific implementations.

IDL establishes the interface contract between client and server.

Language-independent IDL specifications are compiled by an IDL compiler into APIs for a specific implementation language.

IDL is purely declarative. You can neither write executable statements in IDL nor say anything about object state.

IDL specifications are analogous to C++ type and abstract class definitions. They define types and interfaces that client and server agree on for data exchange.

You can exchange data between client and server only if the data's types are defined in IDL.



1
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



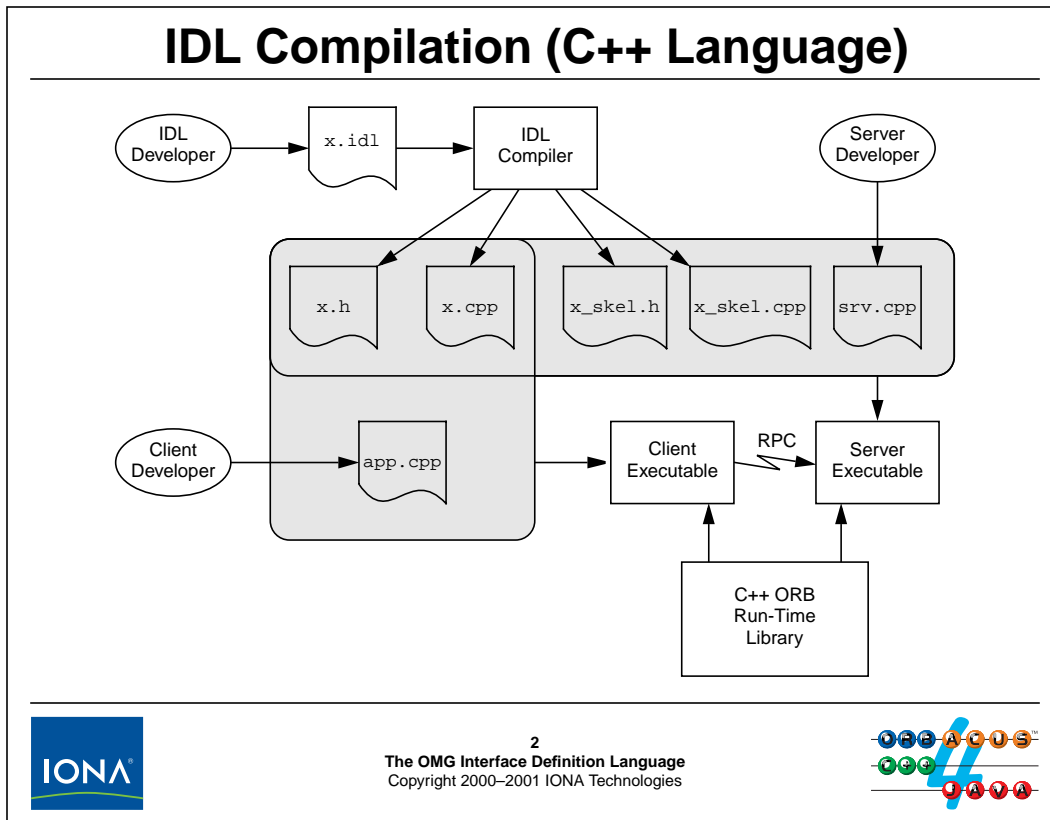
2.1 Introduction

The OMG Interface Definition Language (IDL) is CORBA's fundamental abstraction mechanism for separating object interfaces from their implementations. IDL establishes a contract between client and server that describes the types and object interfaces used by an application. This contract ensures that client and server agree on the types and interfaces used by an application.

IDL specifications are independent of the implementation language, so client and server can be written in different languages. An IDL compiler translates IDL specifications into APIs in a specific implementation language, such as C++. You use these generated APIs to interact with applications and the ORB. The translation algorithms from IDL into APIs for specific implementation languages are known as *language mappings* and defined by the OMG. Currently, CORBA defines language mappings for C, C++, Ada, COBOL, Smalltalk, Java, and Python, as well as a scripting language called CORBAscript, which is useful for rapid prototyping. Independent efforts are underway to provide language mappings for Eiffel, Modula 3, Perl, Lisp, Visual Basic, and a number of others; some of these language mappings may eventually become a standard.

IDL defines interfaces, not implementations. This means that IDL is a purely declarative language. You cannot say anything about object state in IDL and you cannot write executable statements. Instead, you use IDL to define types, interfaces, and operations (which permit the exchange of data between client and server). IDL is analogous to C++ header files, which define types, classes, and methods

Data can be exchanged only if it is defined in IDL. You cannot, for example, pass a C++ type directly to a client or server because doing so would destroy the language independence of CORBA. (For example, a Java server would not be able to use a C++ data type.)



2.2 IDL Compilation (C++)

An IDL compiler produces source files that must be combined with application code to produce client and server executables. (Note that the CORBA standard does not specify the name and number of files that should be produced; the above names are therefore specific to ORBacus.)

The above diagram shows the development steps for a client and server written in C++ and using the same ORB. Client and server developer must agree to use the same IDL source. The IDL source file (`x.idl`) is compiled by the IDL compiler into four files:

- `x.h`

This file contains C++ type definitions that correspond to the data types defined in `x.idl`. In addition, it contains stub class definitions that correspond to the interfaces defined in IDL. This header file is included in both client and server to ensure that they agree on the types that are used by the application.

- `x.cpp`

This file contains the source code for the types and classes declared in `x.h`. It is linked into both client and server executables.

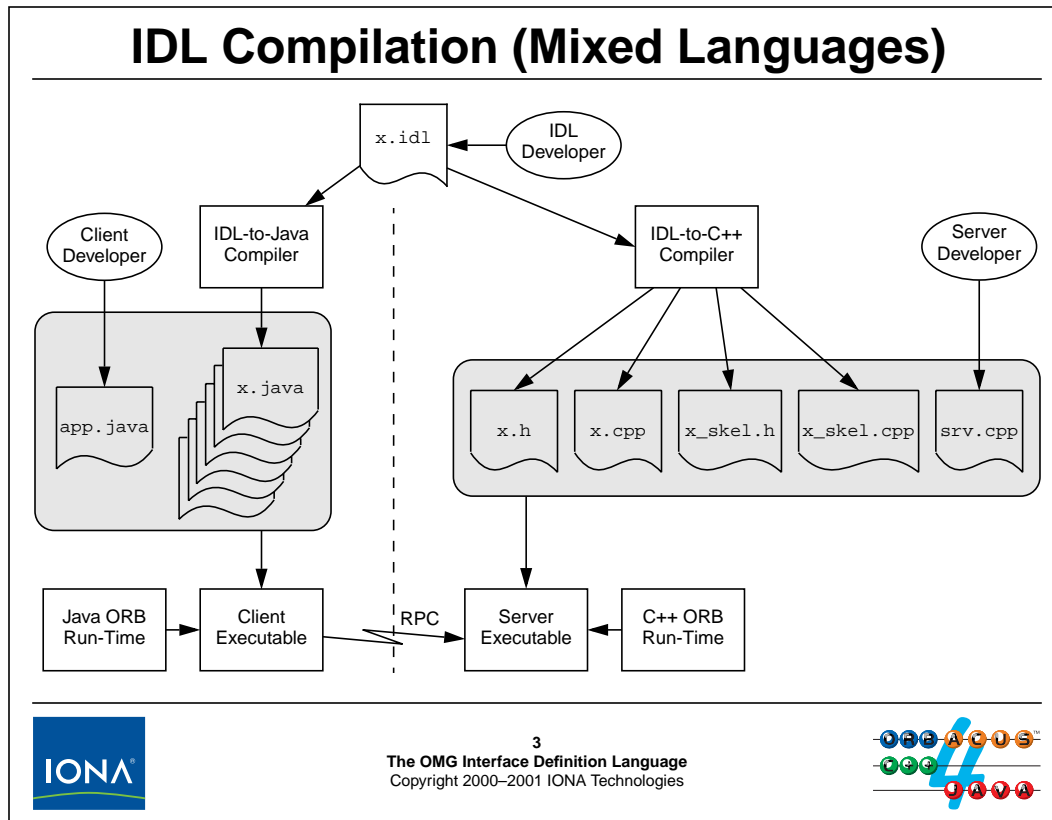
- `x_skel.h`

This file contains definitions that are specific to the server side, so it is included only in the server source code. (`x_skel.h` includes `x.h`, so it is sufficient to write a single `#include "x_skel.h"` statement in `srv.cpp` in order to include both `x_skel.h` and `x.h`.)

- `x_skel.cpp`

This file contains source code for the skeleton classes that provide the server-side up-call interface. It is linked into the server executable only.

For the client, the client application code (`app.cpp` in this example) is linked with the stubs into the client executable. For the server, the stubs, skeletons, and the server application code (`srv.cpp` in this example) is linked into server executable. Both client and server also are linked against a library that provides the necessary run-time support.



2.3 IDL Compilation (Mixed Languages)

If client and server use different languages, they cannot share source or binary components. Despite that, they can communicate with each other, provided they both use an interoperable ORB. The above diagram shows the development steps for a client written in Java and a server written in C++.

Again, the only thing that links client and server developer is the IDL definition for the application. Otherwise, client and server developer use completely separate development environments and language mappings, and they can use ORBs from different vendors.

For the server side, the same development steps apply as for a pure C++ environment. For the client side, the developer uses an IDL compiler that generates stubs and skeletons in Java instead of C++. (Of course, for a Java client, only the Java stubs are relevant and the Java skeletons are ignored.) The IDL-to-Java compiler produces a number of Java files that, together with the client application code (`app.java` in this example), form the client executable. As for a C++ ORB, a library provides the necessary run-time support for the client application.

IDL Source Files

The CORBA specification imposes a number of rules on IDL source files:

- IDL source files must end in a `.idl` extension.
- IDL is a free-form language. You can use white space freely to format your specification. Indentation is not lexically significant.
- IDL source files are preprocessed by the C++ preprocessor. You can use `#include`, macro definitions, etc.
- Definitions can appear in any order, but you must follow the “define before use” rule.



4
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.4 IDL Source Files

You must use a `.idl` extension for your IDL source files. For example, `CCS.idl` is a legal IDL file name. If you are working in an environment with case-insensitive file names, `CCS.IDL` is legal. However, in environments with case-sensitive file names, `CCS.IDL` is not a legal IDL file name.

Like C++, IDL permits free use of white space (spaces, tabs, horizontal and vertical tabs, form feeds, and newlines). All of these act as token separators (as does a C-style `/* . . . */` comment). Indentation does not carry semantics, so you can use any layout you prefer. (You may want to follow the layout and punctuation used here, which follows the OMG style guide for IDL.)

IDL source files are preprocessed exactly as C++ source files are. This means that you can use all of the C++ preprocessor features, such as `#include`, macro definitions, and so on.

As with C++, you can define types in any order that is convenient, with the proviso that you must define things before you can use them. (For recursive types, a forward declaration permits you to avoid violating this rule.)

Comments and Keywords

- IDL permits both C++-style and C-style comments:

```
/*  
 * A C-style comment  
*/
```

```
// A C++-style comment
```

- IDL keywords are in lower case (e.g. **interface**), except for the keywords **TRUE**, **FALSE**, **Object**, and **ValueBase**, which must be spelled as shown.



5
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.5 Comments and Keywords

IDL permits both C-style and C++-style comments.

IDL keywords must be spelled in lower case. For example, `interface`, `struct`, and `union` are valid keywords, whereas `Interface`, `STRUCT`, and `uNion` are not. There are four exceptions to this rule: the keywords `TRUE`, `FALSE`, `Object`, and `ValueBase` must be capitalized as shown.

Identifiers

- IDL identifiers can contain letters, digits, and underscores. For example:
Thermometer, nominal_temp
- IDL identifiers must start with a letter. A leading underscore is permitted but ignored. The following identifiers are treated as identical:
set_temp, _set_temp
- Identifiers are case-insensitive, so **max** and **MAX** are the same identifier, but you must use consistent capitalization. For example, once you have named a construct **max**, you must continue to refer to that construct as **max** (and not as **Max** or **MAX**).
- Try to avoid identifiers that are likely to be keywords in programming languages, such as **class** or **package**.



6
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.6 Identifiers

IDL identifiers can contain letters, digits, and underscores, and they must start with a letter. A single leading underscore is ignored, so `set_temp` and `_set_temp` are considered the same identifier. Identifiers with two or more leading underscores are illegal.

The rules for leading underscores exist to provide an escape hatch: as new features are added over time to IDL, new keywords must be introduced. The leading underscore rule permits existing specifications that clash with a new keyword to be retained by adding a leading underscore. For example, the IDL identifier `_ValueBase` is not treated as a keyword, but as an identifier. You should not use leading underscores in your specifications unless a newer version of the ORB introduces a keyword that clashes with one of your identifiers.

IDL identifiers are case-insensitive, so you cannot use, for example, `temp` and `Temp` in the same scope. However, once you have used an identifier to name a construct, you must use the same capitalization to name that same construct; otherwise, the IDL compiler flags an error. (These rules exist to permit mapping of IDL into languages that are case-sensitive as well as languages that are case-insensitive, without having to resort to name mangling.)

In order to avoid awkward mappings into the target language, you should try and avoid identifiers that are likely to be programming language keywords. For example, `class`, `package`, `while`, `import`, `PERFORM`, and `self` are poor choices.

Built-In Types

IDL provides a number of integer and floating-point types:

Type	Size	Range
short	≥ 16 bits	-2^{15} to $2^{15}-1$
unsigned short	≥ 16 bits	0 to $2^{16}-1$
long	≥ 32 bits	-2^{31} to $2^{31}-1$
unsigned long	≥ 32 bits	0 to $2^{32}-1$
long long	≥ 64 bits	-2^{63} to $2^{63}-1$
unsigned long long	≥ 64 bits	0 to $2^{64}-1$
float	≥ 32 bits	IEEE single precision
double	≥ 64 bits	IEEE double precision
long double	≥ 79 bits	IEEE extended precision

Types **long long**, **unsigned long long**, and **long double** may not be supported on all platforms.



7
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.7 Built-In Types

IDL has a number of built-in types, similar to C++ and other programming languages.

2.7.1 Numeric Types

IDL supports integer, floating-point, and fixed-point types.

Integer Types

IDL provides **short**, **long**, and **long long** integer types, both signed and unsigned. Note that the size guarantees shown here must be maintained by language mappings.

NOTE: 64-bit integers were added with CORBA 2.1 and do not interoperate with older ORBs.

Floating-Point Types

IDL provides single, double, and extended precision floating-point types. IEEE format is not supported in all environments; if that is the case, the ORB will provide an approximation to IEEE floating-point semantics.

NOTE: Be careful when using type **long double**: it may not be supported in all environments, depending on your CPU architecture and your compiler. In addition, type **long double** was added with CORBA 2.1 and does not interoperate with older ORBs. In addition, type **long double** is not supported by the Java mapping.

Built-In Types (cont.)

CORBA 2.1 added type **fixed** to IDL:

```
typedef fixed<9,2>  AssetValue;    // up to 9,999,999.99
                                     // accurate to 0.01
typedef fixed<9,4>  InterestRate; // up to 99,999.9999,
                                     // accurate to 0.0001
typedef fixed<31,0> BigInt;       // up to 10^31 - 1
```

Fixed-point types have up to 31 decimal digits.

Fixed-point types are not subject to the imprecision of floating-point types.

Calculations are carried out internally with 62-digit precision.

Fixed-point types are useful mainly for monetary calculations.

Fixed-point types are not supported by older ORBs.



8
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



Fixed-Point Types

Fixed-point types specify a total number of digits and a scale that sets the decimal point. For example, `fixed<5, 2>` specifies a fixed-point type with five digits, two of which are fractional, so it can represent values from -999.99 to 999.99 in increments of 0.01 .

Fixed-point types can represent decimal fractions exactly (to the precision of the number of fractional digits) and so are not plagued by the representational idiosyncrasies of floating-point types. (For example, the number 0.1 cannot be represented accurately as an IEEE floating-point value because IEEE floating-point format can represent fractions without error only if they are a fractional power of 2.) Internally, calculations on fixed-point types are carried out with a precision of 62 digits, making them especially useful to represent monetary values.

NOTE: Fixed-point types were added with CORBA 2.1 and do not interoperate with older ORBs, so be sure to use them only if you know that they are supported by all environments that are relevant to you.

Built-In Types (cont.)

IDL provides two character types, **char** and **wchar**.

- **char** is an 8-bit character, **wchar** is a wide (2- to 6-byte) character.
- The default codeset for **char** is ISO Latin-1 (a superset of ASCII), the codeset for **wchar** is 16-bit Unicode.

IDL provides two string types, **string** and **wstring**.

- A **string** can contain any character except NUL (the character with value zero). A **wstring** can contain any character except a character with all bits zero.
- Strings and wide strings can be unbounded or bounded:

```
typedef string      City;           // Unbounded
typedef string<3>  Abbreviation;  // Bounded
typedef wstring    Stadt;         // Unbounded
typedef wstring<3> Abkuerzung;    // Bounded
```



9
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.7.2 Character Types

The IDL **char** type can hold an 8-bit character. IDL does not mandate a particular codeset for characters, so you can use CORBA in, for example, an EBCDIC environment. If client and server use different codesets, the ORB takes care of appropriately translating characters during transmission.

The default codeset for **char** is ISO Latin-1, which is a superset of ASCII. (The bottom 128 character positions are identical to ASCII; the top 128 character positions are occupied by various European characters, such as ‘Å’.)

Wide characters permit support of scripts with large numbers of characters, such as Kanji. CORBA does not mandate a particular codeset for wide characters. Instead, client and server ORB transparently negotiate which codeset to use. This means that, for example, a Kanji client using Shift-JIS can transparently communicate with a Kanji server using Unicode (provided that either the client ORB or the server ORB can translate between Shift-JIS and Unicode).

NOTE: Wide characters were added with CORBA 2.1 and do not interoperate with older ORBs, so use them with caution.

2.7.3 String Types

An IDL `string` can contain any character except the NUL character.¹ Strings can be unbounded or bounded. An unbounded string can hold any number of characters (up to the memory limits of your platform). A bounded string contains at most the number of characters specified in its bound. The bound does *not* count any terminating NUL characters, so “Hello” *does* fit into a `string<5>`. (The notion of NUL-termination does not make sense in IDL because many programming languages do not represent strings as a NUL-terminated sequence of bytes.)

Wide strings also can be unbounded or bounded. As for wide characters, IDL does not mandate a particular codeset for wide strings; instead, a codeset that is common to both client and server is negotiated at run time. A wide string cannot contain a wide character whose value is zero (consist exclusively of zero bits). For bounded wide strings, the bound counts characters, not bytes.

1. This restriction is a concession to C and C++, in which it would be very difficult to deal with strings if they were allowed to contain embedded NUL characters.

Built-In Types (cont.)

- IDL type **octet** provides an 8-bit type that is guaranteed not to be tampered with in transit. (All other types are subject to translation, such as codeset translation or byte swapping.)

Type **octet** is useful for transmission of binary data.

- IDL type **boolean** provides a type with values **TRUE** and **FALSE**.
- IDL type **any** provides a universal container type.
 - A value of type **any** can hold a value of any type, such as **boolean**, **double**, or a user-defined type.
 - Values of type **any** are type safe: you cannot extract a value as the wrong type.
 - Type **any** provides introspection: given an **any** containing a value of unknown type, you can ask for the type of the contained value.



10
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.7.4 Other Built-In Types

IDL provides a few other built-in types, namely, **octet**, **boolean**, and **any**.

Type **octet**

Type **octet** is an uninterpreted 8-bit binary type. Values of type **octet** can store any pattern of eight bits and are guaranteed not to be tampered with in transit. This is important if you want to transmit binary data because all other IDL types are subject to translation in transit. (For example, **char** values may undergo translation from ASCII to EBCDIC, and **short** and **long** values may be byte-swapped.) IDL **octet** is the *only* type that is suitable for transmission of binary data. Do not be tempted to use some other type.

Type **boolean**

IDL provides the usual Boolean type. Its only values are **TRUE** and **FALSE**. (**TRUE** and **FALSE** are IDL keywords and must be capitalized as shown.)

Type **any**

Type **any** is a universal container type. A value of type **any** can hold a value of any other type, such as **long** or **string**, or even another value of type **any**. Type **any** can also hold user-defined complex types, such as structures or arrays.

Language mappings ensure that type **any** is type-safe: you cannot accidentally extract, for example, a value of type **long** and treat it as if it were a **double**. Such type mismatches are detected at run time. In addition, type **any** provides introspection capabilities. Given an **any** containing a value of unknown type, you can ask the **any** what type of value it contains.

Type Definitions

You can use **typedef** to create a new name for a type or to rename an existing type:

```
typedef short      YearType;
typedef short      TempType;
typedef TempType   TemperatureType;    // Bad style
```

You should give each application-specific type a name once and then use that type name consistently.

Judicious use of **typedef** can make your specification easier to understand and more self-documenting.

Avoid needless aliasing, such as **TempType** and **TemperatureType**. It is confusing and can cause problems in language mappings that use strict rules about type equivalence.



2.8 Type Definitions

As we saw on page 2-11, IDL provides a **typedef** keyword, which can use to rename (or alias) a type. The usual style considerations apply to **typedef**. The above definition of **YearType** is useful to the reader because it indicates that a value is a year, rather than some other, non-specific number. Similarly, **TempType** is also useful; it indicates that some other value indicates a temperature. As far as the application is concerned, years and temperatures are entirely different things and you should not pass one where the other is expected. The fact that they are both represented by the same underlying type is coincidental and effectively abstracted away by this style of specification.

Conversely, the above definition of **Temperature** type is simply bad style because it creates a needless alias.

Be careful about the semantics of **typedef**. It depends on the language mapping whether an IDL **typedef** results in a new, separate type or only an alias. In C++, **YearType** and **TempType** are compatible types that can be used interchangeably. However, IDL provides no guarantee that this must be true for all language mappings. For a mapping to a more strictly-typed language, such as Pascal, **YearType** and **TempType** could conceivably be mapped to incompatible Pascal types. To avoid problems down the road, define each logical type exactly once and then use that definition consistently throughout your specification.

Enumerations

You can define enumerated types in IDL:

```
enum Color { red, green, blue, black, mauve, orange };
```

- The type **Color** becomes a named type in its own right. (You do not need a **typedef** to name it.)
- A type name (such as **Color**) is mandatory. (There are no anonymous enumerated types.)
- The enumerators enter the enclosing naming scope and must be unique in that scope:

```
enum InteriorColor { white, beige, grey };
enum ExteriorColor { yellow, beige, green }; // Error!
```

- You cannot control the ordinal values of enumerators:

```
enum Wrong { red = 0, blue = 8 }; // Illegal!
```



12
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.9 Enumerations

IDL enumerations look much like their C++ counterpart. However, you cannot control the ordinal values of enumerators.² IDL guarantees that the ordinal values for enumerators increase left to right, so in the above **Color** example, **red** is guaranteed to compare less than **green**. However, the origin of ordinal values is not defined by IDL, and ordinal values may not even be contiguous. (Each language mapping establishes its own rules for how to assign ordinal values to enumerators.)

In practice, you do not care about ordinal values because the ORB ensures that they are translated correctly. For example, if you send the value **red** from a C++ client to a server written in an unknown language, it is guaranteed that the value **red** will be delivered to the server, even though it may have a different ordinal value in the server's programming language. Do not be tempted to send ordinal values as enumerators. Doing so has undefined behavior (and fortunately causes a compile-time error for most language mappings).

Enumerations cannot be empty.

Do not use **typedef** with enumerations because it results in needless aliasing:

```
typedef enum Direction { up, down } DirectionType; // Bad style!
```

2. This would be difficult to map to languages without direct support for this feature.

Structures

You can define structures containing one or more members of arbitrary type (including user-defined complex types):

```
struct TimeOfDay {
    short  hour;    // 0 - 23
    short  minute; // 0 - 59
    short  second; // 0 - 59
};
```

- A structure must have at least one member.
- The structure name is mandatory. (There are no anonymous structures.)
- Member names must be unique with the structure.
- Structures form naming scopes.
- Avoid use of **typedef** for structures.



13
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.10 Structures

IDL supports structures, that is, sets of named members of arbitrary type. The above definition creates a new type (`TimeOfDay`) which becomes a new type in its own right. As for enumerations, do not use `typedef` with structures because it needlessly aliases the type:

```
typedef struct TimeOfDay {
    short  hour;    // 0 - 23
    short  minute; // 0 - 59
    short  second; // 0 - 59
} DayTime;           // Bad style!
```

Structures form naming scopes, so the following is legal (if ugly) IDL:

```
struct Outer {
    struct FirstNested {
        long  first;
        long  second;
    } first;

    struct SecondNested {
        long  first;
        long  second;
    } second;
};
```

This example demonstrates that the various `first` and `second` identifiers do not cause a name collision. However, such in-line definition of types is hard to read, so the preceding is better expressed as follows:

```
struct FirstNested {
    long    first;
    long    second;
};

struct SecondNested {
    long    first;
    long    second;
};

struct Outer {
    FirstNested    first;
    SecondNested   second;
};
```

Note that this definition is more readable but is *not* exactly the same as the preceding definition. The nested version only adds the single type name `Outer` to the global scope, whereas the non-nested version also adds `FirstNested` and `SecondNested`.

Of course, the second version must still be considered bad style because it ruthlessly reuses the identifiers `first` and `second` for different purposes. Even though such reuse is legal, in the interest of clarity, you should avoid it.

Unions

IDL supports discriminated unions with arbitrary member type:

```
union ColorCount switch (Color) {
case red:
case green:
case blue:
    unsigned long    num_in_stock;
case black:
    float            discount;
default:
    string           order_details;
};
```

- A union must have at least one member.
- The type name is mandatory. (There are no anonymous unions.)
- Unions form naming scopes with unique member names.



14
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.11 Unions

IDL offers discriminated unions. Unlike C or C++ unions, IDL unions are discriminated. At most one member of a union is active at a time, depending on the discriminator value. In the above example, `num_in_stock` is active if the discriminator value is `red`, `green`, or `blue`.

The discriminator type must be an integral type, that is, `char`, an integer type, `boolean`, or an enumeration type. You cannot use `octet` or `wchar` as a discriminator.

NOTE: Even though it is legal, you should avoid defining unions that use `char` as the discriminator type. This avoids problems if client and server use different codesets.

Union members can be of any type, including user-defined complex types. As in C++, unions establish a naming scope, so member names need be unique only within their enclosing union.

As for enumerations and structures, you should avoid using `typedef` with unions because it creates needless aliases:

```
typedef union DateTime switch (boolean) {
case FALSE:
    Date    d;
case TRUE:
    Time    t;
} DateTime; // Bad style!
```


The default case for a union is optional. If it is present, there must be at least one discriminator value that is not used by explicit case labels. For example, the following is illegal:

```
union BadUnion switch (boolean) {
  case FALSE:
    string member_1;
  case TRUE:
    float member_2;
  default:
    octet member_3;      // Error!
};
```

One particular use of unions has become idiomatic and deserves special mention:

```
union AgeOpt switch (boolean) {
  case TRUE:
    unsigned short age;
};
```

Unions such as this one are used to implement optional values. A value of type `AgeOpt` contains an age only if the discriminator is `TRUE`. If the discriminator is `FALSE`, the union is empty and contains no value other than the discriminator itself.

IDL does not support optional or defaulted operation parameters, so the preceding union construct is frequently used to simulate that functionality. This is particularly useful if no special sentinel (“dummy”) value is available to indicate the “this value is absent” condition for a parameter.

Guidelines for Unions

A few guidelines to make life with unions easier:

- Do not use **char** as a discriminator type.
- Do not use unions to simulate type casting.
- Avoid using multiple **case** labels for a single union member.
- Avoid using the **default** case.
- Use unions sparingly. Often, they are abused to create operations that are like a Swiss army knife.



15
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.12 Guidelines for Unions

As mentioned previously, avoid using `char` as a discriminator type for unions because it creates problems if the target environment does not use a codeset that can accurately represent the discriminator value. For example, the following union causes problems if the client uses ISO Latin-1 and the server uses EBCDIC:

```
union U switch (char) {
  case '~':
    long    long_member;
  //...
};
```

The problem here is that EBCDIC does not have a ‘~’ character, so the union discriminator cannot be represented in EBCDIC. The client ORB could send the ASCII code for ‘~’ (0x7e). However, doing so would result in the server ORB receiving the EBCDIC character ‘”’ (which also has code 0x7e). Conversely, the client ORB could attempt to translate ‘~’ into a roughly equivalent character, such as EBCDIC ‘¬’. However, that is no longer the same character as ‘~’, so either approach (sending codes or attempting character translation) has problems.

Do not use unions as a back-door mechanism for type casting. Depending on the language mapping, this either does not work at all or it causes undefined behavior.

Avoid using multiple `case` labels for the same union member; in addition, you should avoid using the `default` label. Unions using these features are legal and can be used without portability problems, but make life harder for programmers because such unions are not as easy to use as unions that avoid the `default` label and restrict themselves to a single `case` label per member.

You should exercise caution before deciding to use unions in your IDL. Sometimes, they are appropriate; however, quite often, unions end up being abused to build operations that are like Swiss army knives, for example:

```
enum InfoKind { text, numeric, none };

union Info switch (InfoKind) {
  case text:
    string  description;
  case numeric:
    long    index;
};

interface Order {
  void set_details(in Info details);
};
```

The operation `set_details` (see page 2-37 for IDL operations) can do triple duty in this specification and accept a `string`, a `long`, or (conceptually) no parameter at all. This is not only confusing but also makes it harder for programmers to use the API because they must correctly initialize and pass a union value, something that is more complex and error-prone than passing a simple value. The following does the same job and is easier to understand:

```
interface Order {
  void set_text_details(in string details);
  void set_details_index(in long index);
  void clear_details();
};
```

Arrays

IDL supports single- and multi-dimensional arrays of any element type:

```
typedef Color   ColorVector[10];
typedef string  IdTable[10][20];
```

You must use a **typedef** to define array types. The following is illegal:

```
Color ColorVector[10]; // Syntax error!
```

You must specify all array dimensions. Open arrays are not supported:

```
typedef string OpenTable[][20]; // Syntax error!
```

Be careful when passing array *indexes* between address spaces.



16
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.13 Arrays

IDL supports both single- and multi-dimensional arrays of arbitrary element type. As in C++, the array bounds must be positive and non-zero constant integer expressions. You must specify all array dimensions. IDL does not support open arrays because IDL does not support pointer types. (In C and C++, open arrays are pointers in disguise.)

NOTE: Even though you are allowed to specify multi-dimensional arrays as a single definition, it is preferable to use a separate **typedef** for each dimension. The above definition for `IdTable` is better written as:

```
typedef string    IdVector[20];
typedef IdVector  IdTable[10];
```

Defining a separate row type avoids problems in some language bindings with anonymous types: a direct definition using a single **typedef** causes problems if you want to declare or pass a variable that represents a row of the array.

Passing an array index between clients and servers is dangerous. For example, if you were to pass the number 2 to indicate the third element of an array, you may get surprises. That is because not all programming languages use zero as the origin for array indexes. (For example, a Pascal implementation may choose 1 as the origin for array indexes.) If you must pass array (or sequence) indexes between clients and servers, make a convention that array indexes always start at zero; for those language bindings where array indexes have a different origin, explicitly add the origin to the one that is passed. That way, you remain portable across languages.

Sequences

Sequences are variable-length vectors of elements of the same type.

Sequences can be unbounded (grow to any length) or bounded (limited to a maximum number of elements):

```
typedef sequence<Color>      Colors;
typedef sequence<long, 100> Numbers;
```

The sequence bound must be a non-zero, positive integer constant expression.

You must use a **typedef** to define sequence types.

The element type can be any other type, including a sequence type:

```
typedef sequence<Node>      ListOfNodes;
typedef sequence<ListOfNodes> TreeOfNodes;
```

Sequences can be empty.



17
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.14 Sequences

As opposed to IDL arrays, which have a fixed number of elements at all times, IDL sequences are variable-length vectors. An unbounded sequence can contain any number of elements from zero up to the memory limits of your environment. A bounded sequence can contain any number of elements from zero up to the bound.

A sequence can contain any type of elements, including another sequence. In the above example, this is used to model a tree as sequence of sequences.

NOTE: You can define a sequence of sequences in a single definition:

```
typedef sequence< sequence<Node> > TreeOfNodes; // Deprecated!
```

Note the space between the two closing “>” tokens. The space is necessary because, otherwise, “>>” would be as a single right-shift operator. Even though the preceding definition is legal, it has been deprecated to avoid problems with anonymous types (see page 2-60). The construct will become illegal in a future version of CORBA, so you should avoid it.

Sequences or Arrays?

Sequences and arrays are similar, so here are a few rules of thumb for when to use which:

- If you have a list of things with fixed number of elements, all of which exist at all times, use an array.
- If you require a collection of a varying number of things, use a sequence.
- Use arrays of **char** to model fixed-length strings.
- Use sequences to implement sparse arrays.
- You must use sequences to implement recursive data structures.



18
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.15 Sequences or Arrays?

Arrays always contain a fixed number of elements, whereas sequences can vary in length at run time. In addition, for arrays, all elements must be transmitted between client and server, whereas for sequences, only those elements that are actually present need be transmitted. It follows that arrays are appropriate only if you have a collection with a fixed number of elements, all of which must exist at all times.

To model a fixed-length string (as opposed to a bounded string), an array of characters is more appropriate than a bounded string or a bounded sequence. For example, a ZIP code, which always has five digits, is best modeled as:

```
typedef char ZIPCode[5];
```

This definition is more appropriate than either a bounded string or a bounded sequence because it enforces that a ZIP code must have exactly five digits and not any number up to five.

Sequences are also useful to model sparse arrays. (A sparse array is an array in which most elements have a default value, such as zero. Sparse arrays are common in graphics processing.) If we need to, for example, transmit sparse 2-D matrices of numbers, we can naively model a matrix inversion interface as follows:

```

typedef float   RowType[100];
typedef RowType SquareMatrix[100];

interface MatrixProcessor {
    SquareMatrix invert(in SquareMatrix m);
    // ...
};

```

Here we send a matrix of 10,000 values from the client to the server. The server inverts the matrix and returns another matrix of 10,000 values. This requires transmission of 80,000 bytes of data (4 bytes for each `float` in each direction). If the matrices we use contain a large number of zero values, that is, are sparse, we can save considerable bandwidth by using sequences to model them:

```

struct CellType {
    float      value;
    unsigned long col_num;
};
typedef sequence<CellType, 100> RowType;

struct RowInfo{
    RowType      row_vals;
    unsigned long row_num;
};
typedef sequence<RowInfo, 100> SquareMatrix;

```

The idea here is to only send those values in the matrix that are non-zero. A `CellType` structure models the value of a particular cell by recording the value of the cell and the column in which that cell appears. A sequence of such cells (`RowType`) then models all the non-zero cells in a particular row. Similarly, a `RowInfo` structure records one row containing at least one non-zero point, together with an index that indicates which particular row is represented by the structure. A sequence of such `RowInfo` structures then models the 2-D matrix. For example, if we were to send a matrix containing a single non-zero value in the third row and the tenth column, we would send a single `RowInfo` structure with `row_num` set to 2 and containing a one-element sequence in its `row_vals` member; that sequence would contain the value 9 in the `col_num` member and the value of the cell in the `value` member.

With this approach, if we assume that, on average 75% of cells are empty (which is not uncommon), we will transmit a little over 40,000 bytes in total, instead of the 80,000 bytes we sent using the earlier approach.

NOTE: The savings here are small because each cell only contains a single `float` value and the column index in each cell doubles the size of the cell. If the payload in each cell is larger, such as a 3-D point containing a `double` for each coordinate, the savings are quite spectacular. However, the technique does not work for matrices that are not sparse; to get any saving, the sparseness of the array must outweigh the increase in size of each cell.

Recursive Types

IDL does not have pointers, but still supports recursive data types:

```
struct Node {
    long      value;
    sequence<Node> children;
};
```

- Recursion is possible only for structures and unions.
- Recursion can be achieved only via a sequence member. The element type of the sequence must be an enclosing structure or union.
- Recursion can span more than one enclosing level.
- Mutual recursion is not supported by IDL.



19
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.16 Recursive Types

IDL supports recursive types without the need for a pointer type by using sequence members in structures or unions. The sequence element type must be the name of a structure or union currently under definition.

2.16.1 Recursion Via Structures

The above example shows a data structure consisting of nodes, in which each node contains a `long` value and a number of descendant nodes. This approach can be used to model both trees and lists. Leaf nodes, which do not have descendants, contain an `children` sequence that is empty.

2.16.2 Recursion Via Unions

Recursion via unions is possible as well. Here is an example that defines a simple expression tree:

```
enum OpType {
    OP_AND, OP_OR, OP_NOT,
    OP_BITAND, OP_BITOR, OP_BITXOR, OP_BITNOT
};

enum NodeKind { LEAF_NODE, UNARY_NODE, BINARY_NODE };

union Node switch (NodeKind) {
case LEAF_NODE:
    long      value;
case UNARY_NODE:
    struct UnaryOp {
```



```

        OpType      op;
        sequence<Node, 1> child;
    } u_op;
case BINARY_NODE:
    struct BinaryOp {
        OpType      op;
        sequence<Node, 2> children;
    } bin_op;
};

```

Note that in this example, the incomplete type for the recursion is a `union` (instead of a `struct`) and that *bounded* sequences are used. Using a bounded sequence improves the type safety of the specification. For example, it enforces that a binary node cannot have more than two descendants. However, we cannot enforce that a binary node must have *exactly* two descendants. Attempts to achieve this using arrays are illegal:

```

// ...
case BINARY_NODE:
    struct BinaryOp {
        OpType op;
        Node   children[2];    // Illegal!
    } bin_op;
// ...

```

2.16.3 Multilevel Recursion

Recursion can extend over more than one level. Here is an example that shows the recursion on the incomplete type `TwoLevelRecursive` nested inside another structure definition:

```

struct TwoLevelRecursive {
    string id;
    struct Nested {
        long value;
        sequence<TwoLevelRecursive> children;    // OK
    } data;
};

```

2.16.4 Mutual Recursion

Mutual recursion is not supported. (It is impossible to define mutually recursive structures or unions because the element type of the recursive member must be a type currently under definition.) It is possible to approximate mutual recursion by creating a union containing recursive structure members. However, the approach is messy and loses some type safety, so it is best to avoid mutually recursive data structures entirely.

Constants and Literals

You can define a constant of any built-in type (except **any**) or of an enumerated type:

```
const long      FAMOUS_CONST = 42;
const double    Sqrt_2 = 1.1414213;
const char      FIRST = 'a';
const string    GREETING = "Gooday, mate!";
const octet     LSB_MASK = 0x01;
```

```
typedef fixed<6,4> ExchangeRate;
const ExchangeRate UNITY = 1.0D;
```

```
enum Color { ultra_violent, burned_hombre, infra_dead };
const Color NICEST_COLOR = ultra_violent;
```

Constants must be initialized by a literal or a constant expression.



20
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.17 Constants and Literals

IDL allows constant definitions for all built-in types (except type **any**) and for enumerated types. Constants must be initialized with a literal or a constant expression (see page 2-33). The syntax is borrowed from C++, so you can use the familiar C++ escape sequences and radixes.

2.17.1 Integer Constants

IDL permits initialization of integer constants with decimal, octal, or hexadecimal integer literals. Unary plus and minus are permitted:

```
const unsigned short  A = 1;
const long             B = -0234; // Octal 234, decimal 156
const long long       C = +0x234; // Hexadecimal 234, decimal 564
```

2.17.2 Floating-Point Constants

For floating-point constants, IDL uses the same syntax as C++. Here are a few examples:

```
const double  A = 3.7e-12; // integer, fraction, & exponent
const float   B = -2.71;   // integer part and fraction part
const double  C = .88;     // fraction part only
const long double D = 12.; // integer part only
const double  E = .3E8;    // fraction part and exponent
const double  F = 2E11;    // integer part and exponent
```

2.17.3 Fixed-Point Constants

Fixed-point constants do not have an explicit number of digits and scale. The number of digits and the scale are instead inferred from the literal or expression that is used to initialize the constant. (Leading and trailing zeros in literals are ignored.) Fixed-point literals must end in a `d` or `D`. Here are a few examples:

```
const fixed f1 = 99D;           // fixed<2,0>
const fixed f2 = -02.71d;      // fixed<3,2>
const fixed f3 = +009270.00D;  // fixed <4,0>
const fixed f4 = 00.009D;      // fixed <4,3>
```

2.17.4 Character and Wide Character Constants

IDL character constants support the same escape sequences as C++. For example:

```
const char c1 = 'c';           // the character c
const char c2 = '\007';        // ASCII BEL, octal escape
const char c3 = '\x41';        // ASCII A, hex escape
const char c4 = '\n';          // newline
const char c5 = '\t';          // tab
const char c6 = '\v';          // vertical tab
const char c7 = '\b';          // backspace
const char c8 = '\r';          // carriage return
const char c9 = '\f';          // form feed
const char c10 = '\a';         // alert
const char c11 = '\\';         // backslash
const char c12 = '\?';         // question mark
const char c13 = '\'';         // single quote
```

Wide character literals use the prefix `L`. You can use Unicode universal character names to enter characters from codesets other than ISO Latin-1:

```
const wchar X = L'X';          // 'X' as a wide character
const wchar OMEGA = L'\u03a9'; // Unicode universal character name
```

2.17.5 String and Wide String Constants

All of the escape sequences that are legal for characters also work for strings. In addition, the escape sequence `\"` escapes a double quote. As with C++, lines ending in `\` and adjacent string literals are concatenated. (Both these are functions of the preprocessor.)

```
const string S1 = "Quote: \"; // string with double quote
const string S2 = "hello world"; // simple string
const string S3 = "hello" " world"; // concatenate
const string S4 = "\xA" "B"; // two characters \
                               // ('\xA' and 'B'), \
                               // not the single \
                               // character '\xAB'

const string<5> BS = "Hello"; // Bounded string constant
```

Wide string literals use the prefix `L`. As for character literals, you can use Unicode universal character names for characters not from the ISO Latin-1 codeset.

```
const wstring LAST_WORDS = L"My God, it's full of stars!";
const wstring<8> O = L"Omega: \u3A9";
```

2.17.6 Boolean Constants

Boolean constants are supported. However, the only things you can do with them are nonsensical ones, so it is probably best to simply use TRUE and FALSE.

```
const boolean CONTRADICTION = FALSE;    // Bad idea...
const boolean TAUTOLOGY     = TRUE;     // Just as bad...
```

Both these definitions are bad style because they create needless aliases.

2.17.7 Octet Constants

Octet constants must be initialized with a non-negative integer literal or expression. Initialization with a value outside the range 0–255 is illegal.

```
const octet 01 = 0;
const octet 02 = 0xff;
```

NOTE: Octet constants were added in CORBA 2.3, so IDL files containing octet constants cannot be translated with older IDL compilers.

2.17.8 Enumeration Constants

You must initialize an enumeration constant with an enumerator that is a member of the type of the constant. Both scoped and unqualified names are legal.

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

const FavoriteColor = green;
const OtherColor    = ::blue;
```

NOTE: Enumeration constants were added in CORBA 2.3, so IDL files containing enumeration constants cannot be translated with older IDL compilers.

Constant Expressions

IDL defines the usual arithmetic and bitwise operators for constant expressions:

Operator Type	IDL Operators
Arithmetic	+ - * / %
Bitwise	& ^ << >> ~

The bitwise operators require integral operands. (IDL guarantees two's complement representation for integral types.)

The operators do *not* have exactly the same semantics as in C++:

- Arithmetic operators do not support mixed-mode arithmetic.
- The >> operator *always* performs a logical shift.



21
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.18 Constant Expressions

The arithmetic operators are occasionally useful to initialize numeric constants with an expression:

```
const short MIN_TEMP = -10;
const short MAX_TEMP = 35;
const short DFLT_TEMP = (MAX_TEMP + MIN_TEMP) / 2;
```

```
const float TWO_PIES = 3.14 * 2.0; // Cannot use 3.14 * 2 here!
```

```
const fixed YEARLY_RATE = 0.1875D;
const fixed MONTHLY_RATE = YEARLY_RATE / 12D; // Cannot use 12 here!
```

The bitwise operators are rarely used (if ever). Note that IDL specifies that the >> operator always performs a logical shift operation (injects zeros on the left), whereas in C++, the behavior is implementation-defined:

```
const long ALL_ONES = -1; // 0xffffffff
const long LHW_MASK = ALL_ONES << 16; // 0xffff0000
const long RHW_MASK = ALL_ONES >> 16; // 0x0000ffff, guaranteed
```

NOTE: The behavior of constant expressions with respect to overflow is under-specified and therefore not portable. You should ensure that constant expressions do not overflow their target type.

Interfaces

Interfaces, like C++ class definitions, define object types:

```
interface Thermometer {
    string get_location();
    void set_location(in string loc);
};
```

- Invoking an operation on an instance of an interface sends an RPC call to the server that implements the instance.
- Interfaces define a *public* interface. There is no private or protected section for interfaces.
- Interfaces do not have members. Members store state, but state is an implementation (not interface) concern.
- Interfaces define the smallest and only granularity of distribution: for something to be accessible remotely, it must have an interface.



22
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.19 Interfaces

IDL interfaces define object types, much like C++ class definitions. The main difference is that interface instances can be remote, whereas C++ class instances cannot.

Interfaces mainly define operations (or attributes—see page 2-46). Invoking an operation via an object reference on an interface instance (that is, a CORBA object) sends a message to the (usually remote) object. The ORB takes care of locating the object, transmitting the message and its arguments to the destination, and returning the results (if any) to the client.

By definition, everything defined in an interface is public. The notion of a private or protected part does not apply because they are implementation (not interface) concepts. Things are made private simply by not saying anything about them.

Similarly, interfaces do not have member variables because member variables are concerned with state (that is, implementation). IDL attributes (see page 2-46) are not member variables (even though they look somewhat like member variables). Of course, you can *implement* an interface such that it makes use of member variables; it is simply that such member variables are not a visible part of an object's interface.

You can implement instances of an interface in a single server process, or you can implement them in more than one process. Each interface instance represents a CORBA object. An object reference denotes exactly one CORBA object. Each CORBA object has exactly one (most derived) interface.

Interfaces define the smallest grain of distribution in CORBA. For something to be remotely accessible, it must have an IDL interface.

Interface Syntax

You can nest exception, constant, attribute, operation, and type definitions in the scope of an interface definition:

```
interface Haystack {
    exception NotFound { unsigned long num_straws_searched; };

    const unsigned long MAX_SIZE = 1000000;

    readonly attribute unsigned long num_straws;

    typedef long    Needle;
    typedef string  Straw;

    void    add(in Straw s);
    boolean remove(in Straw s);
    boolean find(in Needle n) raises(NotFound);
};
```



23
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.20 Interface Syntax

The above interface illustrates the kinds of definitions that may occur inside an interface definition:

- exception definitions
- constant definitions
- attribute definitions
- type definitions
- operation definitions

We will discuss the various constructs in detail over the next few pages.

Interfaces form naming scopes, so the usual rule applies: each name used within a scope must be unique and must not change meaning throughout the scope (see page 2-59).

IDL uses scope resolution rules that are derived from C++. In order to resolve a name, the compiler first searches the current interface, then searches base interfaces (if any) of the current interface, then searches the scopes enclosing the derived interface toward the global scope.³

You can use the `::` scope resolution operator to explicitly qualify a name. For example, `::Haystack::NotFound` denotes the exception defined in the above interface, regardless of the context in which the qualified name is used.

3. The precise scope resolution rules are quite involved because they need to deal with pathological cases (see page 2-56). However, if you use sensible (that is, unique) names in your specification, you will never have to learn the intricacies of scope resolution.

Interface Semantics

Interfaces are types and can be used as parameter types (or as a member for data structures). For example:

```
interface FeedShed {
    void    add(in Haystack s);
    void    eat(in Haystack s);
};
```

- The parameters of type **Haystack** are object reference parameters.
- Passing an object always passes it by reference.
- The object stays where it is, and the reference is passed by value.
- Invocations on the reference send an RPC call to the server.
- CORBA defines a dedicated nil reference, which indicates no object (points nowhere).



24
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.21 Interface Semantics

The above `FeedShed` interface has operations with parameters of type `Haystack`. For example, we can pass an object of type `Haystack` to the `FeedShed::add` operation. Conceptually, passing an object passes the object itself. However, what is really passed is an object reference. The actual `Haystack` object stays where it is (namely, in the server in which it is implemented) and what is passed to the `add` operation is an object reference to that hay stack. This is similar to passing a C++ pointer to a function. However, C++ pointers can only denote C++ objects in the same address space, whereas CORBA object references can denote objects anywhere, and implemented in any language.

The receiver of an object reference (the `add` or `eat` operation in this example) can use the reference to invoke operations on the particular hay stack that was passed (such as `add`, `remove`, or `find`). Invoking an operation on a reference sends an RPC call to the object.

CORBA defines a special nil value for object references. An object reference that is nil points nowhere. Obviously, you cannot invoke operations on a nil reference because it does not indicate an object that could respond to the invocation. CORBA defines an operation that permits you to test whether a particular reference is nil (see page 6-13).

Operation Syntax

Every operation definition has:

- an operation name
- a return type (**void** if none)
- zero or more parameter definitions

Optionally, an operation definition may have:

- a **raises** expression
- a **oneway** operation attribute
- a **context** clause

You cannot overload operations because operation names must be unique within the enclosing interface.



25
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.22 Operation Syntax

Every operation must have a name, a return type, and a (possibly empty) list of parameters. Here is an interface showing the simplest possible operation:

```
interface Simple {
    void op();
};
```

This operation does not send any values to the server and does not return any values from the server. (As a result, the only possible reason for invoking such an operation is to change the state of the target object as a side effect.) An operation definition must have a return type. If an operation returns no value, its return type is `void`. The return type does not have a default:

```
interface Simple {
    op(); // Error, missing return type
};
```

We discuss the optional parts of operation definitions in Sections 2.28 and 2.29.

You cannot overload operations because each operation must have a unique name within its enclosing interface.⁴

4. Operation overloading is not supported because it is very messy to map into languages without built-in support for overloading. For such languages, overloaded operations would have to be mapped using a mangled name (which is fine for compilers but not for humans).

Operation Example

Here is an interface that illustrates operations with parameters:

```
interface NumberCruncher {
    double square_root(in double operand);
    void    square_root2(in double operand, out double result);
    void    square_root3(inout double op_res);
};
```

- Parameters are qualified with a directional attribute: **in**, **inout**, or **out**.
 - **in**: The parameter is sent from the client to the server.
 - **out**: The parameter is returned from the server to the client.
 - **inout**: The parameter is sent from the client to the server, possibly modified by the server, and returned to the client (overwriting the initial value).



26
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.23 Operation Example

The above interface illustrates operations with parameters. IDL requires use of the **in**, **out**, or **inout** directional attribute for each parameter. The directional attribute determines whether a parameter is sent from client to server, server to client, or in both directions. Directional attributes are necessary mainly in order to preserve bandwidth: without a directional attribute, the ORB would have no way of knowing the origin and destination of a parameter, with the result that it would have to send all parameters in both directions. The directional attributes ensure that only **inout** parameters are sent in both directions, conserving bandwidth.⁵

Note that the above example used the (rather awkward) operation names `square_root`, `square_root2`, and `square_root3` because IDL does not permit overloading of operations. The following is illegal:

```
interface NumberCruncher {
    double square_root(in double operand);
    void    square_root(in double operand, out double result); // Error
    void    square_root(inout double op_res);                  // Error
};
```

5. As we will see in Section 6.27, directional attributes also influence the mapping to the target language.

Note that all three operations in this example achieve the same thing: they compute the square root of a number. However, each operation uses a different style:

- `square_root` accepts a number as an `in` parameter and returns the result as the return value.
- `square_root2` accepts a number as an `in` parameter and returns the result as an `out` parameter.
- `square_root3` accepts a number as an `in` parameter and overwrites it with the result.

Naturally, you would never define an interface like `NumberCruncher`, which offers three operations that all do the same thing. Instead, you would decide which style of interaction you wanted to offer to clients. The question is, which style is best, and how do you choose it? Here are some guidelines:

- If an operation accepts one or more `in` parameters and returns a single result, the result should be returned as the return value.
- If an operation has several return values of equal importance, all values should be returned as `out` parameters, and the return type of the operation should be `void`.

By making all return values `out` parameters, you emphasize that none of them is “special” (whereas if one value is returned as the return value and the others are `out` parameters, you can easily create the impression that the return value is somehow more important).

- If an operation returns several values but one of the values is of special importance, make the special value the return value and return the remainder as `out` parameters.

This style is most often found on iterator operations. For example:

```
boolean get_next(out SomeType next_value);
```

This style allows the caller to write code along the following lines:

```
while (get_next(value)) {  
    // Process value  
}
```

- Treat `inout` parameters with caution.

By using an `inout` parameter, the designer of the interface assumes that the caller will never want to keep the original value and that it is OK to overwrite it. Therefore, `inout` parameters dictate interface policy. If the client wants to keep the original value, it must make a copy first, and that can be inconvenient.

User Exceptions

A **raises** clause indicates the exceptions that an operation may raise:

```
exception Failed {};
exception RangeError {
    unsigned long    min_val;
    unsigned long    max_val;
};

interface Unreliable {
    void can_fail() raises(Failed);
    void can_also_fail(in long l) raises(Failed, RangeError);
};
```

- Exceptions are like structures but are allowed to have no members.
- Exceptions cannot be nested or be part of inheritance hierarchies.
- Exceptions cannot be members of other data types.



27
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.24 User Exceptions

IDL user exceptions provide a standard way of handling errors. User exceptions are similar to structures in that exceptions form naming scopes and can contain members of arbitrary type. However, unlike structures, exceptions are permitted to be empty (have no members). In addition, exceptions are not first-class data types: you cannot, for example, use an exception type as a structure member or as the member of another exception:

```
exception E1 {};
exception E2 {
    long    value;
    E1      exc;    // Illegal!
};

struct S {
    E1      exc;    // Illegal!
};
```

IDL does not provide exception inheritance, so each user exception defines a new type that is unrelated to any other exception type.

An operation can only raise those user exceptions that appear in its **raises** clause.

Use IDL exceptions for all your error handling. Exceptions are integrated into the exception handling mechanism of the target language and provide a well-understood and uniform error handling mechanism. In particular, do not use error codes; doing so results in awkward and difficult-to-use interfaces.

Using Exceptions Effectively

A few rules of thumb for how to use exceptions:

- Use exceptions only for *exceptional* circumstances.
- Make sure that exceptions carry *useful* information.
- Make sure that exceptions carry *precise* information.
- Make sure that exceptions carry *complete* information.

Sticking to these rules make the resulting APIs easier to use and understand and results in better quality code.



28
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.25 Using Exceptions Effectively

You can use the above rules of thumb to help in designing your error handling. Following these rules will result in code that is easier to understand and use (and, because of that, is likely to have fewer defects).

- Use exceptions only for *exceptional* circumstances.

Operations that raise exceptions for expected outcomes are ergonomically poor. For example, a database lookup operation should not raise an exception if no results are found because an empty result is nothing unusual. If you raise an exception for such an expected outcome, you create an awkward programming style because exceptions break the normal flow of control.

- Make sure that exceptions carry *useful* information.

It is worse than useless to tell the caller something that is already known. For example, if you have an operation that accepts a single value as an `in` parameter, there is no point in returning that value as an exception member when the value is out of range. After all, if only one value was passed, that is the only value that can be out of range.

- Make sure that exceptions carry *precise* information.

An exception should convey precisely one semantic error condition. Do not lump several error conditions together so that the caller can no longer distinguish between them.

- Make sure that exceptions carry *complete* information.

If exceptions carry incomplete information, the caller will probably need to make further calls to find out what went wrong. However, if the initial call did not work, there is a good chance that subsequent ones will not work either.

System Exceptions

CORBA defines 35 system exceptions. (The list is occasionally extended.)

- Any operation can raise a system exception.
- System exceptions must not appear in a **raises** clause.
- All system exceptions have the same exception body:

```
enum completion_status {
    COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
};

exception <SystemExceptionName> {
    unsigned long    minor;
    completion_status    completed;
};
```



29
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.26 System Exceptions

CORBA defines a number of system exceptions that cover infrastructure-related error conditions. Currently, 35 system exceptions are defined. The list of system exceptions is occasionally extended as new functionality is added to CORBA. System exceptions are defined in the CORBA module (see page 2-65).

Any operation can raise a system exception. It is understood that this is the case; it is illegal to add a system exception to the **raises** clause of an operation. (Only user exceptions can appear in a **raises** clause.)

All system exceptions have the same data members: a minor exception code and a completion status.

The completion status indicates at what point during call dispatch an error occurred:

- COMPLETED_YES

The failure occurred sometime after the operation in the server completed. This tells you that any state changes made by the failed invocation have happened.

- COMPLETED_NO

The failure occurred on the way out of the client's address space or on the way into the server address space. It is guaranteed that the target operation was not invoked, or, if it was invoked, no side effects of the operation have taken effect.

- COMPLETED_MAYBE

The completion status is indeterminate. This typically happens if the client invokes an operation and loses connectivity with the server while the call is still in progress. In this case,

there is no way for the client ORB to decide whether the operation was actually invoked in the server or whether the problem occurred before the request reached its target.

The minor code of a system exception is used to provide further information about the cause of an error. The CORBA specification defines a number of minor codes. In addition, ORB vendors can allocate a range of minor codes to use for product-specific information.

```
enum completion_status {
    COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
};

#define SYSEX(NAME) exception NAME {
    unsigned long      minor;
    completion_status completed;
}

SYSEX(BAD_CONTEXT);           // error processing context object
SYSEX(BAD_INV_ORDER);        // routine invocations out of order
SYSEX(BAD_OPERATION);        // invalid operation
SYSEX(BAD_PARAM);            // an invalid parameter was passed
SYSEX(BAD_TYPECODE);         // bad typecode
SYSEX(CODESET_INCOMPATIBLE); // incompatible codeset
SYSEX(COMM_FAILURE);         // communication failure
SYSEX(DATA_CONVERSION);      // data conversion error
SYSEX(FREE_MEM);             // cannot free memory
SYSEX(IMP_LIMIT);            // violated implementation limit
SYSEX(INITIALIZE);           // ORB initialization failure
SYSEX(INTERNAL);             // ORB internal error
SYSEX(INTF_REPOS);           // interface repository unavailable
SYSEX(INVALID_TRANSACTION);  // invalid TP context passed
SYSEX(INV_FLAG);             // invalid flag was specified
SYSEX(INV_IDENT);            // invalid identifier syntax
SYSEX(INV_OBJREF);           // invalid object reference
SYSEX(INV_POLICY);           // invalid policy override
SYSEX(MARSHAL);              // error marshaling param/result
SYSEX(NO_IMPLEMENT);         // implementation unavailable
SYSEX(NO_MEMORY);            // memory allocation failure
SYSEX(NO_PERMISSION);        // no permission for operation
SYSEX(NO_RESOURCES);         // out of resources for request
SYSEX(NO_RESPONSE);          // response not yet available
SYSEX(OBJECT_NOT_EXIST);     // no such object
SYSEX(OBJ_ADAPTER);          // object adapter failure
SYSEX(PERSIST_STORE);        // persistent storage failure
SYSEX(REBIND);               // rebind needed
SYSEX(TIMEOUT);              // operation timed out
SYSEX(TRANSACTION_MODE);     // invalid transaction mode
SYSEX(TRANSACTION_REQUIRED); // operation needs transaction
SYSEX(TRANSACTION_UNAVAILABLE); // no transaction
SYSEX(TRANSACTION_ROLLEDBACK); // operation was a no-op
SYSEX(TRANSIENT);            // transient error, try again later
SYSEX(UNKNOWN);              // the unknown exception
```

Some exceptions have the obvious meaning, whereas others are specific to particular features, such as transactions or the DII. We will discuss these exceptions as we discuss the relevant APIs.

Oneway Operations

IDL permits operations to be declared as **oneway**:

```
interface Events {
    oneway void send(in EventData data);
};
```

The following rules apply to **oneway** operations:

- They must have return type **void**.
- They must not have any **inout** or **out** parameters.
- They must not have a **raises** clause.

Oneway operations provide “best effort” send-and-forget semantics.

Oneway operations may not be delivered, may be dispatched synchronously or asynchronously, and may block.

Oneway is non-portable in CORBA 2.3 and earlier ORBs.



30
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.27 Oneway Operations

Operations declared as **oneway** are meant to provide an unreliable send-and-forget delivery mechanism, similar to UDP datagrams. A **oneway** operation may be lost and never be delivered to the server. The specification guarantees only that it will be delivered at most once. Beyond this, there are no guarantees in the CORBA specification. In particular, it is *not* guaranteed that a **oneway** call will not block the caller, and it is not guaranteed that **oneway** calls will be dispatched asynchronously. Further, **oneway** calls may arrive out of order at the server.

Because **oneway** is underspecified, it is non-portable and different ORBs will use different implementations. For ORBacus, **oneway** calls are guaranteed to not block the caller; **oneway** calls will be lost if the caller sends them quicker than the server can accept them. It is probably best to avoid **oneway** entirely. If you do use **oneway**, be aware that your code is likely to behave differently with a different ORB.

NOTE: The **oneway** keyword will effectively be made obsolete by CORBA 3.0, which instead uses quality-of-service policies to permit the caller to take explicit control over the degree of reliability with which a call is dispatched.

Contexts

Operations can optionally define a **context** clause. For example:

```
interface Poor {
    void doit() context("USER", "GROUP", "X*");
};
```

This instructs the client to send the values of the CORBA context variables **USER** and **GROUP** with the call, as well as the value of all context variables beginning with **X**.

Contexts are similar in concept to UNIX environment variables.

Contexts shoot a big hole through the type system!

Many ORBs do not support contexts correctly, so we suggest you avoid them.



31
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.28 Contexts

CORBA provides the notion of context variables, which are similar in concept to UNIX environment variables. Briefly, a client can maintain a number of named context variables. A context variable (if set) stores a string value.

A `raises` clause on an operation transmits the named context variables as additional parameters to the server. (A ‘*’ character acts as a wild card and must appear as the final character of a context variable name.) The server can examine the variables as part of the operation implementation and use them as additional parameter values.

Contexts create a number of problems with respect to type safety:

- If a particular context variable is not set by the client, its value is (silently) not transmitted to the server.

This means that the server cannot rely on the value of a particular context variable being available even though it appears in the context clause.

- Context variables are untyped.

For the preceding example, the server may expect to find a numerical user ID in the `USER` variable. However, the client may have placed the user name into the variable. (Context variables are strings, so there is no guarantee that a particular string will parse correctly as a particular type of value.)

Of course, this completely destroys type safety because context variables bypass the otherwise strict type checking of the compiler.

Contexts are not supported universally by ORB vendors, so we suggest that you avoid them.

Attributes

An interface can contain one or more attributes of arbitrary type:

```
interface Thermostat {
    readonly attribute short temperature;
    attribute short nominal_temp;
};
```

Attributes implicitly define a *pair* of operations: one to send a value and one to fetch a value.

Read-only attributes define a single operation to fetch a value.

Attributes are *not* state or member variables. They are simply a notational short-hand for operations.

Attributes cannot have a **raises** clause, cannot be **oneway**, and cannot have a **context** clause.

If you use attributes, they should be **readonly**.



32
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.29 Attributes

IDL attributes define a pair of operations to send and fetch a named value. Attributes defined as `readonly` only permit a value to be fetched, but not updated. Even though they look like member variables, attributes are no more than a notational short hand. For example, the following interface is semantically identical to the above one:

```
interface Thermostat {
    short get_temperature();
    void set_nominal_temp(in short temp);
    short get_nominal_temp();
};
```

Attributes are second-class citizens because it is illegal to add a **raises** clause to an attribute definition. This makes it impossible to raise user exceptions to indicate errors. Usually, this becomes an issue if an attribute is writable because, at least in most cases, some values are unacceptable and considered out of range (such as setting a temperature on an air conditioning thermostat to 10,000 degrees). Without user exceptions, you must resort to a system exception to indicate that the attribute update failed. However, system exceptions do not provide enough detail (such as being able to indicate whether the temperature setting is out of range because it is too high or too low).

NOTE: CORBA 3.0 will permit a **raises** clause on attributes. However, since attributes are simply a notational shortcut, it is probably best to avoid them completely and define operations in their place.

Modules

IDL modules provide a scoping construct similar to C++ namespaces:

```

module M {
    // ...
    module L {      // Modules can be nested
        // ...
        interface I { /* ... */ };
        // ...
    };
    // ...
};

```

Modules are useful to prevent name clashes at the global scope.

Modules can contain any IDL construct, including other modules.

Modules can be reopened and so permit incremental definition.



2.30 Modules

IDL offers a `module` construct that is very similar in concept to a C++ namespace. An IDL module definition can contain any IDL construct, including another module definition. The scope rules for modules are the same as for C++ namespaces: names must be unique within a module and, when searching for a name, the compiler successively looks in enclosing modules followed by the global scope.

You can use qualified names to explicitly refer to something in another module. For example, `::M::L::I` denotes the interface `I` above, regardless from the context in which it is used.

Modules can be reopened:

```

module A {
    // Some definitions here...
};
module B {
    // Some other definitions here...
};
module A {
    // Reopen module A and add to it...
};

```

Reopening of modules is useful because it shields developers from changes. For example, by splitting a large definition into multiple module definitions, they can be maintained in different source files and modified independently. That way, a small change does not force recompilation of the entire system.

Forward Declarations

IDL permits forward declarations of interfaces so they can be mutually dependent:

```
interface Husband; // Forward declaration
```

```
interface Wife {  
    Husband get_spouse();  
};
```

```
interface Husband {  
    Wife get_spouse();  
};
```

The identifier in a forward declaration must be a simple (non-qualified) name:

```
interface MyModule::SomeInterface; // Syntax error!
```



34
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.31 Forward Declarations

Forward declarations of interfaces permit mutually dependent interfaces, as shown above. Multiple forward declarations of the same interface are legal. You can use a forward-declared interface as a parameter type, return type, or member type. However, you cannot inherit from a forward-declared interface until after it is defined.

The name for a forward declaration must be simple, non-qualified identifier. This means that you cannot use a forward declaration to refer to an interface in another module. If you really require mutually dependent interface in different modules, you can achieve this by reopening a module:

```
module Males {  
    interface Husband;  
};  
  
module Females {  
    interface Wife {  
        Males::Husband get_spouse();  
    };  
};  
  
module Males {  
    interface Husband {  
        Females::Wife get_spouse();  
    };  
};
```

Inheritance

IDL permits interface inheritance:

```
interface Thermometer {
    typedef short TempType;
    readonly attribute TempType temperature;
};
```

```
interface Thermostat : Thermometer {
    void set_nominal_temp(in TempType t);
};
```

You can pass a derived interface where a base interface is expected:

```
interface Logger {
    long add(in Thermometer t, in unsigned short interval);
    void remove(in long id);
};
```

At run time, you can pass a **Thermometer** or a **Thermostat** to **add**.



35
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.32 Inheritance

IDL supports inheritance. In the above example, a *Thermostat* is-a *Thermometer*. A *Thermostat* automatically has the inherited *temperature* attribute as well as the *set_nominal_temp* operation. Scope resolution for inheritance works as for C++: identifiers are resolved by successively searching base interfaces toward the root. This rule allows *TempType* to be used without qualification inside interface *Thermostat*. (We could have also used *Thermometer::TempType* or *::Thermometer::TempType*.)

Inheritance gives rise to polymorphism and obeys the Liskov substitution principle: a derived interface can be treated as if it were a base interface and so can be passed where a base interface is expected. The above *Logger* interface illustrates this. The parameter *t* of the *add* operation has the type *Thermometer*. However, at run time, you can pass either a *Thermometer* or a *Thermostat*. If the *Logger* reads the *temperature* attribute via the passed reference, the call will be dispatched to the most derived implementation, that is, late binding applies.

Inheritance from Object


All IDL interfaces implicitly inherit from type **Object**:

```

classDiagram
    class Object
    class Thermometer
    class Logger
    class Thermostat
    Object <|-- Thermometer
    Object <|-- Logger
    Thermometer <|-- Thermostat
  
```

You must not explicitly inherit from type **Object**.


Because all interfaces inherit from **Object**, you can pass any interface type as type **Object**. You can determine the actual type of an interface at run time with a safe down-cast.



36

The OMG Interface Definition Language

Copyright 2000–2001 IONA Technologies



2.33 Inheritance from Object

All IDL interfaces implicitly have type `Object` as their base interface. The above inheritance diagram illustrates this implicit inheritance for the IDL we saw on page 2-49. Because all interfaces are type compatible with `Object`, you can use `Object` to create operations that can generically deal with any type of interface. For example:

```
interface Generic {
    void    accept(in KeyType key, Object o);
    Object  lookup(in KeyType key);
};
```

If need be, the implementation of an operation can use a type-safe down-cast to narrow an object reference to a more specific type (see page 6-18).

Explicit inheritance from `Object` is illegal:

```
interface Wrong : Object { /*...*/ }; // Error
```

Note that IDL deals only with *interface* inheritance. Interface inheritance is purely a mechanism to determine type compatibility and has nothing to do with implementation. In particular, IDL interfaces that are in an inheritance relationship may or may not be implemented using inheritance, and instances of the base and derived interfaces can be implemented in different address spaces.

Inheritance Redefinition Rules

You can redefine types, constants, and exceptions in the derived interface:

```
interface Thermometer {
    typedef long    IDType;
    const IDType   TID = 5;
    exception      TempOutOfRange {};
};

interface Thermostat : Thermometer {
    typedef string IDType;                // Yuk!
    const IDType   TID= "Thermostat";    // Aargh!
    exception      TempOutOfRange { long temp; }; // Ick!
};
```

While legal, this is too terrible to even contemplate. Do not do this!



2.34 Inheritance Redefinition Rules

Derived interfaces can legally redefine inherited types, constants, and exceptions of the same name. Obviously, doing so is extremely confusing and should be avoided. No other aspects of a base interface can be redefined in a derived interface.

Inheritance Limitations

You cannot override attributes or operations in a derived interface:

```
interface Thermometer {
    attribute long    temperature;
    void            initialize();
};

interface Thermostat : Thermometer {
    attribute long    temperature;           // Error!
    void            initialize();           // Error!
};
```

It is understood that a **Thermostat** already has an inherited **temperature** attribute and **initialize** operation and you are not allowed to restate this.

Overriding is a meaningless concept for *interface* inheritance!



38
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.35 Inheritance Limitations

As shown above, you cannot override attributes or operations in a derived interface. In effect, all attribute and operation names used in a base interface are “used up” in the derived interface. This rule also implies that overloading is not supported in any form:

```
interface Thermometer {
    attribute string    my_id;
    string            get_id();
    void            set_id(in string s);
};

interface Thermostat : Thermometer {
    attribute double    my_id;           // Illegal!
    double            get_id();         // Illegal!
    void            set_id(in double d); // Illegal!
};
```

Overloading is not supported because, for languages without direct support for overloading, it would result in very difficult to use language mappings. (The mapping would have to use mangled names, which is fine for compilers, but not for humans.)

Multiple Inheritance

Multiple inheritance, including inheritance of the same base interface multiple times, is supported:

```

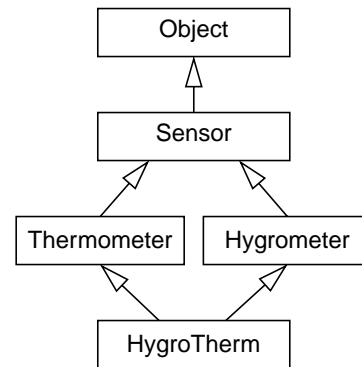
interface Sensor {
    // ...
};

interface Thermometer : Sensor {
    // ...
};

interface Hygrometer : Sensor {
    // ...
};

interface HygroTherm : Thermometer, Hygrometer {
    // ...
};

```



39
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.36 Multiple Inheritance

As shown above, multiple inheritance is supported. It is legal for interfaces to inherit the same base interface multiple times via different intermediate interfaces. The declaration order of base interfaces is not significant. Because IDL only deals with interfaces, there is no notion of virtual or non-virtual inheritance; because virtual versus non-virtual inheritance deals with implementation concerns (namely, whether the state of base implementations is shared or not), the concept simply does not apply to IDL.⁶

6. When you consider that interfaces in an inheritance relationship may not be implemented using inheritance (such as in C), this makes perfect sense.

Scope Rules for Multiple Inheritance

You cannot inherit the same attribute or operation from more than one base interface:

```
interface Thermometer {
    attribute string    model;
                    void    reset();
};

interface Hygrometer {
    attribute string    model;
                    string    reset();
};

interface HygroTherm : Thermometer, Hygrometer { // Illegal!
    // ...
};
```



40
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.37 Scope Rules for Multiple Inheritance

If an interface inherits from more than one base interface none of the attribute and operation names used in the base interfaces may overlap. The reason for this restriction is that it is awkward to map into languages without mechanisms to resolve the ambiguity of which base operation to call if the operation is invoked on the derived interface.⁷

⁷. This restriction may be removed in a future version of CORBA.

Scope Rules for Multiple Inheritance (cont.)

You can multiply inherit ambiguous types, but you must qualify them explicitly at the point of use:

```
interface Thermometer {
    typedef string<16> ModelType;
};

interface Hygrometer {
    typedef string<32> ModelType;
};

interface HygroTherm : Thermometer, Hygrometer {
    attribute ModelType          model;          // Error!
    attribute Hygrometer::ModelType model;      // OK
};
```



41
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



If you inherit the same type name from multiple base interfaces, you must use a qualified name to identify the exact type. In the above example, the unqualified name `ModelType` is ambiguous in the derived interface because it would be unclear whether the type denotes a 16-byte or a 32-byte string. (`ModelType` would be ambiguous even if both base interfaces defined the type identically.) Using a qualified name, such as `Hygrometer::ModelType`, removes the ambiguity and is therefore legal.

IDL Scope Resolution

The following IDL constructs establish naming scopes:

- modules, interfaces, structures, unions, exceptions, parameter lists

Within a naming scope, names must be unique.

To resolve a name, the compiler searches:

1. the current scope
2. if the current scope is an interface, the base interfaces toward the root
3. enclosing scopes of the current scope
4. the global scope

Names not qualified as being part of the global scope with a leading `::` operator are introduced into the current scope when first used.



42
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.38 IDL Scope Resolution

NOTE: The scope resolution rules are presented here for the sake of completeness. If you use sensible names for your IDL constructs, most of these rules will never concern you.

2.38.1 Uniqueness of identifiers

IDL modules, interfaces, structures, unions, exceptions, and parameter lists form naming scopes. Within a naming scope, identifiers must be distinct and must denote the same thing throughout that scope. Intuitively, this means that different things cannot have the same name in the same scope. For example:

```
struct Bad {
    short    temperature;
    long     Temperature;    // Error!
    Temp     temp;          // Error!
};

typedef string SomeType;

interface AlsoBad {
    void op1(in SomeType t, in double t);    // Error!
    void op2(in Temp temp);                 // Error!
    void op3(in sometype t);                 // Error!
};
```

Note that identifiers that differ only in capitalization are considered the same identifier. In addition, once used, you must use the same capitalization for an identifier throughout.

2.38.2 Scope Resolution Rules

The following example illustrates some of the scope resolution rules:

```
module CCS {
    typedef short    TempType;
    const TempType  MAX_TEMP = 99;    // Max_TEMP is a short

    interface Thermostat {
        typedef long    TempType;    // OK

        TempType        temperature(); // Returns a long
        CCS::TempType   nominal_temp(); // Returns a short
    };
};
```

Note that the redefinition of `TempType` inside `Thermostat` is legal (if ugly). Inside `Thermostat`, `TempType` denotes a `long` value. You can use a qualified name (`CCS::TempType` or `::CCS::TempType`) to refer explicitly to the type defined in the `CCS` module.

In the presence of inheritance, base interfaces are searched before the enclosing scopes of the derived interface:

```
module Sensors {
    typedef short    TempType;
    typedef string   AssetType;

    interface Thermometer {
        typedef long    TempType;

        TempType        temperature();    // Returns a long
        AssetType        asset_num();     // Returns a string
    };
};

module Controllers {
    typedef double   TempType;

    interface Thermostat : Sensors::Thermometer {
        TempType        nominal_temp();    // Returns a long
        AssetType        my_asset_num();   // Error!
    };
};
```

In this example, `nominal_temp` does *not* return a `double` because the definition of `TempType` in the `Thermometer` base interface hides the definition of `TempType` in the enclosing scope of `Thermostat`. Also note that the definition of `my_asset_num` is in error because the compiler never finds a definition for `AssetType`. This illustrates that only base interfaces are searched for definitions, but not the enclosing scope of base interfaces.

2.38.3 Implicitly Introduced Identifiers

Using a name that is *not* explicitly qualified with a leading `::` operator introduces that name into the current scope. Using a name that *is* explicitly qualified with a leading `::` operator does *not* introduce that name into the current scope. Here is an example to illustrate this:

```
typedef string ModelType;

module CCS {
    typedef short    TempType;
    typedef string  AssetType;
};

module Controllers {
    typedef CCS::TempType    Temperature; // Introduces CCS _only_
    typedef string          ccs;         // Error!
    typedef long            TempType;    // OK
    typedef ::CCS::AssetType AssetType;  // OK
};

struct Values {
    ::ModelType ModelType; // OK
    ::modelType ModelType2; // Error!
};
```

Note that, once `CCS::TempType` is used inside `Controllers`, the identifier `CCS` is introduced into the current scope and therefore no longer available to name other constructs. If a qualified name consists of several identifiers, only the first identifier is introduced into the current scope. For example, the name `A::B::C` only introduces `A` and leaves `B` and `C` available.

Implicit introduction of identifiers takes place only for relative qualified names. Absolute qualified names (which use a leading `::` operator) do not introduce any identifier into the current scope. This explains why the definition of the member `Values::ModelType` is legal. (Note that the definition of `Values::ModelType2` is illegal because of the requirement for consistent capitalization of identifiers.)

If you implicitly introduce an enumerated type into the current scope, you must still qualify the enumerators:

```
interface Sensor {
    enum DeviceType { READ_ONLY, READ_WRITE };
};

interface Thermometer : Sensor {
    union ThermData switch (DeviceType) {
        case Sensor::READ_ONLY:
            unsigned long    read_addr;
        case READ_WRITE:
            unsigned long    write_addr; // Error!
    };
};
```

Even though `DeviceType` is implicitly introduced as the discriminator type, this does not introduce the enumerators.

Nesting Restrictions

You cannot use directly nested constructs with the same name:

```

module M {
    module X {
        module M { /* ... */ }; // OK
    };
    module M { /* ... */ }; // Error!
};

struct S {
    long s; // Error!
};

interface I {
    void I(); // Error!
};

```



43
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.39 Nesting Restrictions

Named constructs that are directly nested cannot have the same name. In the above example, `M::X::M` is legal, but `M::M`, `S::s`, and `I::I` are not.

NOTE: This rule was added with CORBA 2.3 because of the difficulty of mapping it to C++ and Java, in which the direct nesting of same-named scopes causes clashes. (This affects C++ in particular because the name of a class is reserved for the constructor and therefore cannot be used for anything else, such as a data member or member function.)

Anonymous Types

Anonymous types are currently legal in IDL, but CORBA 3.0 will deprecate them.

Anonymous types cause problems for language mappings because they create types without well-defined names.

You should avoid anonymous types in your IDL definitions.

For recursive structures and unions, they cannot be avoided in CORBA 2.3 and earlier versions.

CORBA 2.4 provides a forward declaration for structures and unions, so anonymous types can be avoided completely.

If you name all your types, you will never have a problem!



44
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.40 Anonymous Types

The following constructs are legal IDL, but use anonymous types. In each case, it is preferable to use a `typedef` to explicitly give the type a name. This not only avoids problems in several implementation languages, but also makes your IDL future-proof because anonymous types have been deprecated and will likely be made illegal in a future version.

Anonymous bounded strings and bounded wide strings should be explicitly named. Currently, they are legal (but deprecated) in constant definitions, attribute declarations, return type and parameter type declarations, sequence and array element declarations, and as the type of structure, union, and exception members. For example:

```
const string<5> GREETING = "Hello";           // Deprecated

interface Deprecated {
    readonly attribute wstring<5> name;       // Deprecated
    wstring<5> op(in wstring<5> param);       // Deprecated
};
typedef sequence<wstring<5> > WS5Seq;        // Deprecated
typedef wstring<5> Name[4];                 // Deprecated

struct Foo {
    wstring<5> member;                         // Deprecated
};

// Similar for unions and exceptions...
```


By using typedef, you can avoid all anonymous types:

```
typedef string<5>    GreetingType;
typedef wstring<5>  ShortWName;

const GreetingType GREETING = "Hello";

interface OK {
    readonly attribute ShortWName name;
    ShortWName op(in ShortWName param);
};
typedef sequence<ShortWName>    WS5Seq;
typedef ShortWName              Name[4];

struct Foo {
    ShortWName member;
};
```

The same considerations apply to fixed-point types in attribute declarations, return type and parameter type declarations, sequence and array element declarations, and for structure, union, and exception member types. For example:

```
interface Account {
    fixed<10,2> balance;    // Deprecated
};
```

This is better written as:

```
typedef fixed<10,2> BalanceType;

interface Account {
    BalanceType balance;
};
```

Anonymous member types for structures, unions, and exceptions are deprecated. For example:

```
exception E {
    long          array_mem[10];    // Deprecated
    sequence<long> seq_mem;        // Deprecated
    string<5>     bstring_mem;     // Deprecated
};
```

This is better expressed by naming each type:

```
typedef long          LongArray[10];
typedef sequence<long> LongSeq;
typedef string<5>     ShortString;

exception E {
    LongArray  array_mem;
    LongSeq    seq_mem;
    ShortString string_mem;
};
```

You should also avoid anonymous sequence and array elements:

```
typedef sequence<sequence<long> > NumberTree;
typedef fixed<10,2>          FixedArray[10];
```

Instead, provide a separate type definition for each element type:

```
typedef sequence<long>          ListOfNumbers;
typedef sequence<ListOfNumbers> NumberTree;
typedef fixed<10,2>            Fixed_10_2;
typedef Fixed_10_2             FixedArray[10];
```

NOTE: With CORBA 2.3 and earlier ORBs, it is impossible to avoid an anonymous member type for recursive structures and unions. For example, there is no way to avoid the anonymous member type in the following:

```
struct Node {
    long          val;
    sequence<Node,2> children; // Anonymous member type
};
```

From CORBA 2.4 onwards, the following will be legal:

```
typedef struct          Node; // Forward declaration
typedef sequence<Node,2> ChildSeq;

struct Node {
    long          val;
    ChildSeq      children; // Avoids anonymous type
};
```

This avoids the anonymous member type.

Repository IDs

The IDL compiler generates a repository ID for each identifier:

```

module M {
    typedef short T;
    interface I {
        attribute T a;
    };
};

```

// IDL:M:1.0
// IDL:M/T:1.0
// IDL:M/I:1.0
// IDL:M/I/a:1.0

The repository ID uniquely identifies each IDL type.

You must ensure that repository IDs are unique.

If you have two IDL specifications with the same repository IDs but different meaning, CORBA's type system is destroyed!



2.41 Repository IDs

When you compile an IDL specification, the compiler creates a unique identifier for each type, known as a repository ID. Repository IDs have the general form `IDL:<name>:1.0`.⁸ The `<name>` component is the fully-qualified name of each type, using `/` as a separator.

Repository IDs serve as unique type handles and, in many cases, are exchanged between client and server as an indication of what data is being transmitted. This means that all clients and servers that communicate with each other must use the same repository ID to denote the same IDL type; otherwise, the CORBA type system falls apart because sender and receiver will no longer agree on the meaning of types and how to interpret data during marshaling.

The need for repository IDs to be unique provides the main motivation for IDL modules: if you place all the IDL for your application into a module, name clashes with other developers' type names become less likely (but not impossible).

8. There are other repository ID formats, such as `DCE:` and `LOCAL:`. However, these do not have any practical significance, so we do not cover them here.

Controlling Repository ID Prefixes

You should routinely add a **#pragma prefix** to your IDL definitions:

```
#pragma prefix "acme.com"
```

```
module M {                                // IDL:acme.com/M:1.0
    typedef short T;                       // IDL:acme.com/M/T:1.0
    interface I {                           // IDL:acme.com/M/I:1.0
        attribute T a;                       // IDL:acme.com/M/I/a:1.0
    };
};
```

#pragma prefix adds the specified prefix to each repository ID.

Use of a prefix makes name clashes with other repository IDs unlikely.



46
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.42 Controlling Repository ID Prefixes

The IDL **#pragma prefix** directive adds the specified prefix to each repository ID. By using a name that you own (such as a registered domain name), name clashes can be avoided.

Of course, you could also avoid name clashes by using a unique module name. For example:

```
module acme_com {                          // IDL:acme_com:1.0
    module M {                               // IDL:acme_com/M:1.0
        // ...
    };
};
```

This approach also makes name clashes unlikely. However, it has the drawback that it can result in rather ugly and long identifiers in some language mappings (such as `acme_com_M` for the C mapping).

A **#pragma prefix** does not suffer from this problem because the prefix is invisible in the generated API and therefore does not add additional nested scopes or cause long and ugly identifiers.

Predefined IDL

The CORBA specification defines a number of IDL types in the **CORBA** module.

For example, the definition for **TypeCode** (a type that describes types) and the definitions for the Interface Repository (IFR) are in the **CORBA** module.

To use such predefined types in your IDL, you must include **orb.idl**:

```
#include <orb.idl> // Get access to CORBA module

interface TypeRepository {
    CORBA::TypeCode lookup(in string name);    // OK
    // ...
};
```



2.43 Predefined IDL

The CORBA specification defines a number of IDL types that all ORBs must provide. For example, ORBs use an IDL interface `TypeCode` which provides functionality to enquire about the details of a type at run time. (`TypeCode` is used, for example, to provide introspection for values of type any.)

Most of the predefined IDL types are nested in the OMG-defined CORBA module. In order to use types defined in the CORBA module, such as `TypeCode`, you must include `orb.idl` in your specification to import the relevant definitions. You can then refer to types in the CORBA module by using a qualified name (`CORBA::TypeCode` or `::CORBA::TypeCode`).

Using the IDL Compiler

The IDL compiler is called `idl`. By default, for a file called `x.idl`, it produces:

<code>x.h</code>	client-side header file
<code>x.cpp</code>	client-side (stub) source file
<code>x_skel.h</code>	server-side header file
<code>x_skel.cpp</code>	server-side (skeleton) source file

Major options:

<code>-D<name>[=<val>]</code>	define preprocessor symbol <code><name></code> [with value <code><val></code>]
<code>-U<name></code>	undefine preprocessor symbol
<code>-I<dir></code>	add directory to include search path
<code>-E</code>	run the preprocessor only
<code>--c-suffix <s></code>	change default <code>cpp</code> extension to <code><s></code>
<code>--h-suffix <s></code>	change default <code>h</code> extension to <code><s></code>
<code>--impl</code>	generate implementation files



48
The OMG Interface Definition Language
Copyright 2000–2001 IONA Technologies



2.44 Using the IDL Compiler

The ORBacus IDL-to-C++ compiler is called `idl`.⁹ It compiles one or more IDL files provided as arguments.¹⁰ For example:

```
$ idl x.idl y.idl
```

By default, for each IDL source file, the compiler generates two client-side and two server-side files, with `.h` and `.cpp` as the default extensions for header and source files, respectively.

The `idl` command supports a large number of options. The most frequently used options are to control the preprocessor and to change the default file extensions. You can also suppress the generation of server-side files with the `--no-skeletons` option, to avoid cluttering the build directory with unwanted files if you are writing client-side code only.

One option that is useful is `--impl`. It causes the compiler to generate a header and source file (called `x_impl.h` and `x_impl.cpp`) for the server application code, with operation implementations that are no-ops. This option is useful because it saves you having to cut and paste definitions from the generated skeleton file. (The `--impl` option does not override existing files.)

The compiler supports a large number of other options (see the man page for details).

9. Note that the specification does not define the name of the IDL compiler command, the available options, or the name and number of the generated files. Such issues are considered outside the scope of the specification and therefore are vendor-specific.

10. Whenever we show commands in this course, we assume a Bourne shell.

Topics Not Covered Here

There are a few parts of IDL we did not cover in this unit:

- **#pragma ID**

This pragma allows you to selectively change the repository ID for a particular type.

- **#pragma version**

This pragma allows you to change the version number for **IDL :** format repository IDs.

- **Objects By Value (OBV)**

OBV provides a hybrid of structures with inheritance and objects that are passed by value instead of by reference.

OBV is large and complex and covered in a separate unit.



2.45 Topics Not Covered Here

The above list presents a list of topics we did not cover in this unit. IDL supports two additional pragma directives (which are unlikely to ever be of importance to you). A major part of functionality is known as Objects by Value. However, the topic is large and complex, so we do not cover it in this unit.

3. Exercise: Writing IDL Definitions

Summary

This exercise provides you with hands-on experience of writing IDL definitions by asking you to solve a simple interface design problem.

Objectives

By the completion of this exercise, you will have gained experience in the syntax and semantics of IDL definitions, be able to write IDL definitions, and to compile these definitions into C++ stubs and skeletons.

The Climate Control System

The climate control system consists of:

- Thermometers

Thermometers are remote sensing devices that report the temperature at various location.

- Thermostats

Thermostats are like thermometers but also permit a desired temperature to be selected.

- A single control station

A control station permits an operator to monitor all devices and to change the temperature in various parts of a building remotely.

The devices in the system use a proprietary instrument control protocol. We need a CORBA interface to this system.



1
Exercise: Writing IDL Definitions
Copyright 2000–2001 IONA Technologies



3.1 The Climate Control System

For the remainder of this course, we will use a simple climate control system (CCS) as an example application. The CCS controls the air-conditioning for various rooms in one or more buildings. In addition, the system permits control of the temperature of manufacturing devices, such as freezers and semiconductor annealing ovens. The devices are installed at various locations and accessed via a proprietary instrument control protocol. We want to provide a CORBA interface to this system to integrate it with the remainder of the IT infrastructure.

The system contains a number of thermometers and thermostats. Thermostats, in addition to reporting the temperature around them also permit a desired (or nominal) temperature to be selected. The CCS system attempts to maintain the actual temperature around a thermostat as close as possible to the nominal temperature.

The entire system can be controlled from a central monitoring station that permits an operator to monitor and adjust the temperature at different locations.

A CORBA server will act as a gateway to this system, by receiving CORBA messages and translating them into corresponding messages that are sent to devices via the proprietary instrument control protocol.

Thermometers

Thermometers are simple devices that report the temperature and come with a small amount of non-volatile memory that stores additional information:

- Asset number (read-only)

Each thermometer has a unique asset number, assigned when it is manufactured. This number also serves as the device's network address.

- Model (read-only)

Each thermometer can report its model (such as "Sens-A-Temp").

- Location (read/write)

Each thermometer has non-volatile RAM that can be set to indicate the device's location (such as "Annealing Oven 27" or "Room 414").



2
Exercise: Writing IDL Definitions
Copyright 2000–2001 IONA Technologies



3.2 Thermometers

Thermometers are simple devices that report the current temperature. Each thermometer has a unique read-only asset number (which also acts as the network address for the instrument control protocol), a read-only model string, and location string (which can be set remotely).

Thermostats

Thermostats are like thermometers:

- They can report the temperature, and have an asset number, model, and location.
- The asset numbers of thermometers and thermostats do not overlap. (No thermostat has the same asset number as a thermometer).
- Thermostats permit remote setting of a nominal temperature.
- The CCS attempts to keep the actual temperature as close as possible to the nominal temperature.
- The nominal temperature has a lower and upper limit to which it can be set.
- Different thermostats have different temperature limits (depending on the model).



3
Exercise: Writing IDL Definitions
Copyright 2000–2001 IONA Technologies



3.3 Thermostats

Thermostats are just like thermometers but also permit a desired temperature to be selected. The CCS system attempts to keep the actual temperature as close as possible to the selected temperature by controlling other devices, such as heaters or refrigeration units (which are not further considered for this example). It is possible to remotely read the current setting of a thermostat, as well as change it.

Thermostats, depending on their model, have different temperature ranges they can be set to. For example, a thermostat for an annealing oven has a different temperature range than a thermostat for an air-conditioner. Attempts to set a thermostat outside its legal range of nominal temperatures are reported by the instrument control protocol.

The Monitoring Station

The monitoring station provides central control of the system. It can:

- read the attributes of any device
- list the devices that are connected to the system
- locate devices by asset number, location, or model
- update a number of thermostats as a group by increasing or decreasing the current temperature setting relative to the current setting



4
Exercise: Writing IDL Definitions
Copyright 2000–2001 IONA Technologies



3.4 The Monitoring Station

The monitoring station (known as the controller), permits access to and control of the devices in the system. An operator can list all devices in the system, locate specific devices by various search criteria, and make relative changes to the temperature setting of a group of thermostats.

3.4.1 Listing Devices

A list operation returns a list of all devices connected to the system.

3.4.2 Making a Relative Temperature Change

A change operation permits an operator to increase or decrease the temperature of a group of thermostats by a delta value. (This is useful to, for example, decrease the temperature of rooms along the western side of a building during summer without disturbing the relative setting of the thermostats in those rooms.)

Some thermostats may not be able to raise or lower their temperature setting by the requested amount because they may already be set at or close to their temperature limit. In such cases, a change operation sets those thermostats that can make the change in full to the new temperature; those thermostats that are already at or near their limit are set to the limit. In addition, an error report provides the details of all those changes that did not succeed in full.

3.4.3 Searching for Devices

A search operation permits an operator to find all devices matching a specified set of asset numbers, matching one or more locations, or matching one or more model descriptions.

3.5 What You Need to Do

Create an IDL definition that captures the functionality of the climate control system.

Place your IDL definition into a file named `CCS.idl` and compile it using the `idl` command.

We suggest that you develop the IDL incrementally, compiling occasionally to verify that your approach is free from static errors up to that point.

Remember what you learned about name clashes and how to prevent them and use your knowledge to avoid such name clashes.

Think about what interfaces should be present in your IDL and how these interfaces should relate to each other.

Consider the various error conditions that may arise and how to best deal with them.

Note that there are many ways to capture this particular problem domain in a specification. There is not necessarily a single one and true way. The purpose of this exercise is to get you to think about the different approaches and their trade-offs, and to familiarize you with the IDL syntax and the compiler.

When you have compiled your specification, have a look at the generated header and source files. What parts of the specification do you recognize in the generated code?

4. Solution: Writing IDL Definitions

4.1 IDL for the Climate Control System

Below is one possible solution to the exercise. Note that this is by far not the only possible solution. In addition, we have made the IDL deliberately complex (particularly for the `find` operation) in order to exercise a representative subset of IDL features for use in later units. You should find it an instructive exercise to criticize this specification and look for ways to improve it.

```
#pragma prefix "acme.com"

module CCS {
    typedef unsigned long    AssetType;
    typedef string          ModelType;
    typedef short           TempType;
    typedef string          LocType;

    interface Thermometer {
        readonly attribute ModelType    model;
        readonly attribute AssetType    asset_num;
        readonly attribute TempType     temperature;
        attribute LocType               location;
    };

    interface Thermostat : Thermometer {
        struct BtData {
            TempType    requested;
            TempType    min_permitted;
            TempType    max_permitted;
            string      error_msg;
        };
        exception BadTemp { BtData details; };

        TempType    get_nominal();
        TempType    set_nominal(in TempType new_temp)
            raises(BadTemp);
    };

    interface Controller {
        typedef sequence<Thermometer>    ThermometerSeq;
        typedef sequence<Thermostat>    ThermostatSeq;

        enum SearchCriterion { ASSET, LOCATION, MODEL };

        union KeyType switch(SearchCriterion) {
            case ASSET:
                AssetType    asset_num;
            case LOCATION:
                LocType      loc;
            case MODEL:
                ModelType    model_desc;
        };

        struct SearchType {
            KeyType    key;
            Thermometer device;
        };
    };
};
```



```

};
typedef sequence<SearchType>    SearchSeq;

struct ErrorDetails {
    Thermostat          tmstat_ref;
    Thermostat::BtData  info;
};
typedef sequence<ErrorDetails>  ErrSeq;

exception EChange {
    ErrSeq  errors;
};

ThermometerSeq  list();
void            find(inout SearchSeq slist);
void            change(
                in ThermostatSeq tlist, in short delta
                ) raises(EChange);
};
};

```

4.1.1 Some Notes About This Specification

Much of the specification is self-explanatory. The general structure is shown in the following UML diagram:



We have used inheritance here to capture that thermostats are like thermometers with additional functionality. (Note that this does not mean that we have to implement a thermostat by inheriting from a thermometer.) This inheritance relationship is exploited in the `list` operation, which returns a sequence of thermometers; that is, at run time, the actual sequence elements can be either thermometers or thermostats.

The controller acts as a collection manager for devices. Note that we can locate devices by navigation from the controller to each device. However, the converse is not true; given a device, we cannot find out which controller is responsible for it.

The `change` operation is fairly self-explanatory. It accepts a sequence of thermostats and a delta value and applies the requested temperature change to the supplied thermostats. Some complexity arises from the need to provide detailed error information for those thermostats that could not make the change; that information is returned in an `EChange` exception.

The `find` operation is quite complex. Note how it accepts a sequence of search records, each of which contains a union to indicate by what criteria to search. Also note that the search sequence is an `inout` parameter. This allows the operation to fill in the object reference member of each element with the object reference of a matching device (or a `nil` reference if no matching device is found). If more than one device matches a specific search key, `find` can grow the length of the returned sequence.

Spend some time thinking about the design of `find`. It has many drawbacks that could be remedied quite easily. Also (even though we have not yet seen the C++ mapping), consider the steps a caller has to go through in order to set up a search and to process the results. How would you simplify this?

NOTE: We made `find` deliberately complex because it serves to illustrate a number of features of the C++ mapping. A realistic design would (hopefully) look quite different.

If you had a look at the C++ files that are generated by the IDL compiler, you will probably have realized that they are not very readable. As a general rule, these files are not meant for human consumption. Quite often, they use complex macros or cryptic work-arounds for compiler bugs. You should resist the temptation to look at the generated code to work out how to write your application. As we will see in Unit 5, it is easier to learn how the C++ mapping works and to look at the IDL specification instead of the generated files when writing your code.

5. Basic C++ Mapping

Summary

This unit presents the mapping of IDL types to C++. The part of the C++ mapping covered here applies to both client and server. The client-side mapping is covered in Unit 6 and the server-side mapping is presented in Unit 9.

Objectives

By the completion of this unit, you will know how to manipulate IDL data types in C++, including how to correctly deal with memory management for these types.

Introduction

The basic C++ mappings defines how IDL types are represented in C++. It covers:

- mapping for identifiers
- scoping rules
- mapping for built-in types
- mapping for constructed types
- memory management rules

For each IDL construct, the compiler generates a definition into the client-side header file, and an implementation into the client-side stub file.

General definitions (in the **CORBA** module) are imported with

```
#include <OB/CORBA.h>
```



1
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.1 Introduction

The basic C++ mapping covers how IDL types map to C++ types and includes rules for the mapping of identifiers, preserving IDL naming scopes, and memory management rules. The client-side mapping (which deals with how to invoke operations) is covered in Unit 6 and the server-side mapping (which deals with how to implement objects) is covered in Unit 9.

Mapping for Identifiers

- IDL identifiers map to corresponding C++ identifiers:

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

The generated C++ contains:

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

- IDL identifiers may clash with C++ keywords:

```
enum class { if, this, while, else };
```

Such identifiers are mapped with a `_cxx_` prefix:

```
enum _cxx_class {  
    _cxx_if, _cxx_this, _cxx_while, _cxx_else  
};
```

You should avoid using IDL identifiers that are likely to be keywords in one or more programming languages.



5.2 Mapping for Identifiers

As you can see above, IDL identifiers map into C++ identifiers without change. There is one exception to this rule: if an IDL identifier happens to be a C++ keyword, it is escaped by prefixing it with `_cxx_`. This results in legal but ugly code, so you should try and avoid IDL identifiers that are likely to be programming language keywords.

NOTE: These examples coincidentally also cover the mapping for enumerated types. As you can see, IDL enumerations map unchanged to C++ enumerations.

Scoping Rules

IDL scopes are preserved in the mapped C++. For example:

```
interface I {  
    typedef long L;  
};
```

As in IDL, you can refer to the corresponding constructs as `I` or `::I` and as `I::L` or `::I::L`.

The specific kind of C++ scope a particular IDL scope maps to depends on the specific IDL construct.



5.3 Scoping Rules

The C++ mapping preserves IDL scoping rules. This means that if you have an identifier that you can refer to as `::I::L` in IDL, you can refer to the corresponding identifier as `::I::L` in C++.

Depending on what type an IDL naming scope represents, it may end up as a C++ namespace, class, or structure. However, nesting of scopes is preserved regardless of the specific C++ construct that is generated by the IDL compiler.

Mapping for Modules

IDL modules map to C++ namespaces:

```
module Outer {
    // More definitions here...
    module Inner {
        // ...
    };
};
```

This maps to:

```
namespace Outer {
    // More definitions here...
    namespace Inner {
        // ...
    }
}
```



5.4 Mapping for Modules

IDL modules map one-to-one to C++ namespaces. If an IDL module is reopened, so is the corresponding namespace:

```
module M1 {
    // Some M1 definitions here...
};
module M2 {
    // M2 definitions here...
};
module M1 {
    // More M1 definitions here...
};
```

This translates to C++ as:

```
namespace M1 {
    // Some M1 definitions here...
}
namespace M2 {
    // M2 definitions here...
}
namespace M1 {
    // More M1 definitions here...
}
```

Mapping for Built-In Types

IDL built-in types map to type definitions in the CORBA namespace:

IDL	C++
short	CORBA::Short
unsigned short	CORBA::UShort
long	CORBA::Long
unsigned long	CORBA::ULong
long long	CORBA::LongLong
unsigned long long	CORBA::ULongLong
float	CORBA::Float
double	CORBA::Double
long double	CORBA::LongDouble

The type definitions are used to hide architecture-dependent size differences.

You should use these type names for portable code.



5.5 Mapping for Built-In Types

All built-in IDL types map to names in the CORBA namespace (except for `string` and `wstring`, which map to pointer types).

The integer and floating-point types map to type definitions in the CORBA namespace. The reason for mapping, for example, IDL `long` to `CORBA::Long` instead of C++ `long` is that it helps to hide architecture-dependent size differences. By using the type definitions instead of a specific native C++ type, you avoid problems when porting, for example, from a 32-bit to a 64-bit environment.

Mapping for Built-In Types (cont.)

IDL	C++
char	<code>CORBA::Char</code>
wchar	<code>CORBA::WChar</code>
string	<code>char *</code>
wstring	<code>CORBA::WChar *</code>
boolean	<code>CORBA::Boolean</code>
octet	<code>CORBA::Octet</code>
fixed<n,m>	<code>CORBA::Fixed</code>
any	<code>CORBA::Any</code>

`CORBA::Fixed` and `CORBA::Any` are C++ classes. The remaining types map to C++ native types.

`WChar` and integer types may not be distinguishable for overloading.

`Boolean`, `Char`, and `Octet` may all use the same underlying character type.



IDL `char` and `wchar` map to `CORBA::Char` and `CORBA::WChar`, respectively.

`CORBA::Char` is a synonym for one of the three C++ character types (signed, unsigned, or plain `char`). Do not make assumptions in your code about whether characters are signed or unsigned—such assumptions result in non-portable code.

`CORBA::WChar` is a synonym for `wchar_t`. Be aware that in older, non-standard C++ compilers, `wchar_t` in turn is merely an alias for one of the integer types. This means that you cannot safely overload among `WChar` and the integer types if you are using such an older compiler. (With a standard C++ compiler, overloading is safe because then `wchar_t` is a type in its own right.)

IDL `string` maps to `char *` and IDL `wstring` maps to `CORBA::WChar *` (which is the same as `wchar_t *`). This mapping permits you to use standard library functions (such as `strcat`) without the need for conversions.

IDL `boolean` maps to either one of the C++ character types or, for a standard C++ compiler, to C++ `bool`.

IDL `octet` maps to one of the C++ character types.

IDL `fixed` maps to a C++ class that provides appropriate semantics and permits conversion among fixed-point, floating-point, and integer types.¹

IDL `any` maps to the C++ class `CORBA::Any`. We cover this class in Section 5.29.

1. Because type `fixed` is used little, we do not cover it here. See Henning & Vinoski for details.

Overloading on Built-In Types

- Do not overload functions solely on `CORBA::Char`, `CORBA::Boolean`, and `CORBA::Octet`. They may all use the same underlying type.

ORBacus maps `CORBA::Boolean` to `bool`, `CORBA::Char` to `char`, and `CORBA::Octet` to `unsigned char`.

- Do not overload functions solely on `CORBA::WChar` and one of the integer types. With older C++ compilers, `wchar_t` may be indistinguishable from an integer type for overloading.

If you are working exclusively with standard C++ compilers, `wchar_t` is a type in its own right and so does not cause problems.



5.6 Overloading on Built-In Types

If you need to port code among different ORBs (particularly if you also need to run in environments with older C++ compilers), avoid overloading among types that are permitted to map to the same underlying type. For example, the following may not compile in all environments:

```
void foo(CORBA::Char param)    { /* ... */ };
void foo(CORBA::Boolean param) { /* ... */ }; // !!!
void foo(CORBA::Octet param)  { /* ... */ }; // !!!
void foo(CORBA::Short param)  { /* ... */ };
void foo(CORBA::Long param)   { /* ... */ };
void foo(CORBA::WChar param)  { /* ... */ }; // !!!
```

For ORBacus, the above presents no problem because all built-in types map to different C++ types. However, the same may not be true for other ORBs.

Memory Allocation for Strings

For dynamic allocation of strings, you *must* use the provided functions:

```
namespace CORBA {
    // ...
    char * string_alloc(ULong len);
    char * string_dup(const char *);
    void string_free(char *);
    WChar * wstring_alloc(ULong len);
    WChar * wstring_dup(const WChar *);
    void wstring_free(WChar *);
    // ...
};
```

Calling `(w)string_alloc(n)` allocates `n+1` characters!

These functions are necessary for environments with non-uniform memory architectures (such as Windows).



5.7 Memory Allocation for Strings

To allocate and deallocate strings and wide strings, you *must* use the provided functions. Use of `new`, `new[]`, `delete`, or `delete[]` is non-portable (and will in fact cause a crash in some environments).

`string_alloc` allocates one more byte than was requested, in order to make room for the terminating NUL byte:

```
char * p = CORBA::string_alloc(5); // Allocates 6 bytes
strcpy(p, "Hello");                // OK, "Hello" fits
```

`string_dup` allocates and copies a string in a single step, so you can write the preceding as:

```
char * p = CORBA::string_dup("Hello");
```

`string_alloc` and `string_dup` return a null pointer if memory is exhausted. (They never throw an exception.)

You must use `string_free` to eventually deallocate a string returned from `string_alloc` or `string_dup`. Passing a null pointer to `string_free` is safe and does nothing.

The `wstring_*` functions for wide string have analogous behavior. (Note that `wstring_alloc` counts characters, not bytes.)

Mapping for Constants

Constants map to corresponding constant definitions in C++:

```
const long ANSWER = 42;
const string NAME = "Deep Thought";
```

This maps to:

```
const CORBA::Long ANSWER = 42;
const char * const NAME = "Deep Thought";
```

Global constants and constants that are nested in namespaces (IDL modules) are initialized in the header file.

Constants that are defined inside interfaces *may* be initialized in the header file if:

- they are of integral or enumerated type
- the target compiler complies with standard C++



5.8 Mapping for Constants

The mapping for constants is straight-forward: for every IDL constant, a corresponding constant definition is generated into the client-side header file.

A minor problem arises if a constant definition is nested inside an interface. In that case, the constant definition at the C++ level is nested inside a class (because interfaces map to classes and the mapping preserves the scoping of IDL identifiers). However, only integral constants and enumerated constants can be initialized inside a class (and then only with a standard C++ compiler):

```
interface I {
    const long ANSWER = 42;
    const string NAME = "Deep Thought";
};
```

The generated C++ code could look like this:

```
class I /* ... */ {
    static const CORBA::Long ANSWER = 42;
    static const char * const NAME; // "Deep Thought"
};
```

Note that the string constant cannot be initialized in the class header because C++ does not permit that.

Another legal mapping (which is used by ORBacus) is:

```
class I /* ... */ {
    static const CORBA::Long ANSWER;    // 42
    static const char * const NAME;      // "Deep Thought"
};
```

Note that neither constant is initialized in the header file. Instead, the appropriate initialization is generated into the source file.

Usually, this does not matter, except when you want to use a constant in a context where a compile-time constant expression is required, such as in a `switch` statement or an array definition:

```
char * wisdom_array[I::ANSWER];        // Compile-time error
```

This is no great hardship because you can use a dynamic allocation instead:

```
char ** wisdom_array = new char *[I::ANSWER];    // OK
```

In order to keep your code portable, you should not rely on initialization of constants that are nested inside interfaces, even for standard C++ environments.

Variable-Length Types

The following types are variable-length:

- strings and wide strings (bounded or unbounded)
- object references
- type **any**
- sequences (bounded or unbounded)

Structures, unions, and arrays can be fixed- or variable-length:

- They are fixed-length if they (recursively) contain only fixed-length members or elements.
- They are variable-length if they (recursively) contain variable-length members or elements.

Variable-length values require the sender to dynamically allocate the value and the receiver to deallocate it.



5.9 Variable-Length Types

The C++ mapping has the concept of fixed- and variable-length types. Most of the built-in types, such as `char` or `long` are fixed-length. Fixed-length types are easy to handle in the C++ mapping because their size is known at compile time. However, variable-length types require more thought because their length is undecided until run time. This means that, when values are passed from sender to receiver (or from caller to callee), they must be dynamically allocated by the sender and deallocated by the receiver once the values are no longer needed.

This memory management rule places the burden of deallocating a variable-length value on the receiver of the value. If the receiver forgets to deallocate a value, it suffers a memory leak. However, forcing the receiver of a value to deallocate it is the only viable option for CORBA. Other approaches all have serious drawbacks. For example, fixed-length buffers arbitrarily limit the size of values; retaining ownership of memory in the callee means that calls are not reentrant; and requiring the caller to preallocate memory means that repeated calls are necessary if a value is larger than the preallocated memory, and that is too expensive for distributed calls.

Example: String Allocation

The callee allocates the string and returns it to the caller:

```
char * getstring()
{
    return CORBA::string_dup(some_message); // Pass ownership
}
```

The caller takes ownership of the string and must deallocate it:

```
{
char * p = getstring(); // Caller becomes responsible
// Use p...
CORBA::string_free(p); // OK, caller deallocates
}
```

All variable-length types follow this basic pattern.

Whenever a variable-length value is passed from server to client, the server allocates, and the client must deallocate.



5.10 Example: String Allocation

The above code illustrates the general approach to passing variable-length values from the callee (the server) to the caller (the client). The server allocates and initializes the variable-length value and returns a pointer to the allocated memory; the client uses the returned value and deallocates it when it is no longer needed.

This allocation pattern is used for all variable-length values. It avoids arbitrary size limitations, is reentrant, and does not require repeated calls to return large values. The only drawback is that the caller must remember to deallocate the returned value; otherwise, memory is leaked.

_var Types

_var types are smart pointer classes you can use to make memory leaks unlikely.

A _var type is initialized with a pointer to dynamic memory. When a _var type goes out of scope, its destructor deallocates the memory.

```


class String_var {
public:
    String_var(char * p) { _ptr = p; }
    ~String_var() { CORBA::string_free(_ptr); }
    // etc...
private:
    char * _ptr;
};

```

▶

H | e | l | l | o | \0


The *only* purpose of _var types is to “catch” a dynamically-allocated value and deallocate it later. You need not (but should) use them.



12

Basic C++ Mapping

Copyright 2000–2001 IONA Technologies



5.11 _var Types

To make life with variable-length types easier, the C++ mapping provides what is known as _var types. For every variable-length and user-defined complex type, the C++ mapping creates a corresponding _var type. The use of _var types is completely optional. If you choose to ignore _var types, you simply must remember to explicitly deallocate everything that was dynamically allocated. If you choose to use _var types, you need not remember deallocation because, once initialized, the _var type deallocates memory when its destructor runs.

Given the String_var type above, we can rewrite the preceding code example as follows:

```

{
    CORBA::String_var sv(getstring());
    // Use sv...

} // No explicit deallocation required here.

```

No memory leak occurs in this code because the destructor of sv deallocates the memory sv took ownership of when it was initialized.

Note that this simple example may appear trivial. However, _var types substantially simplify memory management for real-life code (which is invariably more complex). In general, _var types exist for the same purpose as the standard C++ auto_ptr template:² they make memory leaks less likely (and also contribute to exception safety).

2. The C++ mapping does not use the standard C++ auto_ptr type because it must work with older C++ compilers. Note that _var types do *not* have exactly the same ownership semantics as auto_ptr.

C++ Mapping Levels

The IDL compiler generates a pair of types for every variable-length and user-defined complex type, resulting in a low- and high-level mapping:

IDL Type	C++ Type	C++ <code>_var</code> Type
<code>string</code>	<code>char *</code>	<code>CORBA::String_var</code>
<code>any</code>	<code>CORBA::Any</code>	<code>CORBA::Any_var</code>
<code>interface foo</code>	<code>foo_ptr</code>	<code>class foo_var</code>
<code>struct foo</code>	<code>struct foo</code>	<code>class foo_var</code>
<code>union foo</code>	<code>class foo</code>	<code>class foo_var</code>
<code>typedef sequence<X> foo;</code>	<code>class foo</code>	<code>class foo_var</code>
<code>typedef X foo[10];</code>	<code>typedef X foo[10];</code>	<code>class foo_var</code>

- The low level does not use `_var` types and you must deal with memory management explicitly.
- The high level provides `_var` types as a convenience layer to make memory management less error-prone.



5.12 C++ Mapping Levels

`_var` types act as a convenience layer over the basic (or low-level) C++ mapping. For each IDL type, the compiler creates a pair of types: an actual (low-level) type that represents the corresponding IDL type, and a `_var` type. The low-level type implements the semantics of the type. For example, for a type corresponding to an IDL union, the generated C++ class contains member functions that permit you to manipulate the value of a union. The union's `_var` type is a simple memory-management wrapper class that stores a pointer to its underlying union instance and deallocates the memory for the underlying union when the `_var` instance goes out of scope.

The 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;
    operator char * &();

    // ...
};

```



5.13 The string_var Class

As an example of a `_var` class, let us examine the class `CORBA::String_var` in some detail. This is useful because `_var` types for other IDL types (such as interfaces or structures) follow a similar pattern.³

`String_var()`

The default constructor initializes the `String_var` to contain a null pointer. Do not use a default-constructed `String_var` until after you have initialized it:

```

CORBA::String_var s;
cout << "s = \"\" << s << \"\" << endl; // Crash imminent!

```

`String_var(char *)`

The `char *` constructor initializes the `String_var` by taking ownership of its argument. It assumes that the passed string was allocated with `CORBA::string_alloc` or `CORBA::string_dup` because the destructor calls `CORBA::string_free`.

`String_var(const char *)`

If you construct a `String_var` using the `const char *` constructor, the `String_var` makes a deep copy of the string. When the `String_var` goes out of scope, it deallocates its copy of the string but leaves the original copy unaffected. For example:

3. The `String_var` class is the most complex of all `_var` types, so once you know `String_var`, the other `_var` types are easy to learn.

```

const char * message = "Hello";
// ...

{
    CORBA::String_var s(message);    // Makes a deep copy
    // ...
} // ~String_var() deallocates its own copy only.

cout << message << endl;           // OK

```

String_var(const String_var &)

The copy constructor makes a deep copy. If you initialize one `String_var` from another `String_var`, modifications to one copy do not affect the other copy.

~String_var()

The destructor calls `CORBA::string_free` to deallocate the string held by the `String_var`.

String_var & operator=(char *)

String_var & operator=(const char *)

String_var & operator=(const String_var &)

The assignment operators follow the conventions of the constructors. The `char *` assignment operator assumes that the string was allocated with `string_alloc` or `string_dup` and takes ownership of the string.

The `const char *` assignment operator and the `String_var` assignment operator each make a deep copy.

Before accepting the new string, the assignment operators first deallocate the current string held by the target. For example:

```

CORBA::String_var target;
target = CORBA::string_dup("Hello");    // target takes ownership

CORBA::String_var source;
source = CORBA::string_dup("World");    // source takes ownership

target = source;    // Deallocates "Hello" and takes
                   // ownership of deep copy of "World".

```

operator char *()

operator const char *()

operator char * &()

The conversion operators permit you to pass a `String_var` as if it were a `char *`. This makes use of `String_var` transparent. For example, you can print a `String_var` or pass it to library functions as if you were using the underlying `char *` directly:

```

CORBA::String_var s = CORBA::string_dup("Hello");
cout << "Length of \" << s << "\" is \" << strlen(s) << endl;

```

The String_var Class (cont.)

```

class String_var {
public:
    // ...

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

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

ostream & operator<<(ostream, const CORBA::String_var);
istream & operator>>(istream, CORBA::String_var &);

```



15
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



```

char & operator[] (ULong)
char operator[] (ULong) const

```

The overloaded subscript operators permit you to use an index to get at the individual characters of a `String_var` as if it were an array. For example:

```

CORBA::String_var s = CORBA::string_dup("Hello");
cout << s[4] << endl;

```

```

const char * in() const
char * & inout()
char * & out()

```

These operators are provided mainly to deal with compilers that have problems applying the C++ conversion rules correctly. For example, your compiler may (incorrectly) reject the following:

```

CORBA::String_var s = ...;
cout << strlen(s) << endl; // Bad compiler can't handle this...

```

In this case, you can call the `in` operator explicitly to force the correct conversion:

```

CORBA::String_var s = ...;
cout << strlen(s.in()) << endl; // Force explicit conversion

```

The `out` member function allows you to pass a `String_var` as an output parameter where a `char * &` is expected. For example, assume we are using a `read_string` helper function, defined as follows:

```

void read_string(char * & s)
{
    // Read a line of text from a file...
    s = CORBA::string_dup(line_of_text);
}

```

Without `_var` types, if you want to call `read_string` twice with the same argument, you must remember to deallocate in between the two calls:

```

char * s;
read_string(s);
cout << s << endl;
CORBA::string_free(s); // Must deallocate here!
read_string(s);
cout << s << endl;
CORBA::string_free(s); // Must deallocate here!

```

If you use the `out` member function on a `String_var` instead, no deallocation is necessary:

```

CORBA::String_var s;
read_string(s.out());
cout << s << endl;
read_string(s.out()); // No leak here.
cout << s << endl;

```

The memory leak is avoided because the `out` member function first deallocates whatever string is currently held by a `String_var` and returns a *reference* to a null pointer; this means that the initial argument passed to `read_string` is a reference to a null pointer and `read_string` then assigns the new pointer value via that reference.

char * _retn()

This member function returns the pointer currently held by a `String_var` and sets the pointer inside the `String_var` to null. The net effect is that the call, after returning the pointer value, passes ownership for the string memory from the `String_var` to the caller. This is particularly useful for exception safety. (See page 6-38 for details.)

ostream & operator<<(ostream, const CORBA::String_var)

istream & operator>>(istream, CORBA::String_var &)

The stream insertion and extraction operators permit you to insert and extract a `String_var` as if it were a normal pointer. Note that, for extraction, the operator by default terminates extraction with the next white space character (just as it does for `char *`).

String_var: Summary

Keep the following rules in mind when using `String_var`:

- Always initialize a `String_var` with a dynamically-allocated string or a `const char *`.
- Assignment or construction from a `const char *` makes a deep copy.
- Assignment or construction from a `char *` transfers ownership.
- Assignment or construction from a `String_var` makes a deep copy.
- Assignment of a `String_var` to a pointer makes a shallow copy.
- The destructor of a `String_var` deallocates memory for the string
- Be careful when using string literals with `String_var`.



16
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.14 Main Rules for Using `String_var`

The above slide summarizes the rules for using `String_var` correctly.

Always initialize a `String_var` with a dynamically-allocated string.

Do not use an uninitialized `String_var` or initialize a `String_var` with a string that is not allocated with `CORBA::string_alloc` or `CORBA::string_dup`:

```
{
    CORBA::String_var s1;
    cout << s1 << endl;           // Bad news!

    char message[] = "Hello";
    CORBA::String_var s2 = message; // Bad news!

    CORBA::String_var s3 = strdup("Hello"); // Bad news!
}
```

Assignment or construction from a `const char *` makes a deep copy.

Assignment and construction from a `const char *` makes a deep copy:

```

const char message[] = "Hello";
{
    CORBA::String_var s = message; // OK, deep copy
}
cout << message << endl;          // Fine

```

Assigning or constructing from a char * transfers ownership.

If you assign a char * to a String_var, the String_var takes ownership. This means that the string must initially have been allocated with string_alloc or string_dup and that you must not explicitly deallocate the string after the assignment:

```

{
    char * p = CORBA::string_dup("Hello");
    CORBA::String_var s = p; // s takes ownership
    // Do not deallocate p here!
    // ...
} // OK, s deallocates the string

```

Copying or assigning a String_var makes a deep copy.

If you assign one String_var to another, or use the copy constructor, you get a deep copy:

```

String_var s1 = CORBA::string_dup("Hello");
String_var s2 = s1;
cout << s1 << endl; // Prints "Hello"
cout << s2 << endl; // Prints "Hello"
s1[0] = 'h';
s1[4] = '0';
cout << s1 << endl; // Prints "hello"
cout << s2 << endl; // Prints "Hello"

```

Assigning a String_var to pointer makes a shallow copy.

There is nothing wrong with assigning a String_var to a char * or const char *. However, keep in mind that the String_var retains ownership of the string in this case, so you must not dereference the pointer after the String_var goes out of scope:

```

char * p;
{
    CORBA::String_var s = CORBA::string_dup("Hello");
    p = s; // Fine, p points at memory owned by s
}
cout << p << endl; // Disaster!

```

The destructor of a String_var deallocates memory for the string.

The destructor of a String_var calls CORBA::string_free. Obviously, this means that you must initialize a String_var with memory that was allocated by CORBA::string_alloc or CORBA::string_dup. However, it also means that you must not give ownership of the same string to two different String_var instances:

```
char * p = CORBA::string_dup("Hello");
char * q = p;    // Both p and q point at the same string

CORBA::String_var s1 = p;    // Fine, s1 takes ownership
// ...
CORBA::String_var s2 = q;    // Very bad news indeed!
```

Be careful when using string literals with `String_var`.

As we saw, assignment or construction from a `char *` causes a `String_var` to take ownership of the passed pointer, whereas assignment or construction from a `const char *` makes a deep copy. This causes problems with older compilers that do not adhere to the C++ standard:

```
CORBA::String_var s = "Hello";    // No problem with standard C++,
                                   // but a complete disaster with
                                   // older compilers!
```

This works fine with standard C++ compilers for which the type of a string literal is `const char *`, causing the `String_var` to make its own copy of the string. However, for older compilers, it spells disaster: in classic C++ (as in C), the type of a string literal is `char *`. As a result, the `String_var` takes ownership of the string and eventually calls `string_free` on it. This results in an attempt to deallocate non-heap memory and, in most cases, will cause a core dump.

If you have to use the same code base with both older and newer compilers, one option is to cast the string literal to `const char *`:

```
CORBA::String_var s = (const char *)"Hello";    // Fine
```

This works for both older and standard C++ compilers and forces a deep copy. However, we suggest to avoid casts wherever possible and use an explicit call to `string_dup` instead:

```
CORBA::String_var s = CORBA::string_dup("Hello");    // Fine too
```

Both approaches correctly result in a deep copy but, as a matter of style, the second approach is preferable because it makes it explicit that an allocation takes place, whereas the cast suggests something more serious. In addition, the explicit copy will not work if you accidentally supply an argument of the wrong type, whereas the cast will quite happily let you turn, for example, a floating-point variable into a pointer.

Mapping for Fixed-length Structures

IDL structures map to C++ classes with public members. For example:

```
struct Details {
    double      weight;
    unsigned long count;
};
```

This maps to:

```
struct Details {
    CORBA::double  weight;
    CORBA::ULong   count;
};
```

The generated structure may have additional member functions. If so, they are internal to the mapping and you must not use them.



5.15 Mapping for Fixed-Length Structures

IDL fixed-length structures map to C++ structures with public data members that correspond to the members. This means that you can access structure members as you would for any other C++ struct:

```
Details d;
d.weight = 8.5;
d.count = 12;
```

If the structure only contains trivial members (that is, is a C++ aggregate), you can initialize instances statically:

```
Details d = { 8.5, 12 };
```

(Some compilers still have problems with initialization of automatic aggregates, so you should use this style with caution.)

The generated structure may contain member functions that are internal to the mapping. For example, structures frequently contain a class-specific `operator new` and `operator delete`. If such member functions exist, they are specific to your ORB and you must not call them directly (because doing so would result in non-portable code).

If you need to dynamically allocate and deallocate fixed-length structures, use `new` and `delete`, as usual. If your platform has special memory-management requirements, these will be taken care of by class-specific members, so you need not use a special-purpose allocation function for structures.

Mapping for Variable-Length Structures

Variable-length structures map to C++ classes with public data members. Members of variable-length type manage their own memory.

```
struct Fraction {
    double numeric;
    string alphabetic;
};
```

This maps to:

```
struct Fraction {
    CORBA::Double      numeric;
    OB::StrForStruct   alphabetic; // vendor-specific
};
```

String members behave like a `String_var` that is initialized to the empty string.

Never use internal types, such as `StrForStruct`, in your code!



5.16 Mapping for Variable-Length Structures

The mapping for a structure containing a string (and therefore of variable length) is shown above. As for fixed-length structures, the generated structure may contain additional member functions. If so, you must ignore them and not call such member functions directly.

The type of the string member (`OB::StrForStruct` in this case) is not defined by the C++ mapping. The specific type name does not matter; all you need to know is that string members in user-defined types (such as structures, exceptions, sequences, or arrays) behave like a `String_var`, that is, they are smart pointers. However, nested strings differ in one respect from a normal `String_var`: they are initialized to the empty string by their default constructor. This is different from `String_var`, which is default-constructed to contain a null pointer.⁴

NOTE: In general, the C++ mapping uses self-managed types for the members of user-defined complex types. This means that you do not have to explicitly deallocate memory for every nested variable-length member. Instead, as soon as the outermost (containing) instance of a type is destroyed, the memory for all the contained members is released automatically.

The generated structure is easy to use if you keep in mind that string members behave like a `String_var`. For example:

4. This inconsistency is a historical glitch in the C++ mapping and we simply have to live with this wrinkle.

```

{
    Fraction f;
    f.numeric = 1.0/3.0;
    f.alphabetic = CORBA::string_dup("one third");
} // No memory leak here

```

When `f` goes out of scope, the destructor of each member is invoked, so the string assigned to `f.alphabetic` is correctly deallocated.

NOTE: You cannot statically initialize a variable-length structure because it will always contain at least one member with a user-defined constructor. That is, variable-length structures never are C++ aggregates.

You can treat variable-length structures must like any other variable in your program. Assignment and copy provide the appropriate deep-copy semantics:

```

{
    struct Fraction f1, f2, f3;

    f1.numeric = .5;
    f1.alphabetic = CORBA::string_dup("one half");
    f2.numeric = .25;
    f2.alphabetic = CORBA::string_dup("one quarter");
    f3.numeric = .125;
    f3.alphabetic = CORBA::string_dup("one eighth");

    f2 = f1; // Deep assignment
    f3.alphabetic = f1.alphabetic; // Deep assignment
    f3.numeric = 1.0;
    f3.alphabetic[3] = '\\0'; // Does not affect f1 or f2
    f1.alphabetic[0] = 'O'; // Does not affect f2 or f3
    f1.alphabetic[4] = 'H'; // Does not affect f2 or f3
} // Everything deallocated OK here

```

The following shows the before and after state of the three structures:

		f1	f2	f3
Before	numeric	0.5	0.25	0.125
	alphabetic	one half	one quarter	one eighth
.....				
After	numeric	0.5	0.5	1.0
	alphabetic	One Half	one half	one

NOTE: If a sequence contains a recursive member (as shown on page 2-28), the type name of the member is `<struct_name>::_<member_name>_seq`. For example, the type of the `children` member on page 2-28 is `Node::_children_seq`.

Mapping for Unbounded Sequences

Each IDL sequence type maps to a distinct C++ class.

An unbounded sequence grows and shrinks at the tail (like variable-length vectors).

A `length` accessor function returns the number of elements.

A `length` modifier function permits changing the number of elements.

Sequences provide an overloaded subscript operator (`[]`).

Access to sequence elements is via the subscript operator with indexes from 0 to `length() - 1`.

You cannot grow a sequence by using the subscript operator. Instead, you must explicitly increase the sequence length using the `length` modifier.

Accessing elements beyond the current length is illegal.



19
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.17 Mapping for Unbounded Sequences

Every named sequence type in IDL results in a C++ class of the same name. (Anonymous sequences are mapped to classes with a name that is internal to the mapping, which is why you should avoid them.) The generated class has overloaded `length` member functions that permit you to grow and shrink the sequence at the tail, as well as overloaded subscript operators that permit you to get and set sequence elements. You can grow an unbounded sequence to any number of elements (subject to memory limitations).

In order to add new sequence elements, you must first create them by increasing the length of the sequence and then initialize the newly-added elements; accessing a sequence element beyond the current length of a sequence results in undefined behavior.

No special memory allocation functions are required for sequences. If you want to use dynamically allocated sequences, use `new` and `delete` to allocate and deallocate them. (The sequence class will contain class-specific `operator new` and `operator delete` members for operating systems with non-uniform memory management.)

Mapping for Unbounded Sequences (cont.)

```
typedef sequence<string> StrSeq;
```

This maps to:

```
class StrSeq {
public:
    StrSeq();
    StrSeq(CORBA::ULong max);
    StrSeq(const StrSeq &);
    ~StrSeq();

    StrSeq & operator=(const StrSeq &);

    CORBA::ULong length() const;
    void length(CORBA::ULong newlen);
    CORBA::ULong maximum();

    OB::StrForSeq operator[](CORBA::ULong idx);
    const char * operator[](CORBA::ULong idx) const;
    // ...
};
```



20
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



StrSeq()

The default constructor creates an empty sequence, so calling the `length` accessor of a default-constructed sequence returns zero.

StrSeq(CORBA::ULong)

The constructor permits you provide a hint to the sequence implementation as to the maximum number of elements you expect you will be placing onto the sequence. The sequence can use this information to chunk memory allocations more efficiently. Using this constructor has no visible effect otherwise. In particular, the sequence is still created with zero elements and you can add more elements to the sequence than the maximum provided to this constructor.⁵

StrSeq(const StrSeq &)

```
StrSeq & operator=(const StrSeq &)
```

The copy constructor and assignment operator make deep copies. (The assignment operator releases storage for the original sequence first, of course.)

~StrSeq()

The destructor destroys a sequence. If the sequence contains variable-length elements, dynamic memory for the elements is also released.

5. The sequence maximum is similar to the notion of capacity in some of the standard C++ library classes. While the actual number of elements remains below the maximum, no reallocation of the sequence memory will occur.

CORBA::ULong length() const

The `length` accessor returns the number of elements of the sequence.

void length(CORBA::ULong)

The `length` modifier changes the length of a sequence:

- Increasing the length of a sequence creates new elements at the tail. Growing a sequence initializes the new elements using their default constructor. (In this example, the sequence elements are strings, so the new elements are initialized to the empty string.)
- Decreasing the length of a sequence truncates the sequence by destroying elements at the tail. The truncated elements are permanently destroyed. This means that, if you shrink a sequence and grow it again, you cannot expect the previously truncated elements to still be there. (Those elements will have been destroyed and then recreated using their default constructor.)

CORBA::Long maximum()

The `maximum` accessor returns the current maximum of the sequence. The sequence guarantees that elements will not be relocated in memory as long as the actual length remains below the maximum.

OB::StrForSeq operator[] (CORBA::ULong)**const char * operator[] (CORBA::ULong) const**

The subscript operators provide access to the sequence elements. Note that, in this example, the return type of the non-constant operator is `OB::StrForSeq`. This type name is internal to the mapping and you will never need to use it directly. For all intents and purposes, you can pretend that the element type is `String_var` (with the exception that nested strings, as always, are initialized to the empty string instead of a null pointer).

Of course, the return type of these operators depends on the sequence element type. In general, for a sequence containing elements of type `T`, these operators return values of type `T &` and type `T`, respectively. (For example, for a sequence of `CORBA::Double`, the operators return `CORBA::Double &` and `CORBA::Double`.)⁶

Sequences are indexed from 0 to `length() - 1`. Do not attempt to index a sequence beyond its current length; doing so results in undefined behavior (most likely, a core dump).

6. For complex types, you may find that the actual type returned for the non-constant subscript operator is not a C++ reference. However, in that case, the actual type that is returned will behave as if it were a C++ reference.

Example: Using a String Sequence

```

StrSeq s(5); // Maximum constructor
assert(s.length() == 0); // Sequences start off empty

s.length(4); // Create four empty strings
assert(s[0] && *s[0] == '\0'); // New strings are empty

for (CORBA::ULong i = 0; i < 4; ++i)
    s[i] = CORBA::string_dup(argv[i]); // Assume argv has four elmts

s.length(2); // Lop off last two elements
assert(s.length() == 2);

for (CORBA::ULong i = 2; i < 8; ++i) { // Assume argv has eight elmts
    s.length(i + 1); // Grow by one element
    s[i] = CORBA::string_dup(argv[i]); // Last three iterations may
    // cause reallocation
}
for (CORBA::ULong i = 0; i < 8; ++i)
    cout << s[i] << endl; // Show elements

```



5.18 Example: Using a String Sequence

The above code sequence shows an example of using a string sequence. The constructor for the sequence `s` uses the maximum constructor. This guarantees that the sequence will not be relocated in memory while it has five or fewer elements. Note that this guarantee does not extend to the sequence elements if the elements are variable-length (as they are in this example).⁷

Immediately after construction, the sequence is empty (has zero elements).

The next step adds four sequence elements and verifies that the first element was initialized correctly by its default constructor, before adding four new strings to the sequence. Note that string sequence elements take ownership for the string, so we use `CORBA::string_dup` for the assignment.

The next step shortens the sequence by two and then proceeds to add six more elements. Note that for the second loop, we increment the length by one inside the loop instead of setting it in advance. This second option works fine, but is likely to be less efficient because of additional memory reallocations. Note that the second loop exceeds the maximum for the sequence. There is no problem with this—the sequence grows its maximum as needed. (Depending on the sequence implementation, the maximum may increase in fairly large steps instead of incrementing by one whenever the sequence is extended by one element.)

The final step prints the sequence contents. This illustrates that the sequence elements transparently can be used as if they were of type `char *` or `const char *`.

7. The relocation guarantee is useful if you point at sequence elements. However, seeing that doing so is dangerous, we suggest that you do not use pointers or references to sequence elements. As a rule, use the maximum constructor if you know in advance how many elements you will use. It will avoid unnecessary relocations and so be faster. Otherwise, don't bother with the maximum constructor.

Using Complex Element Types

If a sequence contains complex elements, such as structures, the usual deep copy semantics apply:

- Assignment or copying of sequences makes a deep copy.
- Assignment or copying of sequence elements makes a deep copy.
- Extending a sequence constructs the elements using their default constructor.
- Truncating a sequence (recursively) releases memory for the truncated elements.
- Destroying a sequence (recursively) releases memory for the sequence elements and the sequence.



5.19 Using Complex Element Types

If a sequence contains elements of complex type, such as a structure or another sequence, you get deep copy semantics for assignment and copying. Here is an example that uses a sequence of structures to illustrate this:

```
struct Fraction {
    double numeric;
    string alphabetic;
};
typedef sequence<Fraction> FractSeq;
```

You can freely assign sequences and sequence elements to each other; the correct memory management activities are taken care of automatically:

```
FractSeq fs1;
fs1.length(1);
fs1[0].numeric = 1.0;
fs1[0].alphabetic = CORBA::string_dup("One");
FractSeq fs2 = fs1; // Deep copy
assert(fs2.length() == 1);
fs2.length(2);
fs2[1] = fs1[0]; // Deep copy
```

This results in `fs1` containing the single element `{1.0, "One"}`, and `fs2` containing two elements, `{1.0, "One"}` and `{1.0, "One"}`.

Mapping for Bounded Sequences

Bounded sequences have a hard-wired maximum:

```
typedef sequence<string,5> StrSeq;
```

This maps to:

```
class StrSeq {
public:
    StrSeq();
    StrSeq(const StrSeq &);
    ~StrSeq();
    StrSeq & operator=(const StrSeq &);
    CORBA::ULong length() const;
    void length(CORBA::ULong newlen);
    CORBA::ULong maximum();
    OB::StrForSeq operator[](CORBA::ULong idx);
    const char * operator[](CORBA::ULong idx) const;
    // ...
};
```



23
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.20 Mapping for Bounded Sequences

The mapping for bounded sequences is identical, except that the maximum constructor is absent and the maximum is instead hard-wired into the generated class and the maximum accessor always returns the sequence bound.

Do not attempt to increase the length of a bounded sequence beyond its bound. If you do, you will suffer undefined behavior (most likely, a core dump).

NOTE: Both bounded and unbounded sequences have additional member functions that permit you to directly manipulate the buffer that underlies a sequence. Use of this feature is useful only for octet sequences (which are often use to transmit binary data). Because the feature is rarely used and the API fairly complex, we do not show it here. (See Henning & Vinoski for details.)

Rules for Safe Use of Sequences

Keep the following in mind when using sequences:

- Never point at sequence elements or keep references to them.
If the sequence relocates in memory, the pointers or references will dangle.
- Never subscript beyond the current length.
The behavior is undefined if you read or write an element beyond the current length. Most likely, you will corrupt memory.
- Do not assume that sequence elements are adjacent in memory.
Never perform pointer arithmetic on pointers to sequence elements. The results are undefined.



24
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.21 Rules for Safe Use of Sequences

The above slide summarizes the rules for safe use of sequences. You should avoid keeping pointers or references to sequence elements. If the sequence is relocated in memory, the pointers will dangle. (Of course, you can use the sequence maximum to ensure that this won't happen but the trouble is probably not worth it.) Subscripting beyond the current length has undefined behavior. Do not use pointer arithmetic on pointers to sequence elements. The results are meaningless because sequence elements may not be in adjacent memory locations.

Mapping for Arrays

IDL arrays map to C++ arrays. For example:

```
typedef string  NameList[10];
typedef long   ScoreTable[3][2];
```

This maps to:

```
typedef OB::StrForStruct  NameList[10];
typedef OB::StrForStruct  NameList_slice;

typedef CORBA::Long      ScoreTable[3][2];
typedef CORBA::Long      ScoreTable_slice[2];
```

The slice type of an array is the element type of an array or, for a multi-dimensional array, the element type of the outermost dimension.

This means that an `<array>_slice *` is of type “pointer to element”.



5.22 Mapping for Arrays

IDL arrays map to C++ arrays of the same name. This means that you can take advantage of the guarantees provided by C++ for arrays. (Array elements are contiguous in memory and you can use pointer arithmetic.)

For each array, the compiler also generates a type `<array>_slice`. An array slice is the element type of an array. For one-dimensional arrays, that means the slice type is the same as the element type. For two-dimensional arrays, the slice type is the row type of the array (which is another array type). This means that a pointer to an array slice is a pointer to the first element of the array.

If an array has string elements, the elements behave like a `String_var`, that is, manage their own memory.

The code for arrays looks the same as that for any other array (but you must remember to deal with string members correctly). For example:

```
NameList nl; // Ten empty strings
for (int i = 0; i < 10 && i < argc; ++i)
    nl[i] = CORBA::string_dup(argv[i]);
nl[0] = nl[1]; // Deep copy

ScoreTable st; // Six undefined scores
st[0][0] = 99; // Initialize one score
```


Array Assignment and Allocation

For each array, the compiler generates functions to allocate, allocate and copy, deallocate, and assign arrays:

```
NameList_slice *   NameList_alloc();
NameList_slice *   NameList_dup(const NameList_slice *);
void               NameList_free(NameList_slice *);
void               NameList_copy(
                   const NameList_slice *   from,
                   NameList_slice *        to
                   );

ScoreTable_slice * ScoreTable_alloc();
ScoreTable_slice * ScoreTable_dup(const ScoreTable_slice *);
void               ScoreTable_free(ScoreTable_slice *);
void               ScoreTable_copy(
                   const ScoreTable_slice *   from,
                   ScoreTable_slice *        to
                   );
```



26
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.23 Array Assignment and Allocation

The compiler generates additional functions to deal with dynamically allocated arrays. As for strings, these special-purpose functions are provided to deal with non-uniform memory-management architectures. As for the memory-management functions for strings, you must use these functions instead of `new []` and `delete []`.

```
<array>_slice * <array>_alloc()
```

This function allocates a new array and returns a pointer to the first element. If allocation fails, the function returns null. You must eventually deallocate the array by passing the return value to `<array>_free`.

```
<array>_slice * <array>_dup(const <array>_slice *)
```

This function allocates a new array and initializes it with the elements in the source array (using deep copy semantics). If allocation fails, the function returns null. You must eventually deallocate the array by passing the return value to `<array>_free`.

```
void <array>_free(<array>_slice *)
```

This function deallocates an array previously allocated with `<array>_alloc` or `<array>_dup`. Deallocation of a null pointer is safe and does nothing.

Here is a brief example that illustrates how to use these functions:

```
// Allocate and initialize an array
//
NameList_slice * nlp = NameList_alloc();
for (int i = 0; i < sizeof(nlp) / sizeof(*nlp); ++i)
    nlp[i] = CORBA::string_dup("some name");

// Create copy of nlp
//
NameList_slice * nlp2 = NameList_dup(nlp);

// Clean up
//
NameList_free(nlp);
NameList_free(nlp2);
```

Due to the weak array semantics of C++, be careful if you have arrays of different dimensions but the same element type:

```
typedef string TwoNames[2];
typedef string FiveNames[5];
```

If you pass an array of `FiveNames` where an array of `TwoNames` is expected or vice versa, the results will be disastrous:

```
FiveNames fn;
// Initialize fn...
TwoNames_slice * tnp = FiveNames_dup(fn);    // Bad news!
```

Your best defense against such errors are diligence and a memory management debugging tool.

`void <array>_copy(const <array>_slice * from, <array>_slice * to)`

Because arrays are mapped to C++ arrays instead of classes, the mapping cannot provide an assignment operator for arrays. Instead, it generates an `<array>_copy` function, which performs a deep assignment:

```
FiveNames first_five, last_five;
// Initialize...

// The last will be the first...
FiveNames_copy(last_five, first_five);
```

Note that the `<array>_copy` function does not allocate a new array. Instead, it copies the contents of an existing array into another existing array. (Note that, in general, `<array>_copy` will be faster than copying elements in a loop because the compiler will use a memory block copy where possible.)

Mapping for Unions

IDL unions map to C++ classes of the same name:

- For each union member, the class has a modifier and accessor function with the name of the member.
- If a union member is of complex type, a third overloaded member function permits in-place modification of the active member.
- Every union has an overloaded member function `_d` which is used to get and set the discriminator value.
- The default constructor of a union performs no application-visible initialization.
- You activate a union member *only* by initializing it with its modifier function.



27
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.24 Mapping for Unions

Unions are mapped to classes.⁸ Consider the following IDL union:

```
union U switch (char) {
case 'L':
    long    long_mem;
case 'c':
case 'C':
    char    char_mem;
default:
    string  string_mem;
};
```

The union maps to a class that supports the usual operations (default construction, (deep) copy construction, and (deep) assignment). In addition, the class contains an accessor and a modifier function for each union member, as well as an accessor and modifier for the discriminator. (As with other complex types, the class may contain additional members that are internal to the mapping. You should ignore such members and not call them directly.)

8. They cannot be mapped to C++ unions because the union members may have constructors, but C++ does not permit a union member to have a constructor. In addition, C++ unions are not discriminated.


```

class U {
public:
    U();
    U(const U &);
    ~U();
    U & operator=(const U &);

    CORBA::Char    _d();
    void           _d(CORBA::Char);

    CORBA::Long    long_mem() const;
    void           long_mem(CORBA::Long);
    CORBA::Char    char_mem() const;
    void           char_mem(CORBA::Char);
    const char *   string_mem() const;
    void           string_mem(char *);
    void           string_mem(const char *);
    void           string_mem(const CORBA::String_var &);
};

```

U()

The default-constructor does not perform any visible initialization of the class. You must not access any part of a union until after you have initialized it.

U(const U &)**U & operator=(const U &)**

The copy constructor and assignment operator perform deep copy and deep assignment, as usual.

~U()

The destructor destroys the union and (recursively) deallocates memory for the union member.

CORBA::Char _d()**void _d(CORBA::Char)**

The overloaded `_d` member permits you to read and (with some limitations) set the discriminator value (see page 5-40). Note that the return and parameter type of `_d` depend on the discriminator type of the union. In this example, the union has a discriminator type of `char`, so `_d` uses `CORBA::Char`. (If we had used, for example, `long` as the discriminator, `_d` would use `CORBA::Long`.)

Accessors and Modifiers

For each union member, the compiler generates accessors and modifiers. The accessor returns the corresponding member's value. You must *not* call the accessor for a member unless that member is active. (Doing so has undefined behavior.)

The modifier for a member activates that member (if not active already), sets it to the supplied value, and, as a side effect, sets the discriminator value of the union to be consistent with the active member.

An example will serve to illustrate this.

```

U my_u;                // my_u is not initialized
my_u.long_mem(99);    // Activate long_mem
assert(my_u._d() == 'L'); // Verify discriminator
assert(my_u.long_mem() == 99); // Verify value

```

The call to `long_mem` in this example achieves two things: it activates the member `long_mem` and, as a side effect, sets the discriminator value to `'L'`, as illustrated by the assertions.

Calling a different modifier deactivates the currently active member and activates the new member. So, continuing this example:

```

// Deactivate long_mem, activate char_mem
//
my_u.char_mem('X');
assert(my_u.char_mem() == 'X');

// The discriminator is now either 'C' or 'c',
// but we don't know which...
//
assert(my_u._d() == 'c' || my_u._d() == 'C');

my_u._d('C'); // Now the discriminator is definitely 'C'

```

The call to `char_mem` deactivates `long_mem` and activates `char_mem`. Again, activating the new member sets the discriminator as a side effect. However, for `char_mem`, the union has two legal discriminator values, `'C'` and `'c'`. The C++ mapping guarantees that one of these two values will be used, but does not define which one, so the behavior is implementation-dependent.

NOTE: For this reason, we recommend that you avoid unions with more than one case label per member. Multiple case labels per member result in the above ambiguity and make it harder to write correct code, particularly when testing for the value of the discriminator.

You can use `_d` to assign the value of the discriminator explicitly, as shown in the preceding example. However, setting the discriminator is legal only if the new value is consistent with the currently active member. In other words, you cannot use `_d` to activate a member for a union that has no active member, and you cannot use `_d` to change the currently active member. Attempts to do so have undefined behavior:

```

my_u.char_mem('Z'); // Activate/assign char_mem
assert(my_u._d() == 'c' || my_u._d() == 'C');

my_u._d('C'); // OK
my_u._d('c'); // OK too, doesn't change active member
my_u._d('X'); // Undefined behavior, would activate string_mem

```

Activating the default member of a union leaves the discriminator in a partially defined state:

```

// Activate string_mem
//
my_u.string_mem(CORBA::string_dup("Hello"));

```

```
// Discriminator value is now anything except 'c', 'C', or 'L'
//
assert(my_u._d() != 'c' && my_u._d() != 'C' && my_u._d() != 'L');

// Now the discriminator has the value 'A'
//
my_u._d('A'); // OK, consistent with active member
```

NOTE: The default label suffers from the same problem as multiple case labels per member. However, in this case, the discriminator value is even less defined and could be a non-printing character, such as a Ctrl-S. This can be inconvenient during debugging and tracing, so we suggest that you avoid use of the default label.

Note that the preceding example also illustrates that string members of unions behave like a `String_var`, that is, they assume ownership of the assigned string. Conversely, when you read a string member, you assume responsibility for deallocation, because read access makes a deep copy:

```
if (my_u._d() != 'c' && my_u._d() != 'C' && my_u._d() != 'L') {
    // string_mem is active
    CORBA::String_var s = my_u.string_mem();
    cout << "member is " << s << endl;
} // s will deallocate the string
```

Mapping for Unions (cont.)

It is easiest to use a `switch` statement to access the correct member:

```
switch (my_u._d()) {
  case 'L':
    cout << "long_mem: " << my_u.long_mem() << endl;
    break;
  case 'c':
  case 'C':
    cout << "char_mem: " << my_u.char_mem() << endl;
    break;
  default:
    cout << "string_mem: "
         << my_u.string_mem() << endl;
    break;
}
```



28
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



To access a union, typically the easiest way is to switch on the discriminator value to find out which member is active.

NOTE: *Never* attempt to read a union member that is not active. Doing so has undefined behavior and will most certainly result in a core dump sooner or later. Unions are not meant to be used as a back-door mechanism for type casts!

Mapping for Unions (cont.)

A union without a `default` label has an extra member function called `_default`:

```
union AgeOpt switch (boolean) {
case TRUE:
    unsigned short age;
};
```

The generated class contains:

```
class AgeOpt {
public:
    // ...
    void _default();    // Sets discriminator to FALSE
};
```

`_default` picks a discriminator value that is not used by any of the explicit case labels of the union.



If a union does not have a `default` label, the compiler generates an additional member function, called `_default`.⁹ The `_default` member function leaves the union without an active member and a discriminator value that is not used by any of the explicit case labels.

The function is needed to initialize a union to contain no active member:

```
AgeOpt age; // Nothing is initialized
age._default(); // Sets discriminator to FALSE
assert(age._d() == 0);
```

Note that the following attempt to achieve the same thing is illegal:

```
AgeOpt age; // Nothing is initialized
age._d(0); // Illegal!
```

This does not work because we cannot initialize a union by setting the discriminator.

If more than one discriminator value is available to indicate the “no member” case, `_default` picks a discriminator value. If you care about the precise value, you must first call `_default` and then `_d` to set the value exactly.

9. It is a little unfortunate that a union *without* a `default` case has an extra member function called `_default`. The function would have better been called `_deactivate`.

Mapping for Unions (cont.)

Unions with members that are sequences, structures, unions, a fixed-point type or of type **any** contain a referent function:

```
typedef sequence<long> LongSeq;
union U switch (long) {
case 0:
    LongSeq ls;
};
```

The generated C++ contains:

```
class U {
public:
    const LongSeq & ls() const;           // Accessor
    void ls(const LongSeq &);           // Modifier
    LongSeq & ls();                       // Referent

    // Other member functions here...
};
```



If a union contains a member of complex type (a sequence, structure, union, fixed-point, or any type), an additional referent member function is generated by the compiler. The referent function is provided for efficiency reasons. The accessor and modifier functions make deep copies. This is convenient but expensive if union members are large (because of the cost of copying the member).

The referent function permits you to manipulate a union member in place without additional data copies. For example:

```
LongSeq ls;                               // Empty sequence
U my_u;                                   // Uninitialized union
my_u.ls(ls);                              // Activate sequence member
LongSeq & lsr = my_u.ls();                // Get reference to sequence member
lsr.length(max);                          // Create max elements

// Fill the sequence inside the union,
// instead of filling the sequence first
// and then having to copy it into the
// union member.
//
for (int i = 0; i < max; ++i)
    lsr[i] = i;
```

Using Unions Safely

A few rules for using unions safely:

- Avoid multiple **case** labels for a single member.
- Avoid the **default** label.
- Never access a union member that is inconsistent with the discriminator value.
- Only set the discriminator value if a member is already active and only set it to a value that is consistent with that member.
- To deactivate all members, use **_default**.
- Do not assume that union members will overlay each other memory.
- Members are activated by their copy constructor.
- Do not rely on side effects from the destructor.



31
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.25 Using Unions Safely

The above slide summarizes how to use unions safely.

Note that the C++ mapping does not guarantee that union members will overlay each other in memory.¹⁰

Note that, for efficiency reasons, members are activated by their copy constructor, rather than being initialized by the default constructor followed by an assignment.

Do not rely on any side effects from destructor calls (which is rather difficult to do anyway). This is because the union implementation may delay destructor calls and internally keep several members active at a time.

¹⁰.ORBacus does overlay members in memory. If members have constructors, they are heap-allocated and only one member exists at a time. Other ORBs may use a different strategy.

Mapping for typedef

IDL **typedef** maps to a corresponding C++ **typedef**.

Note that aliases are preserved:

```
typedef short   TempType;
typedef string  LocType;
typedef LocType LocationType;
```

The corresponding C++ is:

```
typedef CORBA::Short           TempType;

typedef char *                  LocType;
typedef CORBA::String_var      LocType_var;

typedef LocType                 LocationType;
typedef LocType_var             LocationType_var;
```



5.26 Mapping for typedef

IDL **typedef** simply maps to a corresponding C++ **typedef**. If a single IDL type results in more than one C++ type, the compiler generates a typedef for each C++ type. For example, the single `LocType` definition above results in two C++ definitions, one for `LocType` and one for `LocType_var`. If an IDL definition results in a C++ function definition (such as `<array>_alloc`), the compiler generates a function definition for the alias type name as well.

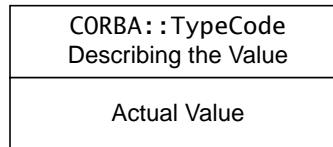
C++ **typedef** does not create a new type but only an alternate name for an existing type. This means that IDL **typedef** has no effect on the C++ mapping (other than to provide syntactic convenience).¹¹ As a matter of style, you should still use the proper type names in your code. This not only makes the code easier to understand, but also avoids problems if, for example, the definition of a type changes during development.

¹¹ The same need not be true in other languages, in which IDL **typedef** may be mapped to incompatible types.

Type any: Concepts

A value of type **any** contains a pair of values internally:

- a **TypeCode** that describes the type of the value in the **any**
- the actual value



The **TypeCode** inside an **any** is used to enforce type safety. Extraction of a value succeeds only if it is extracted as the correct type.

During marshaling, the **TypeCode** precedes the value on the wire, so the receiving end knows how to interpret the bit pattern that constitutes the value.



5.27 Type any: Concepts

Values of type **any** internally consist of a pair of values.

One member of the pair is an object of type `CORBA::TypeCode`. A type code is an object that stores a description of a type. (For example, for a value of type `long`, the type code would simply indicate `long`; for a complex type, such as a structure, the type code (recursively) describes the name and type of each structure member.)

The second member of the pair is a binary buffer whose contents are interpreted by the ORB according to the description that is provided by the type code. (In other words, the type code describes how to make sense of the binary blob that represents the value.)

Because every **any** value contains a type code, extraction of values can be made type safe. For example, if you are passed an **any** that contains a `double` value and attempt to extract the value as a string, the extraction fails.

Note that the introspection facilities for type **any** are also provided by the type code; the `TypeCode` object offers operations that permit you to interrogate its contents at run time. In addition, CORBA defines an interface called `DynAny`, which permits you to dynamically decompose and compose values of type **any** even if you do not have compile-time knowledge of the types that are involved. These features are quite advanced and rarely needed, so we do not cover them here. (See Henning & Vinoski for details.)

Applications of Type any

Type any is useful if you cannot determine the types you will have to use at compile time. This permits generic interfaces:

```
interface ValueStore {
    void    put(in string value_name, in any value);
    any    get(in string value_name);
};
```

You can also use this to implement variable-length parameter lists:

```
struct NamedValue {
    string name;
    any    value;
};
typedef sequence<NamedValue> ParamList;

interface Foo {
    void op(in ParamList pl);
};
```



5.28 Applications of Type any

Type any is useful whenever it is impossible to know the precise type of a value in advance (at the time you write the IDL). Type any therefore permits you to create generic interfaces that can deal with *any* type of value (even those types of value that have not yet been conceived). The CORBA Event Service is an example where this approach is used.

Another use of any is to simulate variable-length parameter lists as well as optional parameters. For example, the above IDL uses a sequence of name–value pairs as a variable-length parameter list. This approach has the advantage of extensibility. For example, you can add new parameters to the list without having to change the IDL (which can be useful for versioning and gradual evolution of deployed applications).

Note that you should exercise caution regarding the use of type any. While the flexibility of any is attractive, it comes at a significant cost. For one, type any delays compile-time type checking until run time. This not only means that you have to do more work at run time (and write more complex code), but also means that you will not know about type errors unless you actually have test cases that expose them. Second, type any has more run-time overhead than static types and requires more CPU and memory during marshaling, as well as more bandwidth on the wire. This means that there is an inevitable performance penalty associated with type any.

Note that if you have a small number of types that are known at compile time, but you want to make a generic operation for those types, you can use a union instead of an any. The advantage of a union is that it only permits a limited set of types as its members (and that set is determined at compile time), which means that a union offers better type safety than any. Use type any only if you truly need to deal with types that are not known in advance, or if you deliberately want to design an interface to be extendable with new parameters in the future.

Mapping for Type any

IDL **any** maps to a class `CORBA::Any`:

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

    // ...
};
```

The constructor constructs an **Any** containing no value.

The usual deep copy semantics apply to the copy constructor and the assignment operator.



5.29 Mapping for Type any

The IDL type `any` maps to class `Any` in the `CORBA` namespace.¹²

5.29.1 Basic Member Functions

The class provides the usual constructor, copy constructor, and assignment operator. The constructor initializes an `Any` with a type code that indicates “no value”. The copy constructor and assignment operator perform the usual deep copy.

¹²Note that the name of the IDL type is `any`, whereas the name of the C++ type is `Any`. When we use `any`, we mean IDL type, or use the term in its language-independent sense. When we use `Any`, we are referring to the C++ type.

Mapping for Type any (cont.)

Built-in types are inserted using overloaded `<<=` operators in the CORBA namespace:

```
namespace CORBA {
    // ...
    void operator<<=(CORBA::Any &, Short);
    void operator<<=(CORBA::Any &, UShort);
    void operator<<=(CORBA::Any &, Long);
    void operator<<=(CORBA::Any &, ULong);
    void operator<<=(CORBA::Any &, LongLong);
    // More insertion operators for other types here...
    // ...
};
```

Each insertion operator inserts the value and sets the type code of the `Any` as a side effect.

Note that string insertion makes a deep copy.



5.29.2 Insertion of Built-In Types

Insertion into an `Any` is a simple matter of using `operator<<=`. For example:

```
CORBA::Any a;
CORBA::UShort us = 99;
a <<= us; // Insert 99 as an unsigned short
a <<= "Hello"; // Insert deep copy of "Hello"
a <<= (CORBA::Double)3; // Deallocate "Hello", insert 3.0
```

After default construction, `a` contains no value. The first insertion statement inserts the value 99. Because the right-hand side is a variable of type `CORBA::UShort`, the type code of the `Any` is set to indicate `unsigned short`. The second insertion statement inserts a string. This results in the `Any` replacing the previous value (99) with the string “Hello”. Note that string insertion makes a deep copy by default, whether the right-hand side is of type `char *` or `const char *`.¹³ (See page 5-54 for how to achieve consuming insertion.) The third insertion inserts the value 3 as a `double` (which results in deallocation of the previously copied “Hello”).

Be aware of the following frequent mistake:

```
a <<= 99; // Dubious!
a <<= (CORBA::Short)99; // Much better
```

The first insertion is non-portable because it will insert whatever type is mapped to C++ `int`.

¹³ This differs from `String_var`, which makes a copy only if the right-hand side is of type `const char *` and assumes ownership of the right-hand side is of type `char *`.

Mapping for Type any (cont.)

Extraction uses overloaded >>= operators:

```
namespace CORBA {
    // ...
    Boolean operator>>=(const CORBA::Any &, Short &);
    Boolean operator>>=(const CORBA::Any &, UShort &);
    Boolean operator>>=(const CORBA::Any &, Long &);
    Boolean operator>>=(const CORBA::Any &, ULong &);
    Boolean operator>>=(const CORBA::Any &, LongLong &);
    // More extraction operators for other types here...
    // ...
};
```

Each operator returns true if the extraction succeeds.

Extraction succeeds only if the type code in the `Any` matches the type as which a value is being extracted.



5.29.3 Extraction of Built-In Types

Extraction is achieved by using the overloaded extraction operators. Each operator returns true if the type of the variable you extract into matches the type code inside the `Any`. Otherwise, the operator returns false and does not change the value of its right-hand side:

```
CORBA::Any a;
a <<= (CORBA::Long)99;

CORBA::Long long_val;
CORBA::ULong ulong_val;

if (a >>= long_val)           // This must succeed
    assert(long_val == 99); // We know that we put 99 in there...
if (a >>= ulong_val)
    abort();                  // Badly broken ORB!
```

If you receive an `Any` but do not know exactly what type it contains, you can write an if-then-else chain that attempts to extract the value as a different type in each branch until it succeeds. (You can also interrogate the type code of an `Any`; see Henning & Vinoski for details.)

Mapping for Type any (cont.)

Insertion and extraction of char, boolean, and octet require use of a helper type:

```
CORBA::Any a;
a <<= CORBA::Any::from_boolean(0); // Insert false
a <<= CORBA::Any::from_char(0);    // Insert NUL
a <<= CORBA::Any::from_octet(0);   // Insert zero byte

CORBA::Boolean b;
CORBA::Char c;
CORBA::Octet o;
if (a >>= CORBA::Any::to_boolean(b)) {
    cout << "Boolean: " << b << endl;
} else if (a >>= CORBA::Any::to_char(c)) {
    cout << "Char: '\\'" << setw(3) << setfill('0') << oct
        << (unsigned)c << "\\'" << endl;
} else if (a >>= CORBA::Any::to_octet(o)) {
    cout << "Octet: " <<
}
}
```



5.29.4 Extraction of Types Not Distinguishable for Overloading

The C++ mapping permits IDL char, boolean, and octet to map to the same C++ character type. In addition, IDL wchar can map either C++ wchar_t or (with older compilers) to one of the C++ integer types. This means that the mapping cannot overload the <<= and >>= operators for these types because, at the C++ level, they may be the same single type.

As shown above, you must use the `from_<type>` and `to_<type>` helper types for insertion and extraction.¹⁴ Be careful to use the correct helper class. Depending on your ORB, using the incorrect helper type may go undetected. (ORBacus uses distinct types, so you will get a compile time error with ORBacus.)

```
CORBA::Any a;
CORBA::Char c = 'X';
a <<= CORBA::Any::from_boolean(c); // Oops!
// ...
a >>= CORBA::Any::to_octet(c);    // Oops!
```

NOTE: The same caveat applies to type `CORBA::WChar`, which must be inserted and extracted using `CORBA::Any::from_wchar` and `CORBA::Any::to_wchar`, respectively.

¹⁴ These are actually the constructors of a class with the same name; because a different class is created by each constructor, the <<= and >>= operators are overloaded on that class and then can set the type code appropriately.

Mapping for Type any (cont.)

Insertion of a string makes a deep copy and sets the type code to indicate an *unbounded* string:

```
CORBA::Any a;
a <<= "Hello"; // Deep copy, inserts unbounded string
```

Extraction of strings is by *constant* pointer:

```
const char * msg;
if (a >>= msg) {
    cout << "Message was: \" << msg << "\" << endl;
```

```
// Do NOT deallocate the string here!
```

Extraction of strings (as for all other types extracted by pointer) is shallow. (The **Any** continues to own the string after extraction.)

Do not dereference the pointer once the **Any** goes out of scope!



5.29.5 Insertion and Extraction of Strings

Insertion of strings by default makes a deep copy:

```
const char * p = "Hello";
CORBA::Any a;
a <<= p; // Deep copy
a <<= (char *)p; // Deep copy too
```

Note that a deep copy is made regardless of whether the right-hand side is of type `const char *` or `char *`.¹⁵

Extraction of strings is by constant pointer. Note that the returned pointer points at memory internal to the **Any**. This means that you must not deallocate the extracted string because the **Any** retains ownership of its memory. You also must take care not to dereference the pointer beyond the life time of the **Any**, or dereference it once the contents of the **Any** have changed. For example:

```
CORBA::Any a;
a <<= "Hello";
const char * p;
a >>= p; // Extract string

cout << "Any contents: \" << p << "\" << endl;
a <<= (CORBA::Double)3.14;
cout << "Any contents: \" << p << "\" << endl; // Big trouble!
```

¹⁵. This differs from `String_var`, which makes a deep copy only for `const char *`.

Mapping for Type any (cont.)

To insert and extract bounded strings, you must use helper functions:

```
CORBA::Any a;
a <<= CORBA::Any::from_string("Hello", 10);

char * msg;
a >>= CORBA::Any::to_string(msg, 10);
cout << "Message: \" << msg << "\" << endl;
```

The bound for extraction must match the bound for insertion.

Do not insert a string with a bound that is less than the string length.

A bound value of zero indicates an unbounded string.

Consuming insertion can be achieved with an additional parameter:

```
CORBA::Any a;
char * p = CORBA::string_dup("Hello");
a <<= CORBA::Any::from_string(p, 0, 1); // a takes ownership
```



40
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.29.6 Insertion and Extraction of Bounded Strings

For bounded strings, you must use the `from_string` and `to_string` helper functions, which require you to specify the bound explicitly. The bound value does not include the terminating NUL byte of a string. Do not supply a string that is longer than the supplied bound; doing so has undefined behavior:

```
CORBA::Any a;
a <<= CORBA::Any::from_string("Hello", 3); // Undefined!
```

As we will see in Section 6.27, strings that are obtained from CORBA remote calls are dynamically allocated. If you want to insert such a string into an `Any`, it is useful to pass ownership of the string memory to the `Any`. This avoids an additional data copy and is called *consuming* insertion. If you pass a non-zero value as the third parameter to `from_string`, the `Any` assumes that the string was allocated with `string_alloc` or `string_dup` and takes ownership of the memory instead of making a deep copy. The `Any` releases that memory when the value of the `Any` changes or when the `Any` goes out of scope. Obviously, you must not deallocate the string once you have passed its ownership with consuming insertion.

NOTE: Insertion and extraction of wide strings is analogous to normal strings, using `from_wstring` and `to_wstring` helper functions.

Mapping for Type any (cont.)

The IDL compiler generates overloaded operators for each user-defined type:

```
CORBA::Any a;  
Color c = blue;    // Assume enumerated type Color is defined  
a <<= c;
```

```
Color c2;  
int ok = (a >>= c2);  
assert(ok && c2 == blue);
```

This also works for aliases of simple types, such as **TempType**.



5.29.7 Insertion of Simple User-Defined Types

Simple user-defined types, such as enumerations and aliases for simple types, are inserted and extracted like other simple types. For enumerations, the compiler generates a separate pair of overloaded operators for each type.

NOTE: If you insert an alias of a simple type, such as `TempType`, the type code inside the `Any` will indicate the underlying type (`short`), not the alias type (`TempType`). You can force the alias type to be inserted instead; see Henning & Vinoski for details.

Mapping for Type any (cont.)

For structures, unions, and sequences, the compiler generates overloaded insertion and extraction operators:

```
CORBA::Any a;
CCS::Thermostat::BtData btd = ...; // Structure
a <<= btd;                          // Deep copy

CCS::Thermostat::BtData * btdp      // *Pointer* to struct
  = new CCS::Thermostat::BtData;
a <<= btdp;                          // Consuming insertion
```

- Insertion of a structure makes a deep copy.
- Insertion of a pointer is a consuming insertion.

Extraction is always by pointer to constant data:

```
const CCS::Thermostat::BtData * p;
a >>= p;                          // Shallow extraction
```



5.29.8 Insertion of Structures, Unions, and Sequences

Structures, unions, and sequences are inserted using overloaded `<<=` operators generated by the compiler. Note that insertion of a *value* makes a deep copy, whereas insertion of a *pointer* to a value transfers ownership of the value to the `Any`.

Extraction of these types is by pointer to constant data (as for strings). The returned pointer points at memory internal to the `Any`, so you must not dereference the extracted pointer once the `Any`'s contents have changed or the `Any` has gone out of scope. If you want to modify the extracted value, you must make a copy and modify the copy. (This is enforced by the fact that extraction is done via a *constant* pointer):

```
const CCS::Thermostat::BtData * btdp;
if (a >>= btdp) {
    // It's a BtData structure...
    CCS::Thermostat::BtData copy = *btdp; // Make copy
    copy.error_msg = another_message;
}
```

Mapping for Type any (cont.)

Arrays are inserted and extracted using generated helper classes called `<array>_forany`.

```
typedef long arr10[10]; // IDL
```

Insertion and extraction use the `arr10_forany` helper class:

```
CORBA::Any a;
arr10 aten = ...;
a <<= arr10_forany(aten);
// ...

arr10_forany aten_array;
if (a >>= aten_array) {
    cout << "First element: " << aten_array[0] << endl;
}
}
```

Insertion makes a deep copy, extraction is shallow.



5.29.9 Insertion and Extraction of Arrays

Insertion and extraction of arrays requires the use of generated helper classes. This is necessary because of the weak array concept in C++: if we have two arrays of different dimensions but of the same element type, we cannot use overloading to distinguish between the two array types because, when passed to a function, both arrays are passed as a pointer to the first element. The helper class for each array serves to create a unique type that can be used to distinguish between different arrays for the overloaded insertion and extraction operators.

Note that you must be careful to use the correct helper class. The following will not work:

```
arr10 aten; // IDL: typedef long arr10[10];
arr20 atwenty; // IDL: typedef long arr20[20];

a <<= arr20_forany(aten); // Bad news!
a >>= arr10_forany(atwenty); // Bad news!
```

For extraction, the `<array>_forany` helper instance behaves like a normal array, so you can pass it to another function and use the subscript operator to read the elements. However, you must treat the extracted array as read-only because the `Any` retains ownership. If you want to modify an extracted array, you must make a copy with the `<array>_dup` or `<array>_copy` helper functions.

NOTE: We conclude our coverage of type `Any` here. There are a few more features, such as insertion and extraction of object references and exceptions. Refer to Henning & Vinoski for these.

Using `_var` Types

The mapping creates a `_var` type for every user-defined complex type. For variable-length types, a `_var` type behaves like a `string_var`:

- Assignment of a pointer to a `_var` transfers ownership of memory.
- Assignment of `_var` types to each other makes a deep copy.
- Assignment of a `_var` to a pointer makes a shallow copy.
- The destructor deallocates the underlying value.
- An overloaded `->` operator delegates to the underlying value.
- `_var` types have user-defined conversion operators so you can pass a `_var` where the underlying value is expected.

As for strings, `_var` types are simply smart pointers to help with memory management.



5.30 Using `_var` Types

The mapping provides the `String_var` type to make life with dynamically allocated strings easier. For the same reason, the mapping generates a `_var` type for each user-defined complex type. Consider the following IDL:

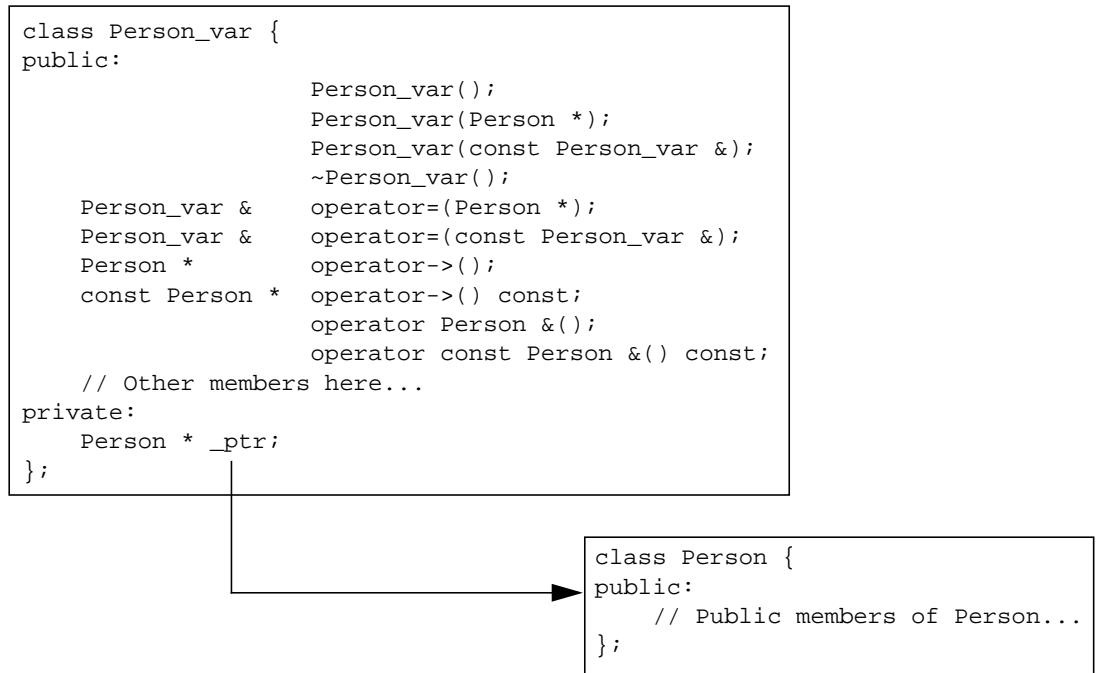
```
struct Person {
    string name;
    string birth_date;
};
```

The mapping generates a `struct Person` from this type, as well as a class called `Person_var`. The `Person_var` class takes the same role as `String_var` does for `char*`: it stores a pointer to a dynamically allocated instance of a `Person` and deallocates that instance when the `Person_var` goes out of scope. Conversion operators on `Person_var` ensure that you can transparently pass a `Person_var` where a `Person` is expected, and an overloaded `->` operator delegates calls on the `Person_var` to the underlying `Person` instance.

The general use pattern for `Person_var` is the same as for a `String_var`. You initialize the `Person_var` with a dynamically allocated instance so you cannot forget to deallocate that instance:

```
{
    Person_var pv = new Person;
    pv->name = CORBA::string_dup("Michi Henning");
    pv->birth_date = CORBA::string_dup("16 Feb 1960");
} // ~Person_var() deallocates here
```

Graphically, we can show this as follows:



Depending on whether the underlying type is fixed-length or variable-length (see page 5-10), the implementation of the member functions varies slightly. We will discuss `_var` types for variable-length underlying types first and then discuss `_var` types for fixed-length underlying types.

Mapping for Variable-Length `_var` Types

For a variable-length structure, union, or sequence `T`, the `T_var` type is:

```
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;
    operator T &();
    operator const T &() const;
    // Other members here...

private:
    T * _ptr;
};
```



45
Basic C++ Mapping
Copyright 2000–2001 IONA Technologies



5.31 Mapping for Variable-Length `_var` Types

The above slide shows the `_var` mapping for variable-length structures, unions, and sequences. (There are a few other member functions, which we discuss on page 5-64.)

`T_var()`

The default constructor initializes the internal pointer to the underlying instance to null. As a result, you cannot use a default-constructed `_var` instance until after you have initialized it.

`T_var(T *)`

The pointer constructor assumes that the passed pointer points to a dynamically allocated instance and takes ownership of the pointer.

`T_var(const t_var &)`

The copy constructor makes a deep copy of both the `T_var` and its underlying instance of type `T`. This means that assignment to a copy-constructed `T_var` affects only that copy and not the instance it was copied from.

`~T_var()`

The destructor deallocates the instance pointed to by the internal pointer.

`T_var & operator=(T *)`

The pointer assignment operator first deallocates the instance of type `T` currently held by the target `T_var` and then assumes ownership of the instance pointed to by its argument.

`T_var & operator=(const T_var &)`

The `T_var` assignment operator first deallocates the instance of type `T` currently held by the target `T_var` and then makes a deep assignment of both the `T_var` argument and the instance of type `T` that the argument points to.

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

The indirection operator is overloaded to permit its use on both constant and non-constant instances of the underlying type. It returns a pointer to the underlying instance. This means that you can use the `T_var` to invoke any member function of the underlying type.

**`operator T &()`
`operator const T &() const`**

The conversion operators permit a `T_var` to be used in places where a constant or non-constant reference to the underlying type is expected.

Additional Member Functions for Sequences and Arrays

If a `_var`'s underlying type is a sequence or array type, the `_var` contains two additional member functions.

**`<elmt_type> & operator[] (CORBA::ULong)`
`const <elmt_type> & operator[] (CORBA::ULong) const`**

The subscript operators are generated if a `T_var` has an underlying sequence or array type and return the element at the given index. This allows you to index into a sequence using the subscript operator on a `_var` and avoids awkward expressions such as `sv->operator[] (0)`.

Example: Simple Use of `_var` Types

```
// IDL: typedef sequence<string> NameSeq;

NameSeq_var ns; // Default constructor
ns = new NameSeq; // ns assumes ownership
ns->length(1); // Create one empty string
ns[0] = CORBA::string_dup("Bjarne"); // Explicit copy

NameSeq_var ns2(ns); // Deep copy constructor
ns2[0] = CORBA::string_dup("Stan"); // Deallocates "Bjarne"

NameSeq_var ns3; // Default constructor
ns3 = ns2; // Deep assignment
ns3[0] = CORBA::string_dup("Andrew"); // Deallocates "Stan"

cout << ns[0] << endl; // "Bjarne"
cout << ns2[0] << endl; // "Stan"
cout << ns3[0] << endl; // "Andrew"
```



5.32 Example: Simple Use of `_var` Types

The above code example illustrates the use of a `_var` type for an underlying sequence type. Construction and assignment make the usual deep copy, so assignment to a `_var` does not affect the `_var` it was initialized with.

Normally, you will use `_var` types mainly to “catch” the return value from a function that returns a dynamically allocated instance. For example:

```
{
    NameSeq_var nsv = get_names(); // Assume get_names returns
                                   // a pointer to a dynamically
                                   // allocated sequence...

    // Use nsv...

} // No need to deallocate anything here
```

As you will see in Section 6.27, this is most useful when you invoke a CORBA operation that returns a variable-length type because variable-length return types are always allocated dynamically.

Mapping for Fixed-Length `_var` Types

`_var` types for fixed-length underlying types is almost identical to `_var` type for variable-length underlying types:

- As usual, the pointer constructor adopts the underlying instance.
- An additional constructor from a `T` value deep-copies the value.
- An additional assignment operator from a `T` deep-assigns the value.

The net effect is that `_var` types for both fixed-length and variable-length underlying types provide intuitive deep copy semantics.

Fixed-length `_var` types are provided for consistency with variable-length `_var` types.

`_var` types hide the memory management difference between fixed-length and variable-length types for operation invocations.



5.33 Mapping for Fixed-Length `_var` Types

The mapping for fixed-length `_var` types is almost identical to the one for variable-length `_var` types. The main difference is that an additional constructor and assignment operator permit initialization and assignment from a value of type `T`. This means that the following code is correct for a fixed-length underlying type:

```
// IDL: struct Point { double x; double y; };

Point origin = { 0.0, 0.0 };
Point_var pv1 = origin;           // Deep copy
Point_var pv2 = new Point;       // pv2 takes ownership
pv2 = pv1;                       // Deep assignment
pv1->x = 99.0;                    // Does not affect pv2 or origin
pv2->x = 3.14;                   // Does not affect pv1, or origin
cout << pv1->x << endl;           // 99.0
cout << pv2->x << endl;           // 3.14
cout << origin->x << endl;        // 0.0
```

NOTE: The mapping for `_var` types may still be confusing at this point. Don't despair just yet—we will see how to use `_var` types to our advantages in Section 6.27.

Dealing with Broken Compilers

Compilers occasionally have problems applying the parameter matching rules correctly when you pass a `_var` type to a function.

Both fixed- and variable-length types have additional member functions to get around such problems:

- `in`: passes a `_var` as an `in` parameter
- `inout`: passes a `_var` as an `inout` parameter
- `out`: passes a `_var` as an `out` parameter

Variable-length `_var` types have a `_retn` member function that return a pointer to the underlying value and transfer ownership.

Fixed-length `_var` types have a `_retn` member function that returns the underlying value itself. No transfer of ownership takes place in this case.



5.34 Dealing with Broken Compilers

As for `String_var`, `_var` types provide an `in`, `inout`, and `out` member function that helps to get around problems with compilers that do not apply the C++ parameter matching rules correctly.

For example, we may have a function `get_vals` that expects a fixed-length parameter `FLT` and a variable-length parameter `VLT` (both `out` parameters). The signature of `get_vals` is:

```
void get_vals(FLT & p1, VLT * & p2);
```

If your compiler chokes on attempts to pass `_var` types to `get_vals`, you can use the `out` member functions to force the appropriate conversion explicitly:

```
FLT_var p1;
VLT_var p2;
get_vals(p1, p2);           // This may not compile,
get_vals(p1.out(), p2.out()); // but this will.
```

6. Client-Side C++ Mapping

Summary

This unit presents the C++ mapping relevant to the client side, that is, initialization and finalization of the ORB run time, how to invoke operations and handle exceptions, and the memory management rules that apply to parameter passing.

Objectives

By the completion of this unit, you will be able to write a client that can communicate with any CORBA server.

Introduction

The client-side C++ mapping covers:

- Mapping for interfaces and object references
- Mapping for operation invocations and parameter passing rules
- Exception handling
- ORB initialization

This unit also covers how to compile and link a client into a working binary program.



1
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.1 Introduction

The C++ mapping for the client side deals mainly with the mapping for interfaces and object references, including the rules for how to pass parameters of various types and the associated memory-management rules. This includes the mapping for IDL exceptions, so clients can deal with errors that arise during operation invocations.

The client-side mapping also addresses how the client must initialize the ORB run time before making CORBA calls.

The specification does not standardize compiling and linking, so how to create a working client executable is necessarily specific to each platform and vendor.

Object References

To make an invocation on an object, the client must have an object reference.

An object reference encapsulates:

- a network address that identifies the server process
- a unique identifier (placed into the reference by the server) that identifies which object in the server a request is for

Object references are opaque to the client. Clients cannot instantiate references directly. (The ORB does this for the client.)

Each object reference denotes exactly one object but an object may have more than one reference.

You can think of references as C++ pointers that can point into another address space.



2
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



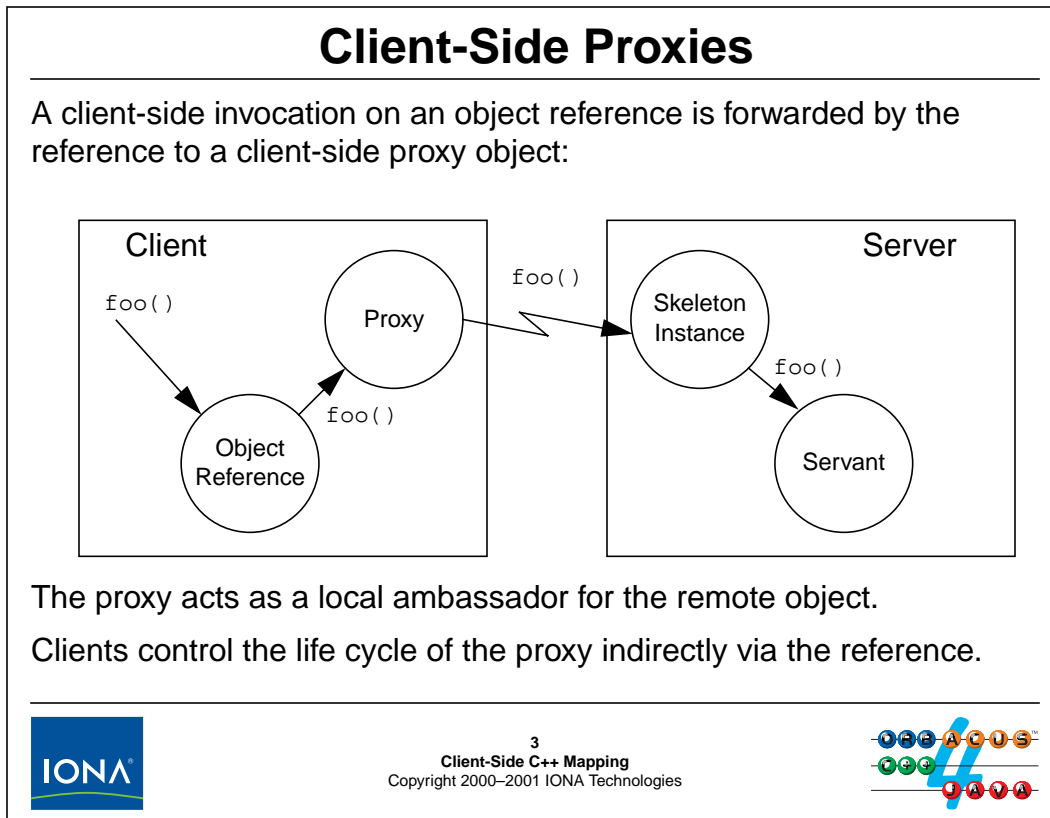
6.2 Object References

For a client to send a message to a CORBA object, the client must have an object reference to the object. The reference encapsulates the details that are required by the ORB to get an invocation to the correct object, namely, the address details for the server and an identifier that determines which object in the server is the target of the invocation.

Because CORBA provides protocol transparency, references are opaque and clients cannot directly instantiate them. Instead, the ORB instantiates references and returns a handle to the reference to the client.

Each object reference uniquely denotes exactly one CORBA object. However, it is possible for a single object to have more than one reference. (This is analogous to maintaining several C++ pointers that all point at the same class instance. Also note that these C++ pointers need not necessarily all have the same value.)

Conceptually, an object reference is much like a C++ class instance pointer, except that it can denote an object in another address space. Otherwise, the semantics are very similar. In particular, an object reference can be nil (point at no object) or dangle (point at an object that is no longer there).



6.3 Client-Side Proxies

When an object reference enters the address space of a client, the ORB instantiates a proxy object and passes an object reference to that proxy to the client application code. The purpose of the proxy is to act as the local ambassador for the remote object. For example, if the remote object has an operation `foo`, the proxy has a C++ member function `foo`. To invoke the `foo` operation, the client invokes `foo` on the proxy via the reference. (The reference is simply a C++ pointer to the proxy instance.)

The implementation of `foo` in the proxy (which is generated by the IDL compiler) then takes all the actions that are required to locate the correct server, marshal the invocation onto the wire, and send it to the server.

The ORB instantiates a proxy whenever a new reference enters a client's address space, so clients never create proxies directly. Once the proxy is instantiated, the client can invoke operations on it, and the ORB locates the server and establishes network connections transparently on behalf of the client. However, the ORB has no way of knowing when a proxy is no longer needed (because the client no longer wants to use the CORBA object represented by that proxy). This means that the client must tell the ORB when it no longer requires a proxy for a particular object. This enables the ORB to reclaim resources associated with a proxy, such as memory and network connections.

Mapping for Interfaces

Interfaces map to abstract base classes:

```
interface MyObject {
    long get_value();
};
```

This generates the following proxy class:

```
class MyObject : public virtual CORBA::Object {
public:
    CORBA::Long get_value();
    // ...
};
```

- For each IDL operation, the class contains a member function.
- The proxy class inherits from **CORBA::Object** (possibly indirectly, if the interface is a derived interface).

Never instantiate the proxy class directly!



4
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.4 Mapping for Interfaces

Each IDL interface results in a separate proxy class of the same name. For every operation on the interface, the proxy class contains a corresponding member function of the same name.

You must not instantiate a proxy class yourself. Instead, the ORB instantiates a proxy instance on your behalf as necessary.

Note that `MyObject` inherits from `CORBA::Object`. This reflects the fact that all IDL interfaces implicitly inherit from `Object` and therefore form an inheritance hierarchy with `Object` as the root.

If the client holds a reference to a proxy instance, calling the `get_value` member function via the reference sends a (possibly remote) message to the object.

The C++ mapping disallows instantiating a proxy directly, as well as declaring a pointer or a reference to a proxy, so the following is illegal:

```
MyObject myobj;        // Cannot instantiate a proxy directly
MyObject * mop;       // Cannot declare a pointer to a proxy
void f(MyObject &);   // Cannot declare a reference to a proxy
```

Note that none of these declarations will produce a compile-time error.

Mapping for Object References

For each interface, the compiler generates two object reference types:

- `<interface>_ptr`

A `_ptr` reference is an unmanaged type that requires you to allocate and deallocate resources explicitly.

- `<interface>_var`

A `_var` reference is a smart type that deallocates resources automatically (similar to `String_var` and other `_var` types).

With either type of reference, you use `->` to call an operation:

```
MyObject_ptr mop = ...;           // Get _ptr reference...
CORBA::Long v = mop->get_value(); // Get value from object

MyObject_var mov = ...;          // Get _var reference...
v = mov->get_value();           // Get value from object
```



5
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.5 Mapping for Object References

For each interface, the IDL compiler generates a `_ptr` and a `_var` reference type. The difference between the two types is the same as for the all the other types: a `_var` reference acts as a wrapper around the underlying `_ptr` type and takes care of resource allocation and deallocation, whereas, if you use `_ptr` types directly, you must deal with these issues yourself.¹

Although the C++ mapping does not require this, a `_ptr` reference is typically implemented as a C++ pointer to the proxy class. In other words, the compiler generates the following definitions for an interface `MyObject`:

```
class MyObject : public virtual CORBA::Object {
    // ...
};

typedef MyObject * MyObject_ptr;

class MyObject_var {
public:
    // ...
private:
    MyObject_ptr _ptr;
};
```

1. We will explore the differences between `_var` and `_ptr` references through the remainder of this unit.

Life Cycle of Object References

Object references can be created and destroyed.

- Clients cannot create references (except for nil references).
- Clients can make a copy of an existing reference.
- Clients can destroy a reference.

The ORB uses the life cycle of references to track when it can reclaim the resources (memory and network connection) associated with a proxy.

Proxies are reference counted. The reference count tracks the number of references that point to a proxy.

Destruction of the last reference to a proxy also destroys the proxy.



6.6 Life Cycle of Object References

Object references have a life cycle, that is, they can be created and destroyed. Reference creation does not apply to clients. This is because references are “pointers” to objects, but the objects are implemented by servers. This means that reference creation is a server-side issue. Clients can always create a nil reference, however. (This is necessary to permit clients to pass nil references to operations.)

Other than creating nil references, clients can copy references and destroy them. Copying a reference creates another reference that denotes the same proxy (and therefore the same CORBA object). Conceptually, copying a reference creates both a new reference and a new proxy for the same object. However, for efficiency reasons, proxies are reference counted. This means that copying a reference creates a new reference that shares the same proxy as the original reference and increments the reference count on the proxy. Destruction of a reference means to decrement the reference count; once the count reaches zero, no more references point at the proxy, so the proxy can be destroyed as well.

Reference Life Cycle Operations

To destroy a reference, you call `release` in the `CORBA` namespace:

```
namespace CORBA {
    // ...
    void release(Object_ptr);
};
```

Every proxy contains a static `_duplicate` member function:

```
class MyObject : public virtual CORBA::Object {
public:
    static MyObject_ptr _duplicate(MyObject_ptr);
    // ...
};
```

`_duplicate` returns a copy of the reference passed as the argument.

The copy of the reference is indistinguishable from the original.



6.7 Reference Life Cycle Operations

To destroy a reference (and indicate to the ORB that it no longer wants to access an object via that reference), the client must call `CORBA::release`. Note that `release` accepts a formal parameter of type `CORBA::Object_ptr`. Because all interfaces ultimately inherit from `Object`, all proxy classes have `CORBA::Object` as their ultimate ancestor. This means that you can pass an object reference to any type of interface to `release`.

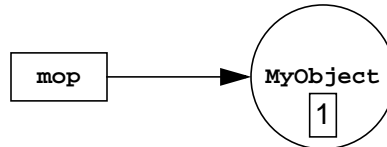
The `_duplicate` member function makes a copy of a reference. the original and the copy are indistinguishable. Every reference created with `_duplicate` must eventually be passed to `release`, to avoid resource leaks. (`_duplicate` and `release` are basically special purpose allocation functions that are used in place of `new` and `delete`. The motivation is much the same as for the other special-purpose allocation functions, such as `string_alloc` and `string_free`: the special-purpose functions can be written to do the correct thing for non-uniform memory-management architectures.).

Object Reference Counts

When the ORB returns a reference to the client, its proxy is always instantiated with a reference count of 1:

```
MyObject_ptr mop = ...; // Get reference from somewhere...
```

This creates the following situation:



The client invokes operations on the proxy via the reference, for example:

```
CORBA::Long v = mop->get_value();
```

The proxy is kept alive in the client while its reference count is non-zero.



6.8 Object Reference Counts

As mentioned on page 6-7, proxies are reference counted for efficiency. Whenever the ORB instantiates a proxy, it creates the proxy with an initial reference count of 1 and returns a reference to the proxy to the client application code.² The client can now invoke operations on the proxy via the reference and the proxy takes care of invoking the operation on the correct object in the server. (Note that this implies that the proxy must be associated with a network connection.)

The ORB keeps the proxy in memory for as long as the reference count remains non-zero.

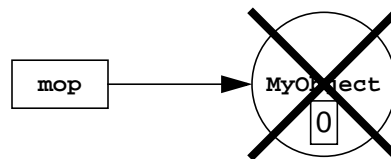
2. For the moment, we will ignore the details of how the client obtains a reference. Suffice it to say that references are returned as the result of invoking an operation.

Object Reference Counts (cont.)

The client is responsible for informing the ORB when it no longer wants to use a reference by calling `release`:

```
MyObject_ptr mop = ...;           // Get reference
CORBA::Long v = mop->get_value(); // Use reference
// ...
CORBA::release(mop);             // No longer interested
                                 // in this object
```

`release` decrements the reference count:



Dropping the reference count to zero causes deallocation.



The client is responsible for informing the ORB when it no longer wants to use a reference (because the ORB has no other way of knowing when a client no longer wants to send messages to an object). This means that every reference that is obtained by the client application code must eventually be passed to `release` (just as every pointer returned from `new` must eventually be passed to `delete`).

A call to `release` decrements the reference count by one. If the count reaches zero, the proxy is destroyed. It follows that you must not use a reference after having released it:

```
MyObject_ptr mop = ...;           // Get reference...
CORBA::Long v = mop->get_value(); // Get a value
CORBA::release(mop);             // Done with object
v = mop->get_value();             // Disaster!!!
```

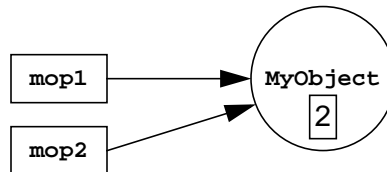
The final statement has undefined behavior (and will most likely cause a core dump) because it accesses deallocated memory.

Object Reference Counts (cont.)

`_duplicate` makes a (conceptual) copy of a proxy by incrementing the reference count:

```
MyObject_ptr mop1 = ...; // Get ref...
MyObject_ptr mop2 = MyObject::_duplicate(mop1); // Make copy
```

The proxy now looks like:



The client must release each of `mop1` and `mop2` exactly once to get rid of the proxy.



After duplicating a reference, the reference count on the proxy goes up by one, with both references pointing to the same proxy. To get rid of the proxy, the client must call `release` exactly once on each reference (in any order):

```
MyObject_ptr mop1 = ...;
MyObject_ptr mop2 = MyObject::_duplicate(mop1);

// Use mop1 and mop2...

CORBA::release(mop1); // Could release mop2 here
CORBA::release(mop2); // Could release mop1 here

// Can't use either mop1 or mop2 from here on
```

You should make it a hard and fast rule to release each reference exactly once and to not use a reference after it was released.

NOTE: Note that with reference-counted proxies, you get away with using a reference after it was released provided that there still exists some other reference that keeps the reference count above zero. However, because the C++ mapping does not require reference counting, such code is non-portable and will fail disastrously with an ORB that does not use reference counting.

Scope of Object References

`_duplicate` and `release` exist purely to manage resources in the local address space.

If a client calls `release` on a reference, the server has no idea that this has happened.

Conversely, if the server calls `release` on one of its references, the client has no idea that this has happened.

Calling `release` *has no effect whatsoever* on anything but the local address space.

You *cannot* implement destruction of objects by calling `release` in the client. Instead, you must add an explicit `destroy` operation.



6.9 Scope of Object References

Object references have a scope that is limited to their local address space and `_duplicate` and `release` exist purely to deal with local resource allocation and deallocation.

If the client calls `release` on an object reference, the server has no idea that this has happened, and vice versa. (Neither `_duplicate` nor `release` cause network traffic, so client and server are necessarily ignorant about what each is doing with respect to these operations.)

The main consequence of this is that the client cannot expect the server to “know” when the client no longer wants an object and expect the server to clean reclaim resources in response to a call to `release` by the client. CORBA simply does not work this way (and a lot of confusion is caused by this misconception about `_duplicate` and `release`).

If you want the server to reclaim resources by destroying an object in response to a client “losing interest” in the object, the client must invoke a `destroy` operation on the object explicitly (see page 12-42).

Nil References

Every proxy class contains a static `_nil` member function. `_nil` creates a nil reference:

```
class MyObject : public virtual CORBA::Object {
public:
    static MyObject_ptr _nil();
    // ...
};
```

You can duplicate a nil reference like any other reference.

You can (but need not) release a nil reference.

Do not invoke an operation on a nil reference:

```
MyObject_ptr nil_obj = MyObject::_nil();    // Create nil ref
nil_obj->get_value();                       // Disaster!!!
```

You can test whether a reference is nil by calling `CORBA::is_nil`.



6.10 Nil References

You can call the static `_nil` member function on the proxy to create a nil reference. You can duplicate and release a nil reference like any other reference. For nil references, `_duplicate` and `release` do nothing, so calling `release` on a nil reference is optional. (It is usually easiest to release nil references like any other reference because that avoids an extra test in the code.)

Never invoke an operation on a nil reference. If you do, you will most likely suffer a core dump.³ This means that you must treat an object reference that is passed to you from somewhere else with caution before using it to make a call (because the reference might be nil). To protect yourself, you can call `CORBA::is_nil` to see whether a reference is nil:

```
MyObject_ptr mop = ...;    // Get reference
if (!CORBA::is_nil(mop)) {
    // OK, not nil, we can make a call
    cout << "Value is: " << mop->get_value() << endl;
} else {
    // We got a nil reference, better not use it!
    cout << "Cannot call via nil reference" << endl;
}
```

3. This is not surprising when you consider that references are usually implemented as pointers. Nil references are naturally implemented as null pointers, so calling an operation via a nil reference ends up dereferencing a C++ null pointer.

References and Inheritance

Proxy classes mirror the IDL inheritance structure.

```
interface Thermometer { /* ... */ };  
interface Thermostat : Thermometer { /* ... */ };
```

The generated proxy classes reflect the same hierarchy:

```
class CORBA::Object { /* ... */ };  
typedef CORBA::Object * Object_ptr;  
  
class Thermometer : public virtual CORBA::Object { /* ... */ };  
typedef Thermometer * Thermometer_ptr;  
  
class Thermostat : public virtual Thermometer { /* ... */ };  
typedef Thermostat * Thermostat_ptr;
```

It follows that object references to a derived interface are compatible with object references to a base interface.



6.11 References and Inheritance

If IDL interfaces inherit from each other, the generated proxy classes reflect the identical inheritance hierarchy (including multiple inheritance). It follows that the C++ subtyping rules apply to object references:

- You can pass a reference to a derived interface where a reference to a base interface is expected.
- You can assign a reference to a derived interface to a reference to a base interface.
- A reference to any interface is compatible with `CORBA::Object_ptr`.

In other words, because `_ptr` references *are* C++ pointers to related classes, they can be implicitly widened to their base classes.⁴

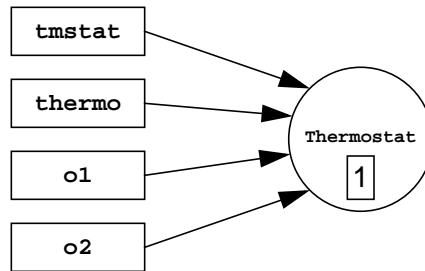
4. If `_ptr` references are not implemented as C++ pointers, the C++ mapping requires that they must obey the same semantics, that is, `_ptr` references support implicit widening whether they are implemented as pointers or not.

Implicit Widening of `_ptr` References

The following is legal code for the CCS:

```
CCS::Thermostat_ptr tmstat = ...;           // Get Thermostat ref
CCS::Thermometer_ptr thermo = tmstat;      // OK, widens
CORBA::Object_ptr o1 = tmstat;            // OK too
CORBA::Object_Ptr o2 = tmstat;            // OK too
```

After these assignments, we have the following situation:



6.12 Implicit Widening of `_ptr` References

As shown in the above code fragment, you can widen object references to any of their base types, including `Object`. Because no calls to `_duplicate` are involved anywhere (except implicitly, during the creation of the proxy), all assignments are shallow and the reference count remains at 1.

In this situation, the client code can invoke operations via any of the reference (but of course can invoke derived operations only on a derived reference):

```
CCS::TempType t;
t = tmstat->get_nominal(); // OK
t = thermo->get_nominal(); // Compile-time error
t = o1->get_nominal();     // Compile-time error
```

A single call to `release` in this situation will destroy the proxy. It does not matter which reference to pass to `release`:

```
CORBA::release(thermo); // or CORBA::release(tmstat)
                        // or CORBA::release(o1)
                        // or CORBA::release(o2)
// Can't use any of the four references from here on...
```

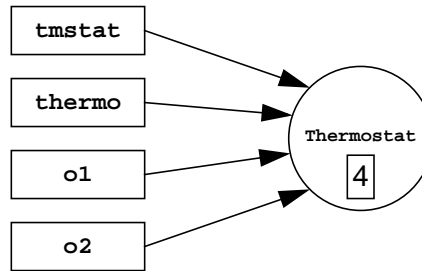
Because the reference count on the proxy is 1, a single call to `release` is sufficient.

Widening with `_duplicate`

You can also explicitly make duplicates during widening:

```
CCS::Thermostat_ptr tmstat = ...; // Get reference
CCS::Thermometer_ptr thermo
    = CCS::Thermometer::_duplicate(tmstat);
CORBA::Object_ptr o1 = CCS::Thermometer::_duplicate(thermo);
CORBA::Object_ptr o2 = CORBA::Object::_duplicate(thermo);
```

The reference count now is 4:



6.13 Widening with `_duplicate`

If you call `_duplicate` on each assignment, the reference count ends up being 4. Consequently, you must call `release` exactly once on each reference to get rid of the proxy again.

It is interesting to think through what happens during the above assignments:

- `CCS::Thermometer_ptr thermo = CCS::Thermometer::_duplicate(tmstat);`

`Thermometer::_duplicate` expects a parameter of type `Thermometer_ptr`, so the `tmstat` argument of type `Thermostat_ptr` is implicitly widened to `Thermometer_ptr`.
- `CORBA::Object_ptr o1 = CCS::Thermometer::_duplicate(thermo);`

Here, the actual parameter type matches the formal parameter type (`Thermometer_ptr`) and the return value is widened from `Thermometer_ptr` to `Object_ptr`. Note that we also could have passed `tmstat` here; the result would be the same because `tmstat` and `thermo` both point at the same object.
- `CORBA::Object_ptr o2 = CORBA::Object::_duplicate(thermo);`

Here, the actual parameter type (`Thermometer_ptr`) is implicitly widened to `Object_ptr`.

Whether or not you call `_duplicate` during assignments really depends on whether you want to decouple the life times of the references. With `_duplicate`, you can independently release each reference in different parts of your code without concern for the other references. Without `_duplicate`, a single call to `release` invalidates all four references.

Narrowing Conversion

The compiler generates a static `_narrow` member function for each proxy that works like a C++ dynamic cast:

```
class Thermometer : public virtual CORBA::Object {
public:
    // ...
    static Thermometer_ptr _narrow(CORBA::Object_ptr);
};

class Thermostat : public virtual Thermometer {
public:
    // ...
    static Thermostat_ptr _narrow(CORBA::Object_ptr);
};
```

`_narrow` returns a non-nil reference if the argument is of the expected type, nil otherwise. `_narrow` implicitly calls `_duplicate`!



16
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.14 Narrowing Conversion

C++ type rules prevent implicit narrowing conversions:

```
CCS::Thermometer_ptr thermo = ...; // Get reference
CCS::Thermostat_ptr tstat = thermo; // Compile-time error
```

You may be tempted to use a cast or a dynamic cast to achieve what you want. However, neither will work:

```
CCS::Thermometer_ptr thermo = ...; // Get ref
CCS::Thermostat_ptr tstat1
    = (CCS::Thermostat_ptr)thermo; // NO!
CCS::Thermostat_ptr tstat2
    = dynamic_cast<CCS::Thermostat_ptr>(thermo); // NO!
```

The first cast is a sledgehammer cast that will get you into trouble as a matter of course. The second attempt (using a dynamic cast) is simply wrong.⁵

Instead, you must use the static `_narrow` member of the proxy to test whether a reference supports a more derived type. `_narrow` returns a non-nil reference if the passed reference supports the corresponding type; otherwise, it returns nil:

5. A call to `_narrow` may need to contact the server but a dynamic cast won't do that.

```
CCS::Thermometer_ptr thermo = ...;           // Get reference

// Try down-cast
CCS::Thermostat_ptr tmstat = CCS::Thermostat::_narrow(thermo);
if (CORBA::is_nil(tmstat)) {
    // thermo isn't a Thermostat
} else {
    // thermo is-a Thermostat
    cout << "Nominal temp: " << tmstat->nominal_temp() << endl;
}
CORBA::release(tmstat);                      // _narrow calls _duplicate!
```

The call to `_narrow` succeeds (returns a non-nil reference) if the actual type of the object denoted by `thermo` is of type `Thermostat` or a type derived from `Thermostat`.

Note that `_narrow` returns a copy of its argument, so you must release the returned reference. In the above code, we release the reference unconditionally, whether `_narrow` returned nil or not. This illustrates that calling `release` on a nil reference does not harm (and can simplify your code).

Illegal Uses of References

The following are illegal and have undefined behavior:

- comparison of references for equality or inequality
- applying relational operators to references
- applying arithmetic operators to references
- conversion of references to and from `void *`
- Down-casts other than with `_narrow`
- Testing for nil other than with `CORBA::is_nil`

Some of these may happen to work and may even do the right thing, but they are still illegal!



17
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.15 Illegal Uses of References

The following is a list of illegal uses of references that are explicitly flagged as producing undefined behavior by the C++ mapping.⁶ Do not use any of these constructs even if they happen to work—they may suddenly stop working in a later release of the ORB.

You cannot compare references for equality or inequality with `==` or `!=`:

```
CORBA::Object_ptr o1 = ...;
CORBA::Object_ptr o2 = ...;
if (o1 == o2)           // Undefined behavior!
    ...;
if (o1 != o2)          // Undefined behavior!
    ...;
```

You cannot use relational operators on references:

```
CORBA::Object_ptr o1 = ...;
CORBA::Object_ptr o2 = ...;
if (o1 < o2)           // Undefined behavior!
    ...;               // <, <=, >, and >= have no meaning
```

6. The C++ mapping does not permit these constructs because doing so would unduly restrict ORB implementations.

You cannot apply arithmetic operators to references:

```
CORBA::Object_ptr o1 = ...;
CORBA::Object_ptr o2;
o2 = o1 + 5;                // Meaningless!
ptrdiff_t diff = o2 - o1;   // Meaningless!
```

You cannot convert references to and from void *:

```
CORBA::Object_ptr o = ...;
void * v = (void *)o;      // Meaningless!
```

You cannot down-cast other than with `_narrow`:

```
CCS::Thermostat_ptr tmstat = ...;    // Get reference
CORBA::Object_ptr o = tmstat;        // OK
CCS::Thermostat_ptr tmstat2;

tmstat2 = dynamic_cast<CCS::Thermostat_ptr>(o);    // Bad!
tmstat2 = static_cast<CCS::Thermostat_ptr>(o);     // Bad!
tmstat2 = reinterpret_cast<CCS::Thermostat_ptr>(o); // Bad!
tmstat2 = (CCS::Thermostat_ptr)o;                 // Bad!

tmstat2 = CCS::thermostat::_narrow(o);             // OK
```

You cannot test for nil other than with `CORBA::is_nil`:

```
if (tmstat) ...                // Illegal!
if (tmstat != 0) ...           // Illegal!
if (tmstat != CCS::Thermostat::_nil()) ... // Illegal!

if (!CORBA::is_nil(tmstat)) ... // OK
```

Pseudo Objects and the ORB Interface

The **CORBA** module contains an interface **ORB**:

```
module CORBA {
    interface ORB {      // PIDL
        // ...
    };
    // ...
};
```

The ORB interface is used to initialize the ORB run time and to get access to initial object references.

The **PIDL** comment indicates Pseudo-IDL. PIDL interfaces are implemented as library objects and used to access the ORB run time.

PIDL objects are not fully-fledged objects because they cannot be accessed remotely.



6.16 Pseudo Objects and the ORB Interface

The CORBA module contains a number of interface definitions, including the ORB interface. The ORB interface contains operations that provide access to the ORB run time, for example, for initialization and finalization. Many of the interfaces in the CORBA module are marked with a PIDL comment, meaning “Pseudo-IDL”. Interfaces defined in PIDL use the usual IDL syntax, but are subject to a number of restrictions. The most important one is that you cannot access a PIDL interface remotely. This is because pseudo-objects are implemented as library objects that are local to each address space and do not make sense if used remotely. You can think of pseudo-objects as objects that take care of low-level and vendor-specific implementation details and offer operations that make sense only for the local address space.

Apart from not being able to invoke them remotely, pseudo-objects are subject to a number of other restrictions:

- Pseudo-interfaces do not inherit from `Object`.
- Object references for pseudo-objects cannot be passed to another address space.
- Operations on pseudo-objects cannot be invoked via the Dynamic Invocation Interface (DII).
- Pseudo-interfaces do not have definitions in the Interface Repository (IFR).
- Pseudo-interfaces may have special-purpose language mappings that deviate from the normal mapping rules.

You are not going to notice any of these restrictions (bar the final one) because doing these things does not make sense for pseudo-objects. The special-purpose mapping rules are something you need to be aware of, and we will illustrate them wherever necessary.

ORB Initialization and Finalization

The ORB interface contains an initialization and finalization operation. You must call these to initialize and clean up the ORB:

```
CORBA::ORB_ptr orb; // Global for convenience

int
main(int argc, char * argv[])
{
    try {
        orb = CORBA::ORB_init(argc, argv);
    } catch (...) {
        cerr << "Cannot initialize ORB" << endl;
        return 1;
    }
    // Use ORB...
    orb->destroy();          // Must destroy!
    CORBA::release(orb);    // Clean up
    return 0;
}
```



19
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.17 ORB Initialization

Every client (and server) must initialize the ORB run time before it can make (or accept) CORBA invocations. You do this by calling the library function `ORB_init`; it has the following signature:

```
namespace CORBA {
    // ...
    ORB_ptr ORB_init(
        int &          argc,
        char **        argv,
        const char *   orb_identifier = ""
    );
    // ...
};
```

You must pass `argc` and `argv` to `ORB_init`. Note that `ORB_init` is passed a reference to `argc`, so it can change the value of `argc`, and is passed a non-constant pointer to the arguments, so it can change the contents of the `argv` vector. The signature of `ORB_init` looks that way because `ORB_init` scans the command line for ORB-specific command-line options. Any option beginning with `-ORB` is taken to be targeted at the ORB and is removed from the argument vector by `ORB_init`. This means that, once `ORB_init` returns, the argument vector has been stripped of ORB-specific options and now only contains application-specific options. (This is the same idea as for the initialization call for the X11 tool kit.)

`ORB_init` returns an object reference to the ORB object. That reference is like any other reference as far as memory management is concerned. This means that you must release it

eventually (and you must release it as the final reference in your code), as shown on the preceding slide. However, before calling `release` on the ORB reference, you must invoke the `destroy` operation on the ORB. This is necessary in order avoid resource leaks, particularly in multi-threaded programs.⁷

NOTE: Because the ORB interface is typically required throughout much of your code, it is common to make its reference a global variable.

The need to pass `argc` and `argv` to `ORB_init` means that you cannot make CORBA calls before you have entered `main`. In particular, you cannot call CORBA operations from global constructors. Do not attempt to cheat by making a dummy argument vector and calling `ORB_init` from a global constructor. Doing so has undefined behavior and may well fail (because the ORB libraries themselves may be using global constructors).

NOTE: In order to be able to use predefined types, such as `ORB_ptr`, you must include a header file. The specification does not define the names of header files, so the name and number of header files varies with each ORB. For ORBacus, you must include `OB/CORBA.h`.

7. Some threads packages do not permit you to leave `main` while there are threads other than the main thread. `ORB::destroy` joins with all threads before it returns.

Stringified References

You can convert an object reference into a string and back:

```
interface ORB {
    string  object_to_string(in Object obj);
    Object  string_to_object(in string str);
    // ...
};
```

Stringified references can be used for bootstrapping:

- The server creates an object and writes its stringified reference to a file.
- The client reads the file and uses the reference to access the object.

While simple, there are drawbacks to this idea. A Naming Service does the same job better.

Stringified references are also useful to store references in databases.



6.18 Stringified References

You can convert an object reference into a string with `object_to_string` and later turn it back into an object reference with `string_to_object`. While stringified, a reference can travel by out-of-band means. For example, you can store it in database or send it via e-mail.

Stringified references can be used to permit a client to bootstrap: the server writes a stringified reference to a key object (such as the controller) into a file and the client reads the string from the file and converts it back to a reference.⁸

A stringified reference looks something like this:

```
IOR:01d072402000000049444c3a61636d652e636f6d2f4343532f436f6e74726
f6c6c65723a312e30000100000000000000bc00000001010240110000006a616e
75732e6f6f632e636f6d2e6175003e950624000000abacab31393531323833343
632005f526f6f74504f410000cafebab38b36f06000000000100000001000000
6c00000001394f40010001000a00000020001000300010004000100050001000
60001000700010008000100090001000100010520000100090101000c00000000
01010001000100020001000300010004000100050001000600010007000100080
00100090001000100010520000100
```

Note that, even though the string is quite long, a stringified reference takes up much less room in memory.

⁸ This is a non-scalable and rather primitive mechanism, but will suffice for now. A better way for clients to obtain application references is via the Naming Service.

If we assume that the CCS server produces a stringified reference to the controller, we can write the client code as follows. (In this example, we pass the stringified reference as `argv[1]`.)

```
#include <OB/CORBA.h>    // Import CORBA module
#include "CCS.h"         // Import CCS system (IDL-generated)

CORBA::ORB_ptr orb;     // Global for convenience

int
main(int argc, char * argv[])
{
    // Initialize the ORB
    orb = CORBA::ORB_init(argc, argv);

    // Get controller reference from argv
    // and convert to object.
    CORBA::Object_ptr obj = orb->string_to_object(argv[1]);
    if (CORBA::is_nil(obj)) {
        cerr << "Nil controller reference" << endl;
        abort();
    }

    // Try to narrow to CCS::Controller.
    CCS::Controller_ptr ctrl;
    ctrl = CCS::Controller::_narrow(obj);
    if (CORBA::is_nil(ctrl)) {
        cerr << "Wrong type for controller ref." << endl;
        abort();
    }

    // Use controller...

    CORBA::release(ctrl); // Clean up
    CORBA::release(obj);  // Ditto...
    orb->destroy();        // Must destroy before leaving main()
    CORBA::release(orb);  // Ditto...

    return 0;
}
```

You will find boilerplate code very similar to this in almost every client, except that you might use the Naming Service to obtain the first application reference. Also note that, for the time being, we are ignoring error handling.

Note that the code is careful to release all the references it has acquired to avoid memory leaks. (We will see in page 9-23 how to avoid leaks in the presence of errors or exceptions.)

Also note that this code is not exception safe, in the sense that it does not call `ORB::destroy` if anything goes wrong. (Calling `abort` in case of an error is rarely a viable error-handling strategy.) Here is a version that uses exceptions to get out of a code block in such a way that `ORB::destroy` will always be called:

```
#include <OB/CORBA.h> // Import CORBA module
#include "CCS.h"       // Import CCS system (IDL-generated)

CORBA::ORB_ptr orb = CORBA::ORB::_nil(); // Nil initialized

int
main(int argc, char * argv[])
{
    CCS::Controller_ptr ctrl = CCS::Controller::_nil();

    try {
        // Initialize the ORB
        orb = CORBA::ORB_init(argc, argv);

        // Get controller reference from argv
        // and convert to object.
        CORBA::Object_ptr obj = orb->string_to_object(argv[1]);
        if (CORBA::is_nil(obj)) {
            cerr << "Nil controller reference" << endl;
            abort();
        }

        // Try to narrow to CCS::Controller.
        ctrl = CCS::Controller::_narrow(obj);
        if (CORBA::is_nil(ctrl)) {
            cerr << "Wrong type for controller ref." << endl;
            abort();
        }

        // Use controller...

    } catch (...) {
        CORBA::release(ctrl); // Clean up
        CORBA::release(obj); // Ditto...
        orb->destroy();       // Must destroy before leaving ma
in()
        CORBA::release(orb); // Ditto...
    }

    return 0;
}
```

Stringified References (cont.)

Stringified references are interoperable and can be exchanged among clients and servers using different ORBs.

Nil references can be stringified.

You must treat stringified references as opaque:

- *Never* compare stringified references to determine whether they point at the same object.
- Do not use stringified references as database keys

The *only* things you can legally do with stringified references are:

- obtain them from **object_to_string**
- store them for later retrieval
- convert them back to a reference with **string_to_object**



21
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



Stringified references are written in a format that is interoperable, so you can exchange stringified references among clients and servers that use different ORBs.

Nil references can be stringified like any other reference. A stringified nil reference looks like this:

```
IOR:01000000010000000000000000000000
```

You must treat stringified references as opaque. Any attempt to make sense of the contents of a reference violates the CORBA object model. In particular, *never* compare stringified references to determine whether they point at the same object. The outcome is completely undefined. (In fact, the same single object may have different stringified references as different times.)

Because references are opaque, it follows that you cannot use them as database keys if you want to store objects in a database (because use of references as keys implies comparison for equality).

The *only* legal uses of stringified references are to create them with `object_to_string`, to store them for later retrieval, and to pass them to `string_to_object` to convert them back into a reference.

NOTE: During debugging, it can be useful to examine the contents of a reference. You can do this with the `iordump` tool supplied with ORBacus.

Stringified References (cont.)

You can use a URL to denote a file containing a stringified reference:

- `file:///usr/local/CCS/ctrl.ref` (UNIX)
- `file://c:\winnt\Program%20Files\CCS\ctrl.ref` (NT)

`string_to_object` accepts such URLs as a valid IOR strings and reads the stringified reference from the specified file.

This mechanism is specific to ORBacus!



22
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



Instead of passing a stringified reference to `string_to_object`, you can also pass it a URL beginning with `file://`. The URL must denote the absolute pathname to a file that contains a stringified IOR as the first line of the file. `string_to_object` reads the stringified IOR from the specified file and the proceeds as usual. This mechanism is useful mainly to conveniently deal with stringified references that must be passed as command-line arguments. For example, in UNIX, you can write

```
./server `cat /usr/local/CCS/ctrl.ref`
```

to pass the stringified reference in a file on the command line. However, for Windows, the default command-line interpreter cannot make such substitutions, which makes it awkward to pass IORs as program arguments. A `file://` URL gets around this problem. For example:

```
./server file://c:\winnt\Program%20Files\CCS\ctrl.ref
```

or:

```
./server file:///usr/local/CCS/ctrl.ref
```

NOTE: Not all characters are legal in URLs without escaping them. For example, a single space must be represented as `%20`.

NOTE: This feature is non-standard and specific to ORBacus.

The Object Interface

The **CORBA** module contains the **Object** interface.

All references provide this interface (because all interfaces inherit from **Object**):

```
interface Object { // PIDL
    Object duplicate();
    void release();
    boolean is_nil();
    boolean non_existent();
    boolean is_equivalent(in Object other_object);
    unsigned long hash(unsigned long max);
    boolean is_a(in string repository_id);
    // ...
};
```

Note that **Object** is defined in PIDL.



6.19 The Object Interface

The **Object** interface is the ultimate base interface, so all interfaces (and therefore, all references) support the operations on **Object**. Note that **Object** is defined in PIDL, so the normal language mapping rules need not necessarily apply. We have already seen a deviation from the normal mapping rules for `duplicate` (which maps to `_duplicate` on each proxy class), `release` (which maps to `CORBA::release`), and `is_nil` (which maps to `CORBA::is_nil`).⁹

The remaining operations are mapped as follows:

```
class Object {
public:
    // ...
    Boolean _non_existent();
    Boolean _is_equivalent(Object_ptr other_object);
    ULong _hash(ULong max);
    Boolean _is_a(const char * repository_id);
    // ...
};
```

Note that the C++ mapping adds a leading underscore to each operation name. This is to prevent name clashes with operations on an actual interface. For example, if you create an interface with a

9. The reason for these deviations is that `_duplicate` must return a different type for each interface, so it cannot be a single member function on **Object**. `release` and `is_nil` map to functions in the CORBA namespace (instead of member functions on the proxy) because, otherwise, they could not be called on nil references.

`non_existent` operation, that operation will be mapped to `non_existent` on the derived proxy class and therefore not clash with `_non_existent` on the inherited Object base class. The operations have the following semantics:

Boolean `_non_existent()`

This operation returns true if an object reference dangles (points at an object that is no longer there because it has been deleted). If `_non_existent` returns true, this is an authoritative answer that the object no longer exists and will never exist again. Conversely, if `_non_existent` returns false, this is an authoritative answer that the object is known to exist. (However, when you try to reach it, it may not work because, for example, the network may have just gone down.)

`_non_existent` contacts the server to determine whether an object exists. This means that the operation may fail (for example, because connectivity cannot be established). If `_non_existent` cannot make a determination of an object's status due to an error, it raises an exception (see page 6-66).

Boolean `is_equivalent(Object_ptr other_object)`

This operation determines whether two references (not objects) are identical. (See page 6-34 for more detail.)

ULong `_hash(ULong max)`

This operation returns a hash value for a reference in the range 0 to `max-1`. You will rarely (if ever) have a use for this operation.¹⁰

Boolean `_is_a(const char * repository_id)`

This operation returns true if the reference supports the specified interface. You can call it like this:

```
CORBA::Object_ptr obj = ...;           // Get a reference
if (!CORBA::is_nil(obj) {
    if (obj->_is_a("IDL:acme.com/CCS/Controller:1.0")) {
        // It's a controller
    } else {
        // It's something else
    }
} else {
    // Got a nil reference
}
```

The `is_a` operation is provided mainly for clients which do not have static type knowledge (are not linked with the stubs for an interface) and are using the DII instead. If you have the stubs linked, it is easier to call `_narrow`, which achieves the same thing.

¹⁰.It was added mainly to make protocol bridges easier to implement.

Object Reference Equivalence

is_equivalent tests if two object references are identical:

- if they are equivalent, the two references denote the same object
- if they are not equivalent, the two references may or may not denote the same object

is_equivalent test object *reference* identity, not *object* identity!

Because a single object may have several different references, a false return from **is_equivalent** does *not* indicate that the reference denote different objects!

is_equivalent is a local operation (never goes out on the wire).



24
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.20 Object Reference Equivalence

You can call `_is_equivalent` as follows:

```
CORBA::Object_ptr o1 = ...; // Get reference
CORBA::Object_ptr o2 = ...; // Another one...

if (o1->_is_equivalent(o2)) {
    // o1 and o2 denote the same object
} else {
    // o1 and o2 may or may not denote the same
    // object, who knows...
}
```

Note that `_is_equivalent` provides only a partial answer with respect to *object* identity. If the answer is true, you know definitively that the two references are identical and, therefore, denote the same object. However, if the answer is false, you definitively know that the references are different; however, that does not imply anything about the objects denoted by the two references. Both references may, in fact, denote the same object, despite the fact that the references are different.¹¹

¹¹ These semantics were chosen deliberately to make it possible to build object domain bridges, which must translate object references as they cross domains.

Providing Object Equivalence Testing

If you require *object* identity, you must supply it yourself:

```
interface Identity {  
    typedef whatever UniqueID;  
    UniqueID id();  
};
```

You can use this interface as a base interface for your objects.

Clients can invoke the **id** operation to obtain a unique ID for each object.

Two objects are identical if their IDs are identical.

Note that the asset number in the CCS serves as object identity.



25
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.21 Providing Object Equivalence Testing

If your clients require object identity (as opposed to reference identity), you must arrange for it by adding an explicit operation to each interface. The operation must return an ID that is unique across all objects. (Not that the asset numbers in the CCS serve the same purpose.)

Object identity is more expensive than reference identity because it requires a remote call (whereas `is_equivalent` is a local operation).

_var References

_var references are used to make memory leaks less likely.

Like other **_var** types, **_var** references make deep copies and release their reference in the destructor:

```

{
    CORBA::Object_var obj = orb->string_to_object(...);
    CCS::Controller_var ctrl = CCS::Controller::_narrow(obj);
    // ...
} // No need to release anything here...

```

Use **_var** references to “catch” object references returned from invocations.

_var references are extremely useful for exception safety!



6.22 **_var** References

_var references, like other **_var** types, help to avoid resource leaks. **_var** references use the usual deep copy semantics and are generated for each interface. For example, here is the **_var** reference type for a **Thermometer**:

```

class Thermometer_var {
public:
    Thermometer_var();
    Thermometer_var(Thermometer_ptr &);
    Thermometer_var(const Thermometer_var &);
    ~Thermometer_var();

    Thermometer_var & operator=(Thermometer_ptr &);
    Thermometer_var & operator=(const Thermometer_var &);
    operator Thermometer_ptr &();
    Thermometer_ptr operator->() const;

    Thermometer_ptr in() const;
    Thermometer_ptr & inout();
    Thermometer_ptr & out();
    Thermometer_ptr _retn();
private:
    Thermometer_ptr _ptr;
};

```

Thermometer_var()

The default constructor initializes the `_var` to a nil reference.

Thermometer_var(Thermometer_ptr &)

The `_ptr` constructor takes ownership of the reference; the `_var` calls `release` on the reference when it goes out of scope.

Thermometer_var(const Thermometer_var &)

The copy constructor makes a deep copy, that is, it increments the reference count on the proxy by calling `_duplicate`.

~Thermometer_var()

The destructor calls `release` on the reference.

Thermometer_var & operator=(Thermometer_ptr &)

The assignment operator from a `_ptr` first release the reference that is currently held and then takes ownership of its argument.

Thermometer_var & operator=(const Thermometer_var &)

The assignment operator from a `_var` first releases the reference that is currently held and then calls `_duplicate` on the right-hand side, taking ownership of the copy.

operator Thermometer_ptr &()

This conversion operator permits you to pass a `_var` reference where a `_ptr` reference is expected and makes passing and assignment of `_var` references transparent.

Thermometer_ptr operator-> const

The indirection operator returns the underlying `_ptr` reference, delegating any operation invocations to the proxy.

Thermometer_ptr in() const**Thermometer_ptr & inout()****Thermometer_ptr& out()**

The explicit conversion operators are provided to get around compiler problems and have the same semantics as for other `_var` types.

Thermometer_ptr _retn()

The `_retn` member function returns the underlying reference and sets the internal `_ptr` reference to nil. This transfers ownership from the `_var` to the caller.

Using `_var` references, we can rewrite the code on page 6-27 as follows:

```
#include <OB/CORBA.h>    // Import CORBA module
#include "CCS.hh"        // Import CCS system (IDL-generated)

CORBA::ORB_var orb;     // Global for convenience

int
main(int argc, char * argv[])
{
    // Initialize the ORB
    orb = CORBA::ORB_init(argc, argv);

    // Get controller reference from argv
    // and convert to object.
    CORBA::Object_var obj = orb->string_to_object(argv[1]);
    if (CORBA::is_nil(obj)) {
        cerr << "Nil controller reference" << endl;
        return 1;
    }

    // Try to narrow to CCS::Controller.
    CCS::Controller_var ctrl = CCS::Controller::_narrow(obj);
    if (CORBA::is_nil(ctrl)) {
        cerr << "Wrong type for controller ref." << endl;
        return 1;
    }

    // Use controller...

    // No need to release anything here (but
    // ORB::destroy() is still necessary).
    orb->destroy();

    return 0;
}
```

This code is far superior to the version on page 6-27. Firstly, the code is not cluttered with explicit calls to `release`. Secondly, in case of an early return out of `main`, the references will still be correctly released (whereas the code on page 6-27 had a leak in this case). Finally, if any call throws an exception, the compiler will invoke the destructors of all local variables, so even in the presence of exceptions, the references will be correctly released.

_var References and Widening

_var references do not mirror the IDL inheritance hierachy:

```
class Thermometer : public virtual CORBA::Object { /* ... */ };
class Thermostat : public virtual Thermometer { /* ... */ };
```

```
typedef Thermometer * Thermometer_ptr;
typedef Thermostat * Thermostat_ptr;
```

```
class Thermometer_var { /* ... */ }; // No inheritance!
class Thermostat_var { /* ... */ }; // No inheritance!
```

Implicit widening on _var references therefore does not compile:

```
Thermostat_var tstat = ...;
Thermometer_var thermo = tstat; // Compile-time error
CORBA::Object_var obj = tstat; // Compile-time error
```

You can use `_duplicate` to widen a reference explicitly:

```
Thermostat_var tstat = ...;
Thermometer_var thermo = Thermometer::_duplicate(tstat);
```



27
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.23 _var References and Widening

As opposed `_ptr` references, `_var` references are not in an inheritance relationship. This means that you cannot implicitly widen among `_var` references; the code simply does not compile.

However, using `_duplicate`, you can widen references explicitly. For example:

```
Thermostat_var tstat = ...;

Thermometer_var thermo;
thermo = Thermometer::_duplicate(tstat);
thermo = Thermostat::_duplicate(tstat);

CORBA::Object_var obj;
obj = Thermostat::_duplicate(tstat);
obj = Thermometer::_duplicate(tstat);
obj = CORBA::Object::_duplicate(tstat);
```

Because `_duplicate` both expects and returns a `_ptr` reference, these statements compile and work correctly. (The implicit widening on `_ptr` references means that the compiler always finds a matching signature.)

References Nested in Complex Types

References that are members of structures, unions, or exceptions, or elements of sequences or arrays behave like `_var` references:

```
struct DevicePair {
    Thermometer mem1;
    Object      mem2;
};
```

The same rules as for strings apply:

```
Thermometer_var thermo = ...;
Thermostat_var tstat = ...;
```

```
DevicePair dp;
dp.mem1 = thermo;           // Deep assignment
dp.mem2 = Object::_duplicate(thermo); // Deep assignment
DevicePair dp2 = dp;       // Deep copy
dp2.mem2 = orb->string_to_object(argv[1]); // No leak here
```



28
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.24 References Nested in Complex Types

If an object reference is a member of a complex type, the member behaves like a `_var` reference. In other words, whenever an object reference member is the target of an assignment, it will take ownership of the right-hand side. If the right-hand side is another reference member or a `_var` reference, `_duplicate` is called by the assignment operator, with the net effect being a deep copy.

Just with `String_var` and `char *`, you can mix `_var` and `_ptr` references. Simply remember that

- when a `_ptr` is on the left-hand side, the assignment is shallow;
- when a `_var` is on the left-hand side, it will take ownership of the right-hand side;
- when a `_var` is on both sides of an assignment, you get a deep copy.

Mapping for Operations

Operations on IDL interfaces map to proxy member functions with the same name.

If you have a `_var` or `_ptr` reference to a proxy instance, you invoke the member function via the reference's `->` operator.

The proxy member function sends the request to the remote object and blocks until the reply arrives.

The proxy unmarshals the results and returns.

The net effect is a synchronous procedure call.



6.25 Mapping for Operations

As we saw on page 6-5, operations map to proxy member functions with the same name. For example:

```
interface Example {
    void      send(in char c);
    oneway void put(in char c);
    long      get_long();
    string    id_to_name(in string id);
};
```

The generated proxy class contains the corresponding member functions:

```
class Example : public virtual CORBA::Object {
public:
    // ...
    void      send(CORBA::Char c);
    void      put(CORBA::Char c);
    CORBA::Long get_long();
    char *    id_to_name(const char * id);
    // ...
};
```

Note that the signature for twoway and oneway operations is the same. (But the generated code for each function does different things for twoway and oneway dispatch.)

Mapping for Attributes

IDL attributes map to a pair of member functions, one to read the value and one to write it.

readonly attributes only have an accessor and no modifier.

```
interface Thermometer {
    readonly attribute unsigned long    asset_num;
    attribute string                    location;
};
```

The proxy contains:

```
class Thermometer : public virtual CORBA::Object {
public:
    CORBA::ULong asset_num();           // Accessor
    char *       location();           // Accessor
    void         location(const char *); // Modifier
};
```



30
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.26 Mapping for Attributes

Attributes map to a pair of member functions, an accessor and a modifier. If an attribute is **readonly**, no modifier is generated by the compiler.

To read an attribute, you simply invoke the accessor:

```
CCS::Thermometer_var t = ...; // Get reference
CORBA::ULong anum = t->asset_num();
cout << "Asset number is " << anum << endl;
```

Similarly, to write an attribute, you invoke the modifier:

```
CCS::Thermometer_var t = ...;
t->location("Room 414");
```

This example illustrates that there is truly no difference between operations and attributes. Either way, the proxy makes a synchronous remote procedure call to the server, so operation and attribute accesses have exactly the same performance.

Parameter Passing

The rules for parameter passing depend on the type and direction:

- Simple **in** parameters are passed by value.
- Complex **in** parameters are passed by constant reference.
- **inout** parameters are passed by reference.
- Fixed-length **out** parameters are passed by reference.
- Variable-length **out** parameters are dynamically allocated.
- Fixed-length return values are passed by value.
- Variable-length return values are dynamically allocated.
- Fixed-length array return values are dynamically allocated.

Note: Variable-length values that travel from server to client are dynamically allocated. Everything else can be allocated on the stack.



31
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.27 Parameter Passing

The parameter passing rules are quite complex and depend on the type of a parameter as well as the direction in which it is passed. The motivation for these rules is efficiency: the mapping avoids data copying and dynamic memory allocation wherever possible. This means that values are dynamically allocated only if they are of variable length and travel from server to client. (This also makes the stubs reentrant so there are no thread-safety issues inherent in the API.)

Parameter Passing (cont.)

Consider an operation that passes a **char** parameter in all possible directions:

```
interface Foo {
    char op(in char p_in, inout char p_inout, out char p_out);
};
```

The proxy signature is:

```
CORBA::Char op(
    CORBA::Char p_in,
    CORBA::Char & p_inout,
    CORBA::Char & p_out
);
```

This signature is no different than for any normal C++ function that passes parameters in the same directions.



6.27.1 Rules for Simple Types

Above is an operation that passes a **char** in all possible directions. The signature of the proxy reflects that the type is simple and of fixed length, and so can be passed by value or, for **inout** and **out** parameters, by reference. (The same rules apply for other simple types of fixed length, such as **long**, **double**, **octet**, and so on.) A call via an object reference to this operation in the client might look like this:

```
Foo_var fv = ...; // Get reference

CORBA::Char inout_val;
CORBA::Char out_val;
CORBA::Char ret_val;

inout_val = 'A';
ret_val = fv->op('X', inout_val, out_val);

cout << "ret_val: " << ret_val << endl;
cout << "inout_val: " << inout_val << endl;
cout << "out_val: " << out_val << endl;
```

Obviously, you must initialize **in** and **inout** parameters before the call, otherwise you will send undefined values.

Parameter Passing (cont.)

Fixed-length unions and structures are passed by value or by reference:

```
struct F {
    char    c;
    short   s;
};

interface Foo {
    F op(in F p_in, inout F p_inout, out F p_out);
};
```

The proxy signature is:

```
typedef F & F_out;
F op(
    const F &    p_in,
    F &          p_inout,
    F_out        p_out
);
```



6.27.2 Rules for Fixed-Length Complex Types

For fixed-length unions and structures, the rules are the same as for simple types, except that `in` parameters are passed by constant reference for efficiency. Note that the compiler generates `<typename>_out` as the parameter type for `out` parameters. For fixed-length types, this is simply an alias to a reference to the type.¹² The client can call the operation as follows:

```
Foo_var fv = ...;    // Get reference...

F in_val = { 'A', 1 };
F inout_val = { 'B', 2 };
F out_val;
F ret_val;

ret_val = fv->op(in_val, inout_val, out_val);

// in_val is unchanged, inout_val may have been changed,
// and out_val and ret_val are filled in by the operation.
```

¹²The `<typename>_out` types are important for variable-length parameters. (See page 6-50.)

Parameter Passing (cont.)

Fixed-length arrays are passed by pointer to an array slice:

```
typedef short SA[2];

interface Foo {
    SA op(in SA p_in, inout SA p_inout, out SA p_out);
};
```

The proxy signature is:

```
typedef SA_slice * SA_out;
SA_slice * op(
    const SA      p_in,
    SA_slice *    p_inout,
    SA_out        p_out
);
```



6.27.3 Rules for Fixed-Length Arrays

Because C++ has a weak array concept, arrays are passed as a pointer to array slice. Note that, even though the formal parameter type in the `in` direction is `SA`, when you pass the array, what will actually be passed is a pointer to the first element (that is, a pointer to an array slice). Also note that, for fixed-length arrays, the `_out` parameter type is simply an alias for a pointer to an array slice. The client can call the operation as follows:

```
Foo_var fv = ...;          // Get reference...

SA in_val = { 1, 2 };
SA inout_val = { 3, 4 };
SA out_val;
SA_slice * ret_val;       // Note pointer to an array slice

ret_val = op(in_val, inout_val, out_val);

// in_val is unchanged, inout_val may have been changed,
// out_val now contains values, and ret_val points
// to a dynamically allocated instance.

SA_free(ret_val);        // Must free here!
```

Note that the return value is dynamically allocated and you must deallocate it after the call. (Of course, you can also use an `SA_var` to catch the return value and let it take care of deallocation.)

Parameter Passing (cont.)

Strings are passed as pointers.

```
interface Foo {
    string op(
        in string      p_in,
        inout string   p_inout,
        out            string p_out
    );
};
```

The proxy signature is:

```
char * op(
    const char *      p_in,
    char * &         p_inout,
    CORBA::String_out p_out
);
```

String_out is a class that behaves (almost) like a `char * &`.



6.27.4 Rules for Strings

Strings are passed as normal `char *` pointers. Note that strings that travel from server to client must be dynamically allocated and the client must deallocate them when they are no longer needed. The client can call the operation as follows:

```
Foo_ref fv = ...;          // Get reference...

// inout strings *must* be dynamically allocated
char * inout_val = CORBA::string_dup("World");

char * out_val;
char * ret_val;

ret_val = fv->op("Hello", inout_val, out_val);

cout << "inout_val: \" << inout_val << \"\" << endl;
cout << "out_val: \" << out_val << \"\" << endl;
cout << "ret_val: \" << ret_val << \"\" << endl;

// Clean up...
CORBA::string_free(inout_val);
CORBA::string_free(out_val);
CORBA::string_free(ret_val);
```

There are several important things to note here:

- The `in` parameter can be stack allocated or heap allocated, and you can even pass a literal. The only thing that is important is that you pass an initialized value.
- The `inout` parameter *must* be dynamically allocated before the call and must be initialized.
- The `inout` parameter *must* be deallocated after the call (because we allocated it).
- The `out` parameter and the return value *must* be deallocated after the call.

Note that the `inout` parameter is passed as a pointer to a reference, not just as a plain pointer. This is necessary because the string that is returned by the server may be longer than the initial string sent by the client. In turn, this implies reallocation, which means that not only the string itself may change, but also its location in memory. To reallocate, the call stub must be able to deallocate the initial value, which requires that the initial value must be dynamically allocated.

Because the `inout` parameter is dynamically allocated, we must deallocate the value when we no longer need it. And, similarly, the `out` parameter and the return value must be deallocated as well because the stub will dynamically allocate these.¹³

Explicit deallocation is tedious, so it is a good idea to use `String_var` to make life easier and less error-prone. Here is the same code example using `String_var`:

```

Foo_ref fv = ...;           // Get reference...

CORBA::String_var inout_val = CORBA::string_dup("World");

CORBA::String_var out_val;
CORBA::String_var ret_val;

ret_val = fv->op("Hello", inout_val, out_val);

cout << "inout_val: \" << inout_val << \"\" << endl;
cout << "out_val: \" << out_val << \"\" << endl;
cout << "ret_val: \" << ret_val << \"\" << endl;

// No deallocation necessary, the String_vars take care of it...

```

NOTE: The mapping prohibits passing of null pointers as `in` or `inout` parameters. The following is guaranteed to result in a crash:

```

CORBA::String_var in_val, inout_val, out_val, ret_val;

ret_val = fv->op(in_val, inout_val, out_val);    // Wrong!

```

This fails because the stub will attempt to dereference a null pointer for `in_val` and `inout_val` and promptly cause a core dump.

Never pass null pointers as `in` or `inout` parameters. This rule holds for *all* types that are passed by pointer.

¹³We will return to the details of the `String_out` type on page 6-50. For now, assume that it is the same as a `char * &`.

Parameter Passing (cont.)

```
interface Foo {
    void get_name(out string name);
};
```

The following code leaks memory:

```
char * name;
fv->get_name(name);
// Should have called CORBA::string_free(name) here!
fv->get_name(name); // Leak!
```

The following code does not:

```
CORBA::String_var name;
fv->get_name(name);
fv->get_name(name); // Fine
```

The `string_out` type takes care of deallocating the first result before passing the parameter to the stub.



6.27.5 Purpose of `_out` Types

As we saw on page 6-48, the caller must deallocate the string out parameter that is allocated by the stub. It follows that the first code example on the above slide leaks memory because the first out parameter is not deallocated before the second call overwrites the name pointer.

On the other hand, if you use `_var` types, no leak occurs because the formal parameter of type `String_out` takes care of deallocating any previous value. Here is an outline of how the `String_out` type works:

```
class String_out {
public:
    String_out(char * & s): _sref(s) { _sref = 0; }
    String_out(String_var & sv): _sref(sv._sref) {
        string_free(_sref);
        _sref = 0;
    }
    // More member functions here...
private:
    char * & _sref;
};
```

Here is the proxy signature for a string out parameter once more:

```
void get_name(CORBA::String_out name);
```

The preceding code examples demonstrate that the effect of calling `get_name` differs depending on whether we pass a `char *` or a `String_var`. Here is what happens if we pass a `char *`:

1. The formal parameter type is `String_out` and the actual argument type is `char *`. The compiler looks for the best argument match and constructs a temporary `String_out` instance using the `char * &` constructor.
2. The `char * &` constructor for `String_out` sets the passed pointer to null without deallocating any memory and binds the passed pointer to its private `char *` reference.
3. The stub for `get_name` gets control and eventually sets the reference inside the `String_out` temporary to a dynamically allocated address. Because the reference is still bound to the actual argument, that assigns to the argument so, when the call completes, the argument has been changed to contain the address of the dynamically allocated string.

If we pass a `String_var` to `get_name`, a slightly different sequence of events takes place:

1. The formal parameter type is `String_out` and the actual argument type is `String_var`. The compiler looks for the best argument match and constructs a temporary `String_out` instance using the `String_var` constructor.
2. The `String_var` constructor binds the pointer contained inside the `String_var` to its private reference and calls `string_free`. This deallocates any previous string that may have been assigned to the `String_var`. It then sets the value of the actual argument to null via the reference.
3. The stub for `get_name` gets control and eventually sets the reference inside the `String_out` temporary to a dynamically allocated address, as in step 3 for the previous scenario.

The net effect is that if you pass a `String_var`, any previously held value is automatically deallocated, whereas if you pass a `char *`, you must deallocate any previous value explicitly.

The compiler generates `_out` types for all variable-length types, such as object references, sequences, etc. In addition, for consistency, `_out` types are generated for fixed-length types as well. However, because no deallocation issues arise for fixed-length out parameters, those `_out` types are simply aliases for a reference to the underlying type.

Parameter Passing (cont.)

Sequences and variable-length structures and unions are dynamically allocated if they are an **out** parameter or the return value.

```
typedef sequence<octet> OctSeq;
interface Foo {
    OctSeq op(
        in OctSeq      p_in,
        inout OctSeq   p_inout,
        out OctSeq     p_out
    );
};
```

The proxy signature is:

```
typedef OctSeq & OctSeq_out;
OctSeq * op( const OctSeq & p_in,
             OctSeq &      p_inout,
             OctSeq_out    p_out);
```



6.27.6 Rules for Variable-Length Complex Types

Sequences and variable-length structures and unions are dynamically allocated if they are an **out** parameter or the return value. Note that for **inout** parameters, no dynamic allocation is necessary. Instead, the stub can modify the parameter via the C++ reference. The client can call the operation as follows:

```
Foo_var fv = ...;           // Get reference...

OctSeq in_val;
OctSeq inout_val;
OctSeq * out_val;         // *Pointer* to OctSeq
OctSeq * ret_val;        // *Pointer* to OctSeq

in_val.length(1);
in_val[0] = 99;
inout_val.length(2);
inout_val[0] = 5;
inout_val[1] = 6;

ret_val = fv->op(in_val, inout_val, out_val);

// inout_val may have been modified, out_val and
```

```

// ret_val point to now initialized sequences

delete out_val;           // Must deallocate here!
delete ret_val;          // Must deallocate here!

```

Note that the `out` parameter and the return value require deallocation.

As before, using `_var` types makes life considerably easier:

```

Foo_var fv = ...;        // Get reference...

OctSeq in_val;
OctSeq inout_val;
OctSeq_var out_val;     // _var type
OctSeq_var ret_val;     // _var type

in_val.length(1);
in_val[0] = 99;
inout_val.length(2);
inout_val[0] = 5;
inout_val[1] = 6;

ret_val = fv->op(in_val, inout_val, out_val);

// out_val and ret_val will eventually deallocate...

```

When `out_val` and `ret_val` go out of scope, they will call `delete` on the pointers that were returned by the call to `op`. You can use `_var` types for `in` and `inout` parameters as well.

However, doing this offers not much of an advantage because no memory management issues arise in this case (but the use of a `_var` types still requires dynamic allocation):

```

Foo_var fv = ...;        // Get reference...

OctSeq_var in_val(new OctSeq);
OctSeq_var inout_val(new OctSeq);
OctSeq_var out_val;
OctSeq_var ret_val;

in_val->length(1);
in_val[0] = 99;
inout_val->length(2);
inout_val[0] = 5;
inout_val[1] = 6;

ret_val = fv->op(in_val, inout_val, out_val);

// ...

```

This code is correct but slightly less efficient because of the unnecessary allocation for the `in` and `inout` parameter.

Parameter Passing (cont.)

Variable-length **out** arrays and return values are dynamically allocated.

```
struct Employee {
    string name;
    long number;
};
typedef Employee EA[2];

interface Foo {
    EA op(in EA p_in, inout EA p_inout, out EA p_out);
};
```

The proxy signature is:

```
EA_slice * op(
    const EA p_in,
    EA_slice * p_inout,
    EA_out p_out
);
```



38
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.27.7 Rules for Variable-Length Arrays

As for variable-length complex types, variable-length arrays are dynamically allocated if they are out parameters or the return value and must be deallocated by the caller. The client can call the operation as follows:

```
Foo_var fv = ...; // Get reference...

EA in_val;
in_val[0].name = CORBA::string_dup("Michi");
in_val[0].number = 1;
in_val[1].name = CORBA::string_dup("Steve");
in_val[1].number = 2;

EA inout_val;
inout_val[0].name = CORBA::string_dup("Bjarne");
inout_val[0].number = 3;
inout_val[1].name = CORBA::string_dup("Stan");
inout_val[1].number = 4;

EA_slice * out_val; // Note pointer to slice
EA_slice * ret_val; // Note pointer to slice

ret_val = fv->op(in_val, inout_val, out_val);
// in_val is unchanged, inout_val may have been changed,
```



```
// out_val and ret_val point at dynamically allocated arrays
```

```
EA_free(out_val);          // Must free here!
```

```
EA_free(ret_val);         // Must free here!
```

As for other variable-length types, use of `_var` types makes the explicit deallocation unnecessary:

```
Foo_var fv = ...;         // Get reference...
```

```
EA in_val;
```

```
// Initialize in_val...
```

```
EA inout_val;
```

```
// Initialize inout_val...
```

```
EA_var out_val;          // _var type
```

```
EA_var ret_val;          // _var type
```

```
ret_val = fv->op(in_val, inout_val, out_val);
```

```
// in_val is unchanged, inout_val may have been changed,
```

```
// out_val and ret_val point at dynamically allocated arrays
```

```
// No need for explicit deallocation here...
```

Parameter Passing (cont.)

Object reference **out** parameters and return values are duplicated.

```
interface Thermometer { /* ... */ };

interface Foo {
    Thermometer op(
        in Thermometer    p_in,
        inout Thermometer p_inout,
        out Thermometer    p_out
    );
};
```

The proxy signature is:

```
Thermometer_ptr op(
    Thermometer_ptr    p_in,
    Thermometer_ptr & p_inout,
    Thermometer_out    p_out
);
```



39
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.27.8 Rules for Object References

For object references, the same rules apply as for strings: the client must deallocate **out** parameters and return values. The client can call the operation as follows:

```
Foo_var fv = ...; // Get reference...

Thermometer_ptr in_val = ...; // Initialize in param
Thermometer_ptr inout_val = ...; // Initialize inout param
Thermometer_ptr out_val;
Thermometer_ptr ret_val;

ret_val = fv->op(in_val, inout_val, out_val);

CORBA::release(in_val);
CORBA::release(inout_val);
CORBA::release(out_val);
CORBA::release(ret_val);
```

As with the previous examples, using `_var` types makes life considerable easier and safer because you cannot accidentally forget to release a reference:

```
Foo_var fv = ...;           // Get reference...

Thermometer_var in_val = ...; // Initialize in param
Thermometer_var inout_val = ...; // Initialize inout param
Thermometer_var out_val;
Thermometer_var ret_val;

ret_val = fv->op(in_val, inout_val, out_val);

// No releases necessary here.
```

Parameter Passing: Pitfalls

Stick to the following rules when invoking operations:

- Always initialize **in** and **inout** parameters .
- Do not pass **in** or **inout** null pointers.
- Deallocate **out** parameters that are passed by pointer.
- Deallocate variable-length return values.
- Do not ignore the return value from an operation that returns a variable-length value.
- Use `_var` types to make life easier for yourself.



40
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.28 Parameter Passing: Pitfalls

The above slide summarizes a few of the pitfalls that you may encounter until you are familiar with the parameter passing rules.

Obviously, you must initialize any **in** and **inout** parameters you pass to an operation. Failure to do so is asking for trouble: for a simple type, you will be passing a garbage value to the server, and for a type passed by pointer (such as a string), you will be passing an uninitialized pointer (which is likely to cause a core dump when the stub dereferences the pointer in order to marshal the string).

The C++ mapping makes it illegal to pass a null pointer as an **in** or **inout** parameter.¹⁴ Failure to obey this rules will cause a core dump in the stub when it attempts to marshal the parameter.

As you saw in the preceding sections, you must deallocate variable-length return values and out parameters that are passed by pointer. Failure to do so results in a resource leak. The most frequent cause of leaks is code like the following:

```
// Assume IDL:
// interface Foo {
//     string get_name();
//     void set_name(in string n);
// };
```

```
Foo_var fv = ...;
```

¹⁴For nil object references, it is OK to pass a nil reference, even if nil references happen to be implemented as null pointers. However, you cannot pass, for example, a null pointer as an **in** string.

```
cout << fv->get_name() << endl;           // Leak!

CORBA::String_var s = fv->get_name();     // Better
cout << s << endl;

// Or use:
cout << CORBA::String_var(fv->get_name()) << endl;
```

The use of a `String_var` temporary prevents this kind of problem. Another common mistake is to write code like the following:

```
Foo_var fv1 = ...;
Foo_var fv2 = ...;

fv1->set_name(fv2->get_name());           // Leak!

CORBA::String_var tmp = fv2->get_name();
fv1->set_name(tmp);                       // Better!

// Or use:
fv1->set_name(CORBA::String_var(fv2->get_name()));
```

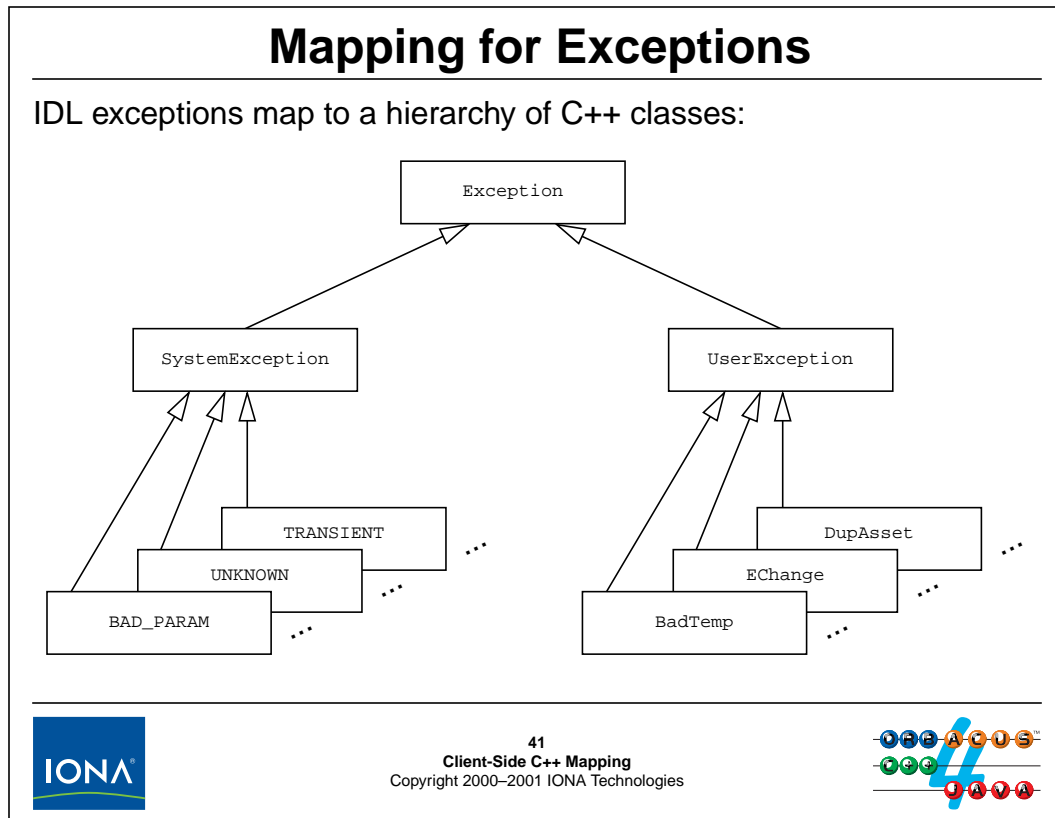
The following table summarizes the parameter passing rules. Note that the formal type of `out` parameters is `<typename>_out`. However, apart from the transparent deallocation for `_var` parameters done by `_out` types, they behave as if they were of the type shown.

IDL Type	in	inout	out	Return Type
<i>simple</i>	<i>simple</i>	<i>simple</i> &	<i>simple</i> &	<i>simple</i>
<i>enum</i>	<i>enum</i>	<i>enum</i> &	<i>enum</i> &	<i>enum</i>
<i>fixed</i>	const <i>Fixed</i> &	<i>Fixed</i> &	<i>Fixed</i> &	<i>Fixed</i>
<i>string</i>	const char *	char * &	char * &	char *
<i>wstring</i>	const WCHAR *	WCHAR * &	WCHAR * &	WCHAR *
<i>any</i>	const Any &	Any &	Any * &	Any *
<i>objref</i>	<i>objref_ptr</i>	<i>objref_ptr</i> &	<i>objref_ptr</i> &	<i>objref_ptr</i>
<i>sequence</i>	const <i>sequence</i> &	<i>sequence</i> &	<i>sequence</i> * &	<i>sequence</i> *
<i>struct, fixed</i>	const <i>struct</i> &	<i>struct</i> &	<i>struct</i> &	<i>struct</i>
<i>union, fixed</i>	const <i>union</i> &	<i>union</i> &	<i>union</i> &	<i>union</i>
<i>array, fixed</i>	const <i>array</i>	<i>array_slice</i> *	<i>array_slice</i> *	<i>array_slice</i> *
<i>struct, variable</i>	const <i>struct</i> &	<i>struct</i> &	<i>struct</i> * &	<i>struct</i> *
<i>union, variable</i>	const <i>union</i> &	<i>union</i> &	<i>union</i> * &	<i>union</i> *
<i>array, variable</i>	const <i>array</i>	<i>array_slice</i> *	<i>array_slice</i> * &	<i>array_slice</i> *

If you use `_var` types to pass parameters, the situation gets considerably simpler:

IDL Type	in	inout/out	Return Type
<i>string</i>	const <i>String_var</i> &	<i>String_var</i> &	<i>String_var</i>
<i>wstring</i>	const <i>WString_var</i> &	<i>WString_var</i> &	<i>WString_var</i>
<i>any</i>	const <i>Any_var</i> &	<i>Any_var</i> &	<i>Any_var</i>
<i>objref</i>	const <i>objref_var</i> &	<i>objref_var</i> &	<i>objref_var</i>
<i>sequence</i>	const <i>sequence_var</i> &	<i>sequence_var</i> &	<i>sequence_var</i>
<i>struct</i>	const <i>struct_var</i> &	<i>struct_var</i> &	<i>struct_var</i>
<i>union</i>	const <i>union_var</i> &	<i>union_var</i> &	<i>union_var</i>
<i>array</i>	const <i>array_var</i> &	<i>array_var</i> &	<i>array_var</i>
<i>string</i>	const <i>String_var</i> &	<i>String_var</i> &	<i>String_var</i>
<i>wstring</i>	const <i>WString_var</i> &	<i>WString_var</i> &	<i>WString_var</i>
<i>any</i>	const <i>Any_var</i> &	<i>Any_var</i> &	<i>Any_var</i>
<i>objref</i>	const <i>objref_var</i> &	<i>objref_var</i> &	<i>objref_var</i>
<i>sequence</i>	const <i>sequence_var</i> &	<i>sequence_var</i> &	<i>sequence_var</i>

Note that `_var` types do work as `in` parameters. This is occasionally useful if you want to pass an `out` `_var` parameter you received from one call as an `in` `_var` parameter to another call. However, keep in mind that `_var` types are useful mainly for `inout` and `out` parameters and return values. Do not use a `_var` type purely for a local variable. While it will work, it is wasteful due to the need to initialize `_var` types with dynamically-allocated memory. If all you need is a local variable, it is easier and more efficient to simply use a stack-allocated instance.



6.29 Mapping for Exceptions

Up to now, we have ignored how to deal with error conditions. Even though CORBA does its best to make a remote procedure call appear like a local one, the reality of networking means that a remote invocation is more likely to fail than a local function call. Such failures are indicated by system exceptions. In addition, user exceptions can be raised by operations to indicate application-specific errors.

The C++ mapping arranges exceptions into the hierarchy shown above. The base classes `CORBA::Exception`, `CORBA::SystemException`, and `CORBA::UserException` are abstract and simply encapsulate common functionality. In addition, because all exceptions derive from `CORBA::UserException`, you can catch all CORBA exceptions in a single generic catch handler, which is useful.

Specific system exceptions are derived from `SystemException`, whereas specific user exceptions are all derived from `UserException`.

Mapping for Exceptions (cont.)

The exception hierarchy looks like this:

```
namespace CORBA {
  class Exception { // Abstract
  public:
    virtual ~Exception();
    virtual const char * _name() const;
    virtual const char * _rep_id() const;
    virtual void _raise() = 0;
  };
  class SystemException : public Exception { // Abstract
    // ...
  };
  class UserException : public Exception { // Abstract
    // ...
  };
  class UNKNOWN : public SystemException { /* ... */ }; // Concrete
  class FREE_MEM : public SystemException { /* ... */ }; // Concrete
  // etc...
};
```



42
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



The above slide shows the exception hierarchy as defined in the CORBA namespace. The `Exception` base class (and therefore all derived exceptions) contains three member functions, `_name`, `_rep_id`, and `_raise`.

- The `_name` member function returns the name of the exception, such as "BAD_PARAM".
- The `_rep_id` member function returns the repository ID of an exception, such as "IDL:omg.org/CORBA/BAD_PARAM:1.0". You will rarely have a need to call these members directly. They are present mainly for debugging and as an aid to implementing the ostream inserter for exceptions.
- The `_raise` member function is rapidly becoming obsolete. It was originally added to support C++ compilers without exception handling, so we will not discuss it further.

Note that all concrete system exceptions derive from `SystemException` and all concrete user exceptions derive from `UserException`.

Given these definitions, a client can implement error handling at the desired level of granularity. For example, here is some code that might be used when calling the `set_nominal` operation on a thermostat:

```
CCS::Thermostat_var ts = ...;
CCS::TempType temp = ...;

try {
  ts->set_nominal(temp);
} catch (const CCS::Thermostat::BadTemp &) {
  // New temperature out of range
```



```
    } catch (const CORBA::UserException & e) {
        // Some other user exception was raised
        cerr << "User exception: " << e << endl;
    } catch (const CORBA::OBJECT_NOT_EXIST &) {
        cerr << "No such thermostat" << endl;
    } catch (const CORBA::SystemException & e) {
        // Some other system exception
        cerr << "System exception: " << e << endl;
    } catch (...) {
        // Some non-CORBA exception -- should never happen
    }
}
```

This code handles exceptions in detail, by catching specific user and system exceptions as well as handling other user and system exceptions generically.

Note that the final `catch` handler should never run because a compliant ORB will not throw anything but CORBA exceptions. Usually, the best thing to do is to have a generic `catch` handler for “impossible” exceptions fairly high up in your call hierarchy; that handler can print a message and terminate the program.

Also note that you can insert both user and system exceptions into an `ostream`. This works because the mapping provides an overloaded `<<` operator for `CORBA::Exception`. The precise formatting of the string that is printed is implementation defined. Typically, it will print the name of the exception and, for system exceptions, the completion status and minor code. You can change the formatting of the string by writing overloaded `ostream` inserters for `SystemException` and `UserException` (or even writing overloaded `ostream` inserters for specific exceptions).

Typically, you will not handle exceptions in this much detail for every call. Instead, calls will typically only handle one or two error conditions of interest and permit other exceptions to percolate back up the call hierarchy, where they can be handled generically.

Note that the code catches exceptions by constant reference rather than by non-constant value. This is more efficient because the compiler can eliminate the need to copy the exception into a temporary. It is also much better if you rethrow an exception because, if you catch and throw it by a base reference, the exception won't be sliced if its actual type is more derived.

Mapping for System Exceptions

All system exceptions derive from the `SystemException` base class:

```
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);
    // ...
};
```



43
Client-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



6.30 Mapping for System Exceptions

The `SystemException` class offers the usual default constructor, copy constructor, assignment operator, and destructor. (The default constructor creates a system exception with a minor code of 0 and a completion status of `COMPLETED_NO`.) In addition, a two-parameter version of the constructor permits you to initialize a system exception during construction (instead of having to first default construct it and then set the members).

The `minor` and `completed` member functions permit you to read and write the minor code and the completion status. The actual (concrete) system exceptions are also defined in the `CORBA` namespace and simply defined by derivation from `SystemException`.

Semantics of System Exceptions

The standard defines semantics for system exceptions. For some exceptions, the semantics are defined only in broad terms (such as **INTERNAL** or **NO_MEMORY**).

The most commonly encountered system exceptions are:

OBJECT_NOT_EXIST, **TRANSIENT**, **BAD_PARAM**, **COMM_FAILURE**, **IMP_LIMIT**, **NO_MEMORY**, **UNKNOWN**, **NO_PERMISSION**, and **NO_RESOURCES**.

The specification defines minor codes for some exceptions to provide more detailed information on a specific error. Standard minor codes are allocated in the range **0x4f4d0000–0x4f4d0fff**.

Vendors can allocate a block of minor code values for their own use. For ORBacus-specific minor codes, the allocated range is **0x4f4f0000–0x4f4f0fff**.



6.31 Semantics of System Exceptions

The specification defines the meaning of the various system exceptions. For some exceptions, because of their general nature, the specification only outlines in general terms when they are to be raised. For other exceptions, the specification is quite specific. Above is a list of the system exceptions you are most likely to see, so it is useful to know their meaning.

- **OBJECT_NOT_EXIST**

This exception is an authoritative indication that the reference for the request is stale (denotes a non-existent object). If you receive this exception, you can safely conclude that the reference to the object is permanently non-functional and therefore you should clean up any application resources (such as database entries) you may have for that object.

- **TRANSIENT**

TRANSIENT indicates that the ORB attempted to reach the server and failed. It is not an indication that the server or the object does not exist. Instead, it simply means that no further determination of an object's status was possible because it could not be reached. **TRANSIENT** is typically raised if connectivity to the server cannot be established—things may work if you try again later.

- **BAD_PARAM**

A parameter passed to a call is out of range or otherwise considered illegal. Some ORBs raise this exception if you pass a null pointer to an operation.

- **COMM_FAILURE**

This exception is raised if communication is lost while an operation is in progress. At the protocol level, the client sends a request to the server and then waits for a reply containing the

results. If the connection drops after the client has sent the request but before the reply has arrived, the client-side run time raises `COMM_FAILURE`.

- `IMP_LIMIT`

This exception indicates that an implementation limit was exceeded in the ORB run time. There are a variety of reasons for this exception. For example, you may have reached the maximum number of references you can hold simultaneously in your address space, the size of a parameter may have exceeded the allowed maximum, or your ORB may impose a maximum on the number of clients or servers that can run simultaneously. Your ORB's documentation should provide more detail about such limits.

- `NO_MEMORY`

The ORB run time ran out of memory at some stage during a call. You can check the completion status to see whether it happened before or after the operation was invoked in the server.

- `UNKNOWN`

This exception is raised if an operation implementation raises a non-CORBA exception or if an operation raises a user exception that does not appear in the operation's `raises` expression. `UNKNOWN` is also raised if the server returns a system exception that is unknown to the client. This can happen if the server uses a later version of CORBA than the client and if new system exceptions have been added to the later version.

- `NO_PERMISSION`

This exception can be raised by ORBs that provide a Security Service if the caller has insufficient privileges to invoke an operation.

- `NO_RESOURCES`

The ORB has encountered a general resource limitation. For example, the run time may have reached the maximum permissible number of open connections.

If you receive a system exception, its minor code may be non-zero and provide additional information about the exception. You cannot use the minor code programmatically (at least not easily) because the C++ mapping does not provide symbolic constants or string error messages for minor codes. However, the minor code is still useful for debugging and logging.

Mapping for User Exceptions

User exceptions map to a class with public members:

```
exception Boom {
    string msg;
    short  errno;
};
```

This generates:

```
class Boom : public CORBA::UserException {
    Boom();
    Boom(const char*, CORBA::Short);
    Boom(const Boom &);
    Boom& operator=(const Boom &);
    ~Boom();

    OB::StrForStruct msg;
    CORBA::Short      errno;
};
```



6.32 Mapping for User Exceptions

The mapping for user exceptions is the same as the one for structures, with the addition of an extra constructor. User exceptions map exactly like structures, except that they have an additional constructor. As you can see in the example above, the two members of the exception map to public data members, and the constructor accepts two parameters, one for each member. The reason for the additional constructor is so that you can initialize and throw the exception in a single statement:

```
if (something_failed)
    throw Boom("Something failed", 99);
```

(On the client side, you will rarely (if ever) throw CORBA exceptions. However, the same mapping applies to the server side, where the functionality is useful.)

The code to catch and handle user exceptions looks very similar to the code to handle system exceptions and accessing a structure:

```
try {
    some_ref->some_op();
} catch (const Boom & b) {
    cerr << "Boom: " << b.msg << " (" << b.errno << ")" << endl;
}
```

Compiling and Linking

To create a client executable, you must compile the application code and the stub code. Typical compile commands are:

```
c++ -I. -I/opt/OB/include -c client.cc
c++ -I. -I/opt/OB/include -c MyIDL.cpp
```

The exact flags and compile command vary with the platform.

To link the client, you must link against `libOB`:

```
c++ -o client client.o MyIDL.o -L/opt/OB/lib -lOB
```

If you are using JThreads/C++, you also need to link against the JTC library and the native threads library. For example:

```
c++ -o client client.o MyIDL.o -L/opt/OB/lib \
-lOB -lJTC -lpthread
```



6.33 Compiling and Linking

The exact compile and link commands you need to use to link an executable depend on the platform. The above examples show compile and link commands for GNU C++ under Linux.

Assuming that ORBacus is installed in `/opt/OB`, the include files are under `/opt/OB/include` in separate hierarchies. For example, the ORBacus include files are in `/opt/OB/include/OB`, whereas the JThreads/C++ include files are in `/opt/OB/include/JTC`. Include directives in the installed and generated header files are always relative to the installation root. For example, ORBacus header files use includes of the form:

```
#include <OB/CORBA.h>
#include <OB/JTC.h>
```

This means that a single `-I/opt/OB/include` directive is sufficient to locate the installed header files. Because the IDL-generated header files are typically in the current directory, you also need add a `-I.` directive.

To link the client, you must link the compiled application code and the compiled stubs. The ORBacus run-time support is provided in `libOB` and threads support (if you use JThreads/C++) is provided in `libJTC` (which in turn depends on the native threads library for your platform).

7. Exercise: Writing a Client

Summary

In this unit, you will develop a simple CORBA client that communicates with a server that is provided for you.

Objectives

By the completion of this unit, you will be able to build CORBA clients that read or write attributes, invoke operations, and handle CORBA exceptions.

7.1 Source Files and Build Environment

In your `client` directory you will find the following files:

- `Makefile`

Use this makefile to compile the code. All the relevant targets have been provided for you, so you do not need to change this file. The targets are:

- `all`
This target builds both client and server executables.
- `client`
This target builds the client executable.
- `server`
This target builds the server executable.
- `clean`
This target removes all intermediate files and the files that are generated by the server when it runs.
- `clobber`
This target does the same thing as `clean` but also removes the `client` and `server` executable files.
- `CCS.idl`
This file contains the IDL for the Climate Control System we presented in Unit 4. You do not need to change this file.
- `server.h`, `server.cpp`
These are the source files for the server you will communicate with. You do not need to change these files.
- `client.cpp`
This file contains the source code for the client. You will need to modify this file as part of the exercise.

7.2 Server Operation

The server creates a single controller object and a number of thermometer and thermostat objects. On start-up, the server writes two stringified object references into files in the current directory:

- `tmstat.ref`
This file contains the stringified reference for a thermostat object.
- `ctrl.ref`
This file contains the stringified reference to the controller object.

To start the server, use

```
./server &
```

to run the server in the background or, alternatively, run the server in a different window in the foreground. (Make sure that the server's working directory is your `client` directory.)

7.3 Client Operation

The client executable expects the stringified references for the thermostat and the controller in `argv[1]` and `argv[2]`. You can start the client as follows:

```
./client file:///home/michi/labs/client/tmstat.ref \
file:///home/michi/labs/client/ctrl.ref
```

The client first uses the thermostat reference to invoke a few operations on the thermostat, and then uses the controller reference to use the more complex operations on the controller.

7.4 What You Need to Do

Much of the client is already implemented for you, so you do not have to waste time on the tedious (and not very instructive) chores of finding include files, setting compiler options, and so on. Instead, you will be focussing on the more interesting aspects of the client-side C++ mapping.

The places in the client code are marked with a

```
// MISSING, step X
```

comment, so you know where you are expected to make changes.

NOTE: If you find yourself trying to read the header files that are generated by the IDL compiler in order to work out what functions to call and what arguments to pass, step back and think again. The header files are largely incomprehensible and not meant for human consumption. In order to decide what to do, look at the IDL and the C++ mapping rules. This will get you to your goal much quicker.

7.4.1 Communicating with the Thermostat

1. Get the stringified thermostat reference from `argv[1]` and unstringify it.
2. The client code contains an overloaded operator to display the details of a device:

```
static ostream &
operator<<(ostream os, CCS::Thermometer_ptr t)
```

With this operator, you can display the details of a thermometer or thermostat reference by inserting the reference into a stream:

```
CCS::Thermometer_var tmstat = ...;
cout << tmstat << endl;
```

Implement the body of this operator. You should display whether the inserted reference is for thermometer or thermostat, show the asset number, model, location, and current temperature, and, if the device is thermostat, show the setting of the nominal temperature.

3. Use the inserter you just created to display the details of the thermostat reference you unstringified in step 1.
4. Change the temperature of the thermostat to a valid temperature. (Use room temperature with a Fahrenheit scale.) Read back the setting you just changed to convince yourself that it actually worked.

5. The client code contains an overloaded operator to display the details of a `BtData` structure on a stream:

```
static ostream &
operator<<(ostream & os, const CCS::Thermostat::BtData & btd)
```

Implement the body of this operator.

6. Change the setting of the thermostat to an illegal value (outside room temperature range). Verify that the setting fails and that an exception is thrown. Display the details of the exception using the operator you implemented in step 5.

7.4.2 Communicating with the Controller

7. Get the stringified controller reference from `argv[2]` and unstringify it.
8. Get the list of devices from the controller. Display the number of devices and the details for each device.
9. Search the CCS for devices in rooms "Earth" and "HAL". Show the devices found in these rooms.
10. The client code contains an overloaded operator to display the details of an `EChange` exception:

```
static ostream &
operator<<(ostream & os, const CCS::Controller::EChange & ec)
```

Implement the body of this operator.

11. Increase the temperature of all thermostats by 40 degrees. (To do this, you will have to somehow get all thermostats first, because you cannot set the temperature of a thermometer.) Some thermostats will raise an exception in response to the change. Display the details of the exception that is returned using the operator you implemented in step 10.

8. Solution: Writing a Client

8.1 Communicating with the Thermostat

Step 1

```
// Get thermostat reference from argv[1]
// and convert to object.
CORBA::Object_var obj = orb->string_to_object(argv[1]);
if (CORBA::is_nil(obj)) {
    cerr << "Nil thermostat reference" << endl;
    throw 0;
}

// Try to narrow to CCS::Thermostat.
CCS::Thermostat_var tmstat;
try {
    tmstat = CCS::Thermostat::_narrow(obj);
} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow thermostat reference: "
         << se << endl;
    throw 0;
}
if (CORBA::is_nil(tmstat)) {
    cerr << "Wrong type for thermostat ref." << endl;
    throw 0;
}
```

Step 2

```
// Show the details for a thermometer or thermostat.

static ostream &
operator<<(ostream & os, CCS::Thermometer_ptr t)
{
    // Check for nil.
    if (CORBA::is_nil(t)) {
        os << "Cannot show state for nil reference." << endl;
        return os;
    }

    // Try to narrow and print what kind of device it is.
    CCS::Thermostat_var tmstat = CCS::Thermostat::_narrow(t);
    os << (CORBA::is_nil(tmstat)?"Thermometer:":"Thermostat:")
        << endl;

    // Show attribute values.
    CCS::ModelType_var model = t->model();
    CCS::LocType_var location = t->location();
    os << "\tAsset number: " << t->asset_num() << endl;
    os << "\tModel          : " << model << endl;
    os << "\tLocation         : " << location << endl;
}
```

```

    os << "\tTemperature : " << t->temperature() << endl;

    // If device is a thermostat, show nominal temperature.
    if (!CORBA::is_nil(tmstat))
        os << "\tNominal temp: " << tmstat->get_nominal() << endl;
    return os;
}

```

Step 3

```

// Show details of thermostat
cout << tmstat << endl;

```

Step 4

```

// Change the temperature of the thermostat to a valid
// temperature.
cout << "Changing nominal temperature" << endl;
CCS::TempType old_temp = tmstat->set_nominal(60);
cout << "Nominal temperature is now 60, previously "
    << old_temp << endl << endl;

cout << "Retrieving new nominal temperature" << endl;
cout << "Nominal temperature is now "
    << tmstat->get_nominal() << endl << endl;

```

Step 5

```

// Show the information in a BtData struct.

static ostream &
operator<<(ostream & os, const CCS::Thermostat::BtData & btd)
{
    os << "CCS::Thermostat::BtData details:" << endl;
    os << "\trequested      : " << btd.requested << endl;
    os << "\tmin_permitted: " << btd.min_permitted << endl;
    os << "\tmax_permitted: " << btd.max_permitted << endl;
    os << "\terror_msg      : " << btd.error_msg << endl;
    return os;
}

```

Step 6

```

// Change the temperature to an illegal value and
// show the details of the exception that is thrown.
cout << "Setting nominal temperature out of range" << endl;
bool got_exception = false;
try {
    tmstat->set_nominal(10000);
} catch (const CCS::Thermostat::BadTemp & e) {
    got_exception = true;
}

```

```

        cout << "Got BadTemp exception: " << endl;
        cout << e.details << endl;
    }
    if (!got_exception)
        assert("Did not get exception");

```

8.2 Communicating with the Controller

Step 7

```

// Get controller reference from argv[2]
// and convert to object.
obj = orb->string_to_object(argv[2]);
if (CORBA::is_nil(obj)) {
    cerr << "Nil controller reference" << endl;
    throw 0;
}

// Try to narrow to CCS::Controller.
CCS::Controller_var ctrl;
try {
    ctrl = CCS::Controller::_narrow(obj);
} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow controller reference: "
         << se << endl;
    throw 0;
}
if (CORBA::is_nil(ctrl)) {
    cerr << "Wrong type for controller ref." << endl;
    throw 0;
}

```

Step 8

```

// Get list of devices
CCS::Controller::ThermometerSeq_var list = ctrl->list();

// Show number of devices.
CORBA::ULong len = list->length();
cout << "Controller has " << len << " device";
if (len != 1)
    cout << "s";
cout << "." << endl;

// Show details for each device.
CORBA::ULong i;
for (i = 0; i < len; ++i)
    cout << list[i].in();
cout << endl;

```


Step 9

```
// Look for device in Rooms Earth and HAL.
cout << "Looking for devices in Earth and HAL." << endl;
CCS::Controller::SearchSeq ss;
ss.length(2);
ss[0].key.loc(CORBA::string_dup("Earth"));
ss[1].key.loc(CORBA::string_dup("HAL"));
ctrl->find(ss);

// Show the devices found in that room.
for (i = 0; i < ss.length(); ++i)
    cout << ss[i].device.in();    // Overloaded <<
cout << endl;
```

Step 10

```
// Loop over the sequence of records in an EChange exception and
// show the details of each record.
```

```
static ostream &
operator<<(ostream & os, const CCS::Controller::EChange & ec)
{
    CORBA::ULong i;
    for (i = 0; i < ec.errors.length(); ++i) {
        os << "Change failed:" << endl;
        os << ec.errors[i].tmstat_ref.in(); // Overloaded <<
        os << ec.errors[i].info << endl;    // Overloaded <<
    }
    return os;
}
```

Step 11

```
// Increase the temperature of all thermostats
// by 40 degrees. First, make a new list (tss)
// containing only thermostats.
cout << "Increasing thermostats by 40 degrees." << endl;
CCS::Thermostat_var ts;
CCS::Controller::ThermostatSeq tss;
for (i = 0; i < list->length(); ++i) {
    ts = CCS::Thermostat::_narrow(list[i]);
    if (CORBA::is_nil(ts))
        continue; // Skip thermometers
    len = tss.length();
    tss.length(len + 1);
    tss[len] = ts;
}

// Try to change all thermostats.
try {
```

```

    ctrl->change(tss, 40);
} catch (const CCS::Controller::EChange & ec) {
    cerr << ec;                // Overloaded <<
}

```

8.3 The Complete Client Code

For your reference, the complete client code is shown below.

```

#include <OB/CORBA.h>
#include <assert.h>

#if defined(HAVE_STD_Iostream)
using namespace std;
#endif

#include "CCS.h"

//-----

// Show the details for a thermometer or thermostat.

static ostream &
operator<<(ostream & os, CCS::Thermometer_ptr t)
{
    // Check for nil.
    if (CORBA::is_nil(t)) {
        os << "Cannot show state for nil reference." << endl;
        return os;
    }

    // Try to narrow and print what kind of device it is.
    CCS::Thermostat_var tmstat = CCS::Thermostat::_narrow(t);
    os << (CORBA::is_nil(tmstat)?"Thermometer:":"Thermostat:")
        << endl;

    // Show attribute values.
    CCS::ModelType_var model = t->model();
    CCS::LocType_var location = t->location();
    os << "\tAsset number: " << t->asset_num() << endl;
    os << "\tModel      : " << model << endl;
    os << "\tLocation    : " << location << endl;
    os << "\tTemperature : " << t->temperature() << endl;

    // If device is a thermostat, show nominal temperature.
    if (!CORBA::is_nil(tmstat)) {
        os << "\tNominal temp: "
            << tmstat->get_nominal() << endl;
    }
    return os;
}

```

```

}

//-----

// Show the information in a BtData struct.

static ostream &
operator<<(ostream & os, const CCS::Thermostat::BtData & btd)
{
    os << "CCS::Thermostat::BtData details:" << endl;
    os << "\trequested      : " << btd.requested << endl;
    os << "\tmin_permitted: " << btd.min_permitted << endl;
    os << "\tmax_permitted: " << btd.max_permitted << endl;
    os << "\terror_msg      : " << btd.error_msg << endl;
    return os;
}

//-----

// Loop over the sequence of records in an EChange exception and
// show the details of each record.

static ostream &
operator<<(ostream & os, const CCS::Controller::EChange & ec)
{
    CORBA::ULong i;
    for (i = 0; i < ec.errors.length(); ++i) {
        os << "Change failed:" << endl;
        os << ec.errors[i].tmstat_ref.in(); // Overloaded <<
        os << ec.errors[i].info << endl;   // Overloaded <<
    }
    return os;
}

//-----

int
main(int argc, char * argv[])
{
    int status = 0;
    CORBA::ORB_var orb;

    try {
        // Initialize ORB and check arguments.
        orb = CORBA::ORB_init(argc, argv);
        if (argc != 3) {
            cerr << "Usage: client IOR IOR" << endl;
            throw 0;
        }
    }

```

```
// Get thermostat reference from argv[1]
// and convert to object.
CORBA::Object_var obj = orb->string_to_object(argv[1]);
if (CORBA::is_nil(obj)) {
    cerr << "Nil thermostat reference" << endl;
    throw 0;
}

// Try to narrow to CCS::Thermostat.
CCS::Thermostat_var tmstat;
try {
    tmstat = CCS::Thermostat::_narrow(obj);
} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow thermostat reference: "
         << se << endl;
    throw 0;
}
if (CORBA::is_nil(tmstat)) {
    cerr << "Wrong type for thermostat ref." << endl;
    throw 0;
}

// Show details of thermostat
cout << tmstat << endl;

// Change the temperature of the thermostat to a valid
// temperature.
cout << "Changing nominal temperature" << endl;
CCS::TempType old_temp = tmstat->set_nominal(60);
cout << "Nominal temperature is now 60, previously "
     << old_temp << endl << endl;

cout << "Retrieving new nominal temperature" << endl;
cout << "Nominal temperature is now "
     << tmstat->get_nominal() << endl << endl;

// Change the temperature to an illegal value and
// show the details of the exception that is thrown.
cout << "Setting nominal temperature out of range"
     << endl;
bool got_exception = false;
try {
    tmstat->set_nominal(10000);
} catch (const CCS::Thermostat::BadTemp & e) {
    got_exception = true;
    cout << "Got BadTemp exception: " << endl;
    cout << e.details << endl;
}
if (!got_exception)
    assert("Did not get exception");
```

```
// Get controller reference from argv[2]
// and convert to object.
obj = orb->string_to_object(argv[2]);
if (CORBA::is_nil(obj)) {
    cerr << "Nil controller reference" << endl;
    throw 0;
}

// Try to narrow to CCS::Controller.
CCS::Controller_var ctrl;
try {
    ctrl = CCS::Controller::_narrow(obj);
} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow controller reference: "
         << se << endl;
    throw 0;
}
if (CORBA::is_nil(ctrl)) {
    cerr << "Wrong type for controller ref." << endl;
    throw 0;
}

// Get list of devices
CCS::Controller::ThermometerSeq_var list = ctrl->list();

// Show number of devices.
CORBA::ULong len = list->length();
cout << "Controller has " << len << " device";
if (len != 1)
    cout << "s";
cout << "." << endl;

// Show details for each device.
CORBA::ULong i;
for (i = 0; i < len; ++i)
    cout << list[i].in();
cout << endl;

// Look for device in Rooms Earth and HAL.
cout << "Looking for devices in Earth and HAL." << endl;
CCS::Controller::SearchSeq ss;
ss.length(2);
ss[0].key.loc(CORBA::string_dup("Earth"));
ss[1].key.loc(CORBA::string_dup("HAL"));
ctrl->find(ss);

// Show the devices found in that room.
for (i = 0; i < ss.length(); ++i)
    cout << ss[i].device.in();    // Overloaded <<
```

```
    cout << endl;

    // Increase the temperature of all thermostats
    // by 40 degrees. First, make a new list (tss)
    // containing only thermostats.
    cout << "Increasing thermostats by 40 degrees." << endl;
    CCS::Thermostat_var ts;
    CCS::Controller::ThermostatSeq tss;
    for (i = 0; i < list->length(); ++i) {
        ts = CCS::Thermostat::_narrow(list[i]);
        if (CORBA::is_nil(ts))
            continue; // Skip thermometers
        len = tss.length();
        tss.length(len + 1);
        tss[len] = ts;
    }

    // Try to change all thermostats.
    try {
        ctrl->change(tss, 40);
    } catch (const CCS::Controller::EChange & ec) {
        cerr << ec; // Overloaded <<
    }
}
catch (const CORBA::Exception& ex) {
    cerr << "Uncaught CORBA exception: " << ex << endl;
    status = 1;
} catch (...) {
    status = 1;
}

if (!CORBA::is_nil(orb)) {
    try {
        orb -> destroy();
    } catch (const CORBA::Exception& ex) {
        cerr << ex << endl;
        status = 1;
    }
}

return status;
}
```

9. Server-Side C++ Mapping

Summary

This unit presents the basics of the server-side mapping. In particular, it covers how to connect a servant to the IDL-generated skeleton classes, how to implement attributes and operations, how to throw exceptions, and the memory management rules that apply to the server side. Note that this unit presents only the basics of server implementation. Unit 12 and Unit 15 cover implementation techniques in more depth.

Objectives

By the completion of this unit, you will be able to implement a simple CORBA server.

Introduction

The server-side C++ mapping is a superset of the client-side mapping.

Writing a server requires additional constructs to:

- connect servants to skeleton classes
- receive and return parameters correctly
- create object references for objects
- initialize the server-side run time
- run an event loop to accept incoming requests

The server-side mapping is easy to learn because most of it follows from the client-side mapping.



1
Server-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



9.1 Introduction

The server-side C++ mapping is a superset of the client-side mapping. IDL data types map identically and the parameter passing rules are simply the mirror image of the client-side rules. The remainder of the APIs is quite small and easy to learn.

Mapping for Interfaces

On the server side, a skeleton class provides the counterpart to the client-side proxy class.

Skeletons provides an up-call interface from the ORB networking layer into the application code.

The skeleton class contains pure virtual functions for IDL operations.

Skeleton classes have the name of the IDL interface with a `POA_` prefix. For example:

- `::MyObject` has the skeleton `::POA_MyObject`
- `CCS::Thermometer` has the skeleton class `POA_CCS::Thermometer`.

Note that modules map to namespaces as for the client side, and that the `POA_` prefix applies only to the outermost scope (whether module or interface).



2
Server-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



9.2 Mapping for Interfaces

For the server side, the IDL compiler generates separate skeleton header and source files. The skeleton classes defined in those files provide an up-call interface for the ORB into the server application code. The skeleton class contains pure virtual functions for each IDL operation or attribute.

The name of the skeleton class is the IDL interface name with a `POA_` prefix. For example, an interface `I` at global scope has the skeleton class `POA_I`. If an interface is nested in a module, the module maps to a namespace as usual, but with the `POA_` prefix on the module; everything inside the module then has the usual name without a prefix. For example, for a module `::M` containing an interface `I`, the name of the skeleton class is `I`, nested within a `::POA_M` namespace.

Skeleton Classes

The skeleton class for an interface contains a pure virtual function for each IDL operation:

```
interface AgeExample {
    unsigned short get_age();
};
```

The skeleton class contains:

```
class POA_AgeExample :
    public virtual PortableServer::ServantBase {
public:
    virtual CORBA::UShort
        get_age() throw(CORBA::SystemException) = 0;
    // ...
};
```



9.3 Skeleton Classes

The skeleton classes generated by the IDL compiler provide the link between the server-side ORB run time and your application code. The following points are worth noting:

- The name of the skeleton class is `POA_AgeExample`, whereas the name of the proxy for the same interface is `AgeExample`. In other words, even if client and server are collocated in the same address space, an object reference held by the client does not point directly at a skeleton class instance. Instead, it still points at the proxy as usual, and the proxy delegates calls to the skeleton. Retaining this extra level of collocation is important to preserve a number of transparencies provided by the POA. In particular, it ensures that client and server code see the same semantics during call dispatch regardless of whether they are collocated.
- The skeleton class inherits from `PortableServer::ServantBase`. In general, all skeletons have `ServantBase` as their ultimate ancestor. This design was chosen because it permits functionality that is common to all skeletons to be factored into a common base class. In addition, it permits you to pass servants generically (regardless of their interface type) by passing a pointer or reference to `ServantBase`.
- Each skeleton class contains a pure virtual function for each IDL operation. To dispatch an incoming request, the server-side run time invokes the corresponding virtual function on an instance of the skeleton class. Because the functions on the skeleton class are pure virtual, it follows that you cannot instantiate a skeleton class directly; instead you must instantiate a class that is derived from the skeleton and provides implementations for those pure virtual functions.

- Each member function of the skeleton has an exception specification. `CORBA::SystemException` always is present in this exception specification because all operations can throw a system exception. If an IDL operation raises user exceptions, the exception specification contains an additional entry for each user exception.

Servant Classes

Servant classes are derived from their skeleton:

```
#include "Age_skel.h"    // IDL file is called "Age.idl".
                        // Header file names are ORB-specific!

class AgeExample_impl : public virtual POA_AgeExample {
public:
    // Inherited IDL operation
    virtual CORBA::UShort
        get_age() throw(CORBA::SystemException);
    // You can add other members here...
private:
    AgeExample_impl(const AgeExample_impl &); // Forbidden
    void operator=(const AgeExample_impl &); // Forbidden
    // You can add other members here...
};
```



9.4 Servant Classes

The above example shows how to derive a servant class from its skeleton. The following points are worth noting:

- The name of the servant class is `AgeExample_impl`. You can use any name you like but, by convention, servant classes usually use their interface name with an `_impl` suffix.
- The servant class inherits from its skeleton class (`POA_AgeExample`) using virtual inheritance. Virtual inheritance is, strictly speaking, necessary only if multiple inheritance is used somewhere in the class hierarchy. However, we recommend that you always use virtual inheritance as a matter of principle because using virtual inheritance never does any harm (but forgetting to use it when it would be necessary has dire consequences).
- The name of the header file containing the skeleton class definition is not standardized and varies among different ORBs. ORBacus uses the base name of the IDL file with a `_skel` suffix as the header file name for the server side.
- The servant class must implement all pure virtual functions it inherits from the skeleton (otherwise, the servant class cannot be instantiated).
- The servant class prevents copying and assignment by making the copy constructor and assignment operation private and leaving them unimplemented. Although it is possible to permit copying and assignment of servants, it is usually not necessary or desirable. We recommend that you habitually hide the copy constructor and assignment operator because it will expose accidental copying and assignment in your code.
- Apart from the need to provide implementations for the inherited pure virtual functions, you can add whatever member functions and data members you want. Typically, servant classes

contain private data members that either store the state of the object represented by the servant, or store a handle that permits access to the object's state. It is also common to add a constructor as well as private helper functions that aid in manipulating a servant's state. In short, you can add whatever you deem suitable to support your servant implementation.

Operation Implementation

The implementation of a servant's virtual functions provides the behavior of an operation:

```
CORBA::UShort
AgeExample_impl::
get_age() throw(CORBA::SystemException)
{
    return 16;
}
```

Typically, the implementation of an operation will access private member variables that store the state of an object, or perform a database access to retrieve or update the state.

Once a servant's function is invoked, your code is in control and can therefore do whatever is appropriate for your implementation.



9.5 Operation Implementation

To implement an operation, you simply read `in` and `inout` parameters, set `inout` and `out` parameters, or return a value as appropriate; to indicate an error condition, you throw an exception. The above example is trivial, of course; typically, you will perform a database or network access as part of the operation implementation, or access private member variables that hold the state of the object. What exactly your operations do is entirely up to you; as far as the ORB is concerned, an operation is simply an up-call into the servant function, which provides whatever behavior is appropriate.

Attribute Implementation

As for the client side, attributes map to an accessor and modifier function (or just an accessor for **readonly** attributes):

```
interface Part {  
    readonly attribute long asset_num;  
    attribute long price;  
};
```

The skeleton code contains:

```
class POA_Part : public virtual PortableServer::ServantBase {  
public:  
    virtual CORBA::Long asset_num() throw(CORBA::SystemException) = 0;  
    virtual CORBA::Long price() throw(CORBA::SystemException) = 0;  
    virtual void price(CORBA::Long) throw(CORBA::SystemException) = 0;  
    // ...  
};
```



9.6 Attribute Implementation

Each IDL attribute maps to a pair of virtual functions with the same name, overloaded as a modifier and accessor. (If an attribute is **readonly**, the skeleton only contains an accessor but no modifier.) Otherwise, the implementation of an attribute is exactly as the same as for an operation. (This example illustrates that there truly is no difference in efficiency or implementation between attributes and operations. Attributes are simply IDL syntactic sugar.)

Servant Activation and Reference Creation

Every skeleton class contains a function called `_this`:

```
class POA_AgeExample :
    public virtual PortableServer::ServantBase {
public:
    AgeExample_ptr _this();
    // ...
};
```

To create a CORBA object, you instantiate the servant and call `_this`:

```
AgeExample_impl age_servant;           // Create servant
AgeExample_var av = age_servant._this(); // Create reference
```

- Instantiating the servant has no effect on the ORB.
- Calling `_this` activates the servant and returns its reference.

`_this` implicitly calls `_duplicate`, so you must eventually release the returned reference.



9.7 Servant Activation and Reference Creation

Merely instantiating a servant does nothing as far as the ORB is concerned. Immediately after instantiation of the servant, you simply have a C++ object instance. Before the ORB can dispatch requests to the servant, you must *activate* the servant. Activating a servant creates the link between an object reference and the C++ instance that is used by the ORB to dispatch requests to. In the above example, the call to `_this` both activates the servant and returns its object reference.¹

Note that `_this` implicitly calls `_duplicate` (which increases the reference count of the proxy from 0 to 1), so you must eventually release the returned reference to avoid a memory leak.

Once activated in this way, further calls to `_this` return the same reference as the first call.

Once you have obtained an object reference by calling `_this`, you can make that reference available to clients so they can invoke operations on the corresponding object. You can make the reference available as a string (by calling `object_to_string`), or you can pass it to a client as the return value of another operation invocation, or you can advertise the reference in a service such as the Naming or Trading Service.

NOTE: The above example instantiates the servant on the stack as a local variable. This is legal, but rare. Instead, you will almost always instantiate servants on the heap by calling `new`. We discuss the reasons for this in Section 12.25.

1. There are many more ways to activate a servant and to obtain its reference. The method shown above is typically used only for transient objects that are implicitly activated. We cover the various servant activation and reference creation options in Unit 12.

Server Initialization

A server must initialize the ORB run time before it can accept requests. To initialize the server, follow the following steps:

1. Call `ORB_init` to initialize the run time.
2. Get a reference to the Root POA.
3. Instantiate one or more servants.
4. Activate each servant.
5. Make references to your objects available to clients.
6. Activate the Root POA's POA manager.
7. Start a dispatch loop.



8
Server-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



9.8 Server Initialization

The above sequence of steps describes basic server initialization. All servers follow these steps (or follow some variation of these steps). Following is the `main` program for a simple server that makes an `AgeExample` object available to clients and then waits for incoming requests. Note that quite a few API calls shown here have not been explained. We will cover these in detail in later sections.

```
#include <iostream.h>
#include <OB/CORBA.h>
#include "Age_skel.h"

// Servant class definition here...

int
main(int argc, char * argv[])
{
    // Initialize ORB
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Get reference to Root POA
    CORBA::Object_var obj =
        orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa =
        PortableServer::POA::_narrow(obj);
```

```
// Create an object
AgeExample_impl age_servant;

// Write its stringified reference to stdout
AgeExample_var aev = age_servant._this();
CORBA::String_var str = orb->object_to_string(aev);
cout << str << endl;

// Activate POA manager
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();

// Accept requests
orb->run();
}
```

The code goes through the following steps:

1. Call `ORB_init` to initialize the run time.

Initialization for the server side is the same as for the client side. The call to `ORB_init` initializes the run time and returns a reference to the ORB. As for the client, `ORB_init` scans the command line for any arguments starting with `-ORB` and removes them. For ORBacus, options beginning with `-OA` are used to control the object adapter and are also removed.

2. Get a reference to the Root POA.

The next step is to obtain a reference to the POA. The ORB has a built-in distinguished POA known as the Root POA. The Root POA is pre-configured with certain policies and always is the first POA that a server must obtain. (You can create other POAs by making calls on the Root POA.) Calling `resolve_initial_references("RootPOA")` returns a reference to the Root POA.²

3. Instantiate one or more servants.

Here, we instantiate a single servant on the stack. (As mentioned earlier, you will mostly instantiate your servants on the heap. However, for this simple example, stack instantiation will be good enough.)

4. Activate each servant.

We call `_this` on the servant. This both activates the servant (makes its existence known to the POA) and returns an object reference for the corresponding object.

5. Make references to your objects available to clients.

For this simple example, we write a stringified reference for our object to `cout`. Calling `object_to_string` on the ORB reference produces a reference in stringified form. Clients can convert the string back into an active reference by calling `string_to_object`.

2. `resolve_initial_references` is used to gain access to a number of important references that clients and servers require initially. For example, you can use `resolve_initial_references` to get access to the Naming and Trading Service, the Interface Repository, and a number of other important references.

6. Activate the Root POA's POA manager.

Every POA has an associated POA manager which controls the flow of requests to that POA. You can obtain the POA manager for a POA by calling the `the_POAmanager` operation. Initially, POA managers start out in a holding state, in which incoming requests are queued. To pass requests through to a POA, the POA manager must be activated. You do this by calling the `activate` operation on the POA manager. For a single-threaded ORB, requests will not be dispatched immediately. Instead, request dispatch starts when you call `ORB::run`. However, for multi-threaded servers, request dispatch starts as soon as you call `activate`.

7. Start a dispatch loop.

The final step is to hand the thread of control to the ORB so it can start accepting incoming client requests. The call to `ORB::run` hands the thread of control to the ORB and does not return until it is interrupted.³ Internally, `ORB::run` creates an infinite loop around a call that monitors network connections (such as `select`); incoming client requests to servants are dispatched in this loop.

3. We will see other ways of making `ORB::run` return control to the caller in Section 9.13.

Parameter Passing

The parameter passing rules for the server side follow those for the client side.

If the client is expected to deallocate a parameter it receives from the server, the server must allocate that parameter:

- Variable-length **out** parameters and return values are allocated by the server.
- String and object reference **inout** parameters are allocated by the client; the server code must reallocate object references to change them and may reallocate **inout** strings or modify their contents in place.
- Everything else is passed by value or by reference.



9.9 Parameter Passing

The parameter passing rules for the server side are the mirror image of the client side. In short, this means that, if the client expects to be handed a dynamically-allocated value by the stub and is responsible for deallocating that value, the skeleton must allocate the corresponding value.

In the next few sections, we examine the parameter passing rules in detail for each possible parameter type and direction.

Parameter Passing (cont.)

Consider an operation that passes a **char** parameter in all possible directions:

```
interface Foo {
    char op(in char p_in, inout char p_inout, out char p_out);
};
```

The skeleton signature is:

```
virtual CORBA::Char op(
    CORBA::Char    p_in,
    CORBA::Char &  p_inout,
    CORBA::Char_out p_out
) throw(CORBA::SystemException) = 0;
```

Parameters are passed by value or by reference, as for the client side.

(Char_out is a typedef for Char &.)



9.9.1 Rules for Simple Types

Simple types are passed by value or by reference. A simple implementation of the above operation might be:

```
CORBA::Char
Foo_impl::
op(
    CORBA::Char    p_in,
    CORBA::Char &  p_inout,
    CORBA::Char_out p_out
) throw(CORBA::SystemException)
{
    // Use p_in, it's initialized already
    cout << p_in << endl;

    // Change p_inout
    p_inout = 'A';

    // Set p_out
    p_out = 'Z';

    // Return a value
    return 'B';
}
```

Parameter Passing (cont.)

Fixed-length unions and structures are passed by value or by reference:

```
struct F {
    char    c;
    short   s;
};

interface Foo {
    F op(in F p_in, inout F p_inout, out F p_out);
};
```

The skeleton signature is:

```
typedef F & F_out;
virtual F op(
    const F &    p_in,
    F &          p_inout,
    F_out        p_out
) throw(CORBA::SystemException) = 0;
```



9.9.2 Rules for Fixed-Length Complex Types

The rules for fixed-length complex types are the same as for simple types, that is, parameter passing is by value and by reference. A simple implementation of the above operation might be:

```
F Foo_impl::
op(const F &    p_in,
   F &          p_inout,
   F_out        p_out
) throw(CORBA::SystemException)
{
    // Use incoming values of p_in and p_inout (not shown)

    // Modify p_inout
    p_inout.c = 'A';
    p_inout.s = 1;

    // Initialize p_out
    p_out.c = 'Z';
    p_out.s = 99;

    // Create and initialize return value
    F result = { 'Q', 55 };
    return result;
}
```

Parameter Passing (cont.)

Fixed-length arrays are passed by pointer to an array slice:

```
typedef short SA[2];

interface Foo {
    SA op(in SA p_in, inout SA p_inout, out SA p_out);
};
```

The skeleton signature is:

```
typedef SA_slice * SA_out;
virtual SA_slice * op(
    const SA      p_in,
    SA_slice *    p_inout,
    SA_out        p_out
) throw(CORBA::SystemException) = 0;
```



9.9.3 Rules for Fixed-Length Array Types

Passing fixed-length arrays is similar to passing other fixed-length complex types. However, because it is impossible in C++ to pass arrays by value, the return value is dynamically allocated by the server. A simple implementation of the above operation might be:

```
SA_slice * Foo_impl::
op(const SA      p_in,
   SA_slice *    p_inout,
   SA_out        p_out
) throw(CORBA::SystemException)
{
    const size_t arr_len = sizeof(p_in) / sizeof(*p_in);

    // Use incoming values of p_in and p_inout (not shown)

    // Modify p_inout
    for (CORBA::ULong i = 0; i < arr_len; ++i)
        p_inout[i] = i;

    // Initialize p_out
    for (CORBA::ULong i = 0; i < arr_len; ++i)
        p_out[i] = i * i;

    // Create and initialize return value.
```



```
    SA_slice * result = SA_alloc();          // Dynamic allocation!  
    for (CORBA::ULong i = 0; i < arr_len; ++i)  
        result[i] = i * i * i;  
  
    return result;  
}
```

Note that the return value is dynamically allocated. You *must* use the generated allocation function (SA_alloc in this example).

Parameter Passing (cont.)

Strings are passed as pointers.

```
interface Foo {
    string op(
        in string      p_in,
        inout string   p_inout,
        out            string p_out
    );
};
```

The skeleton signature is:

```
virtual char * op(
    const char *      p_in,
    char * &         p_inout,
    CORBA::String_out p_out
) throw(CORBA::SystemException) = 0;
```



9.9.4 Rules for Strings

Strings that travel from server to client must be dynamically allocated. A simple implementation of the above operation might be:

```
char * Foo_impl::
op( const char *      p_in,
    char * &         p_inout,
    CORBA::String_out p_out
) throw(CORBA::SystemException)
{
    // Use incoming value
    cout << p_in << endl;

    // Change p_inout
    size_t len = strlen(p_inout);
    for (i = 0; i < len; ++i)
        to_lower(p_inout[i]);

    // Create and initialize p_out
    p_out = CORBA::string_dup("Hello");

    // Create and initialize return value
    return CORBA::string_dup("World");
}
```

Note that it is legal to change an `inout` string in place, as shown in the preceding example. If you want to change an `inout` string such that it becomes shorter than the initial string, you can either write NUL byte into the initial string, or you can reallocate the string:

```
char * Foo_impl::
op( const char *      p_in,
    char * &        p_inout,
    CORBA::String_out p_out
) throw(CORBA::SystemException)
{
    // ...

    // Change p_inout
    *p_inout = '\\0'; // Shorten by writing NUL

    // OR:

    CORBA::string_free(p_inout);
    p_inout = CORBA::string_dup(""); // Shorten by reallocation

    // ...
}
```

If you want to lengthen an `inout` string, you have no choice but to reallocate it:

```
char * Foo_impl::
op( const char *      p_in,
    char * &        p_inout,
    CORBA::String_out p_out
) throw(CORBA::SystemException)
{
    // ...

    // Lengthen inout string by reallocation
    CORBA::string_free(p_inout);
    p_inout = CORBA::string_dup(longer_string);

    // ...
}
```

Parameter Passing (cont.)

Sequences and variable-length structures and unions are dynamically allocated if they are an **out** parameter or the return value.

```
typedef sequence<octet> OctSeq;
interface Foo {
    OctSeq op(
        in OctSeq    p_in,
        inout OctSeq p_inout,
        out OctSeq   p_out
    );
};
```

The skeleton signature is:

```
typedef OctSeq & OctSeq_out;
virtual OctSeq * op(const OctSeq & p_in,
                   OctSeq &      p_inout,
                   OctSeq_out    p_out
                   ) throw(CORBA::SystemException) = 0;
```



9.9.5 Rules for Variable-Length Complex Types

Variable-length complex types are dynamically allocated in the **out** direction and for the return value. A simple implementation of the above operation might be:

```
OctSeq *
Foo_impl::
op( const OctSeq & p_in,
    OctSeq &      p_inout,
    OctSeq_out    p_out
) throw(CORBA::SystemException)
{
    // Use incoming values of p_in and p_inout (not shown)

    // Modify p_inout
    CORBA::ULong len = p_inout.length();
    p_inout.length(++len);
    for (CORBA::ULong i = 0; i < len; ++len)
        p_inout[i] = i % 256;

    // Create and initialize p_out
    p_out = new OctSeq;
    p_out->length(1);
    (*p_out)[0] = 0;
```

```

    // Create and initialize return value
    OctSeq * p = new OctSeq;
    p->length(2);
    (*p)[0] = 0;
    (*p)[1] = 1;

    some_func();                // Potential leak here!

    return p;
}

```

Note that it is necessary to first dereference `p` before applying the subscript operator. If you write

```
p[0] = 0; // Wrong!
```

the compiler will assume that `p` points at an array of sequences and that zero is being assigned to the first sequence in the array. To avoid this mistake, you can use a `_var` type as the return value instead:

```

OctSeq *
Foo_impl::
op( const OctSeq & p_in,
    OctSeq & p_inout,
    OctSeq_out p_out
) throw(CORBA::SystemException)
{
    // ...

    // Create and initialize return value
    OctSeq_var p = new OctSeq;
    p->length(2);
    p[0] = 0;
    p[1] = 1;

    some_func();                // No leak here

    return p._retn();
}

```

This version has the added advantage that it is exception safe. If an exception is thrown anywhere between the allocation and the return, the `_var` type will take care of correctly deallocating the allocated memory, whereas the previous version (which didn't use a `_var` type) will leak that memory.

Parameter Passing (cont.)

```

struct Employee {
    string name;
    Long    number;
};
typedef Employee EA[2];

interface Foo {
    EA op(in EA p_in, inout EA p_inout, out EA p_out);
};

```

The skeleton signature is:

```

virtual EA_slice * op(
    const EA    p_in,
    EA_slice *  p_inout,
    EA_out     p_out
) throw(CORBA::SystemException) = 0;

```



9.9.6 Rules for Variable-Length Arrays

Variable-length array types are passed like fixed-length arrays, with the exception that out parameters for variable-length arrays must be dynamically allocated. A simple implementation of the above operation might be:

```

EA_slice *
Foo_impl::
op( const EA    p_in,
    EA_slice *  p_inout,
    EA_out     p_out
) throw(CORBA::SystemException)
{
    size_t arr_len = sizeof(p_in) / sizeof(*p_in);

    // Use p_in and initial value of p_inout (not shown)

    // Modify p_inout
    p_inout[0] = p_inout[1];
    p_inout[1].name = CORBA::string_dup("Michi");
    p_inout[1].number = 1;

    // Create and initialize p_out
    p_out = EA_alloc();
    for (CORBA::ULong i = 0; i < arr_len; ++i) {

```

```
        p_out[i].name = CORBA::string_dup("Sam");
        p_out[i].number = i;
    }

    // Create and initialize return value
    EA_slice * result = EA_alloc();
    for (CORBA::ULong i = 0; i < arr_len; ++i) {
        result[i].name = CORBA::string_dup("Max");
        result[i].number = i;
    }

    return result;
}
```

Parameter Passing (cont.)

Object reference **out** parameters and return values are duplicated.

```
interface Thermometer { /* ... */ };

interface Foo {
    Thermometer op(
        in Thermometer    p_in,
        inout Thermometer p_inout,
        out Thermometer    p_out
    );
};
```

The skeleton signature is:

```
virtual Thermometer_ptr op(
    Thermometer_ptr    p_in,
    Thermometer_ptr & p_inout,
    Thermometer_out    p_out
) throw(CORBA::SystemException) = 0;
```



9.9.7 Rules for Object References

For object references, you must reallocate inout parameters and allocate out parameters and the return value. A simple implementation of the above operation might be:

```
Thermometer_ptr
Foo_impl::
op( Thermometer_ptr    p_in,
    Thermometer_ptr & p_inout,
    Thermometer_out    p_out
) throw(CORBA::SystemException)
{
    // Use p_in
    if (!CORBA::is_nil(p_in))
        cout << p_in->temperature() << endl;

    // Use p_inout
    if (!CORBA::is_nil(p_inout))
        cout << p_inout->temperature() << endl;

    // Modify p_inout
    CORBA::release(p_inout);
    p_inout = Thermometer::_duplicate(p_in);

    // Initialize p_out
```



```
    p_out = Thermostat::_narrow(p_in);  
  
    // Create return value  
    return _this();  
}
```

Note that, before using `p_in` or `p_inout`, the code ensures that these references are not nil. Also note that the new value of `p_inout` is allocated with `_duplicate`. `p_out` and the return value are allocated as well (`_narrow` and `_this` implicitly call `_duplicate`).

Throwing Exceptions

The exception mapping is identical for client and server. To throw an exception, instantiate the appropriate exception class and throw it.

You can either instantiate an exception as part of the throw statement, or you can instantiate the exception first, assign to the exception members, and then throw the exception.

You should always make your implementation exception safe in that, if it throws an exception, no durable state changes remain.

Avoid throwing system exceptions and use user exceptions instead.

If you must use a system exception, set the **CompletionStatus** appropriately.



9.10 Throwing Exceptions

Throwing an exception is a simple matter of instantiating the exception and throwing it. For example:

```
CCS::TempType
Thermostat_impl::
set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    if (new_temp > max_temp || new_temp < min_temp) {
        CCS::Thermostat::BtData btd;
        btd.requested = new_temp;
        btd.min_temp = min_temp;
        btd.max_temp = max_temp;
        throw CCS::Thermostat::BadTemp(btd);
    }
    // Remember previous nominal temperature and
    // set new nominal temperature...
    return previous_temp;
}
```

In this code example, the exception is initialized with a temporary variable of type BtData.

Alternatively, you can instantiate the exception and initialize it without using a temporary variable:

```

CCS::TempType
Thermostat_impl::
set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    if (new_temp < min_temp || new_temp > max_temp) {
        CCS::Thermostat::BadTemp bt;
        bt.details.requested = new_temp;
        bt.details.min_temp = min_temp;
        bt.details.max_temp = max_temp;
        throw bt;
    }
    // Remember previous nominal temperature and
    // set new nominal temperature...
    return previous_temp;
}

```

For exceptions with a few simple members, it is usually easiest to use the exception constructor in the throw statement. For example:

```

exception ErrorReport {
    string      file_name;
    unsigned long line_num;
    string      reason;
};

```

You can use a single throw statement to construct and throw this exception:

```

throw ErrorReport("foo.cc", 597, "Syntax error");

```

In general, you should avoid throwing system exceptions. Not only is it difficult to convey enough detail to the caller with a system exception, it also makes it harder to debug the code because it is no longer clear whether a particular system exception was raised by the ORB run time or by the application code. If you must throw a system exception, take care to set the `CompletionStatus` correctly. Ideally, you should undo any side-effects of the operation before throwing the exception, so the state of the object is the same as it was before the invocation. In that case `COMPLETED_NO` is the correct status to use. Otherwise, you must set the status to `COMPLETED_YES`, even if not all side-effects happened. (`COMPLETED_MAYBE` is never appropriate when application code throws a system exception.)

Remember that the default constructor for system exceptions sets the completion status to `COMPLETED_NO` and the minor code to zero:

```

if (input_parameter_unacceptable)
    throw CORBA::BAD_PARAM();

```

To set the `CompletionStatus` to `COMPLETED_YES`, you can use a throw like the following:

```

if (db_connection_broke_after_partial_update)
    throw CORBA::PERSIST_STORE(CORBA::COMPLETED_YES, 0);

```

Exception Pitfalls

- If you throw an exception and have allocated memory to a variable-length **out** parameter, you must deallocate that memory first.

Use `_var` types to prevent such memory leaks.

- Do not throw user exceptions that are not part an operation's **raises** clause.

Use a `try` block around calls to other operations that may throw user exceptions.



18
Server-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



9.11 Exception Pitfalls

There are two common pitfalls with respect to exceptions: memory leaks and illegal user exceptions.

9.11.1 Avoiding Memory Leaks

If you have allocated memory for a variable-length `out` parameter and then throw an exception, that memory will be leaked unless you deallocated it first. For example:

```
interface Example {
    void get_name(out string name);
};
```

The following code will leak memory:

```
void
Example_impl::
op(CORBA::String_out name)
{
    name = CORBA::string_dup("Otto");

    // Do some database access or whatever...
    if (database_access_failed)
        throw CORBA::PERSIST_STORE(); // Bad news!
}
```

The memory that was allocated to the name parameter is leaked in this case. To prevent the leak, you could explicitly deallocate the memory again before throwing the exception. However, doing so is needlessly complex and error prone. A better way to deal with memory leaks is to use a `_var` temporary:

```
void
Example_impl::
op(CORBA::String_out name)
{
    CORBA::String_var name_tmp = CORBA::string_dup("Otto");

    // Do some database access or whatever...
    if (database_access_failed)
        throw CORBA::PERSIST_STORE(); // OK, no leak
    name = name_tmp._retn();           // Transfer ownership
}
```

This code is free of leaks because, if an exception is thrown, the destructor of `name_tmp` will deallocate its memory. Once it is clear that no more exception can occur, `name_tmp` is used to pass ownership of the allocated string to the name parameter.

As we saw on page 6-38, the same technique can be used for return values, or in fact for any dynamically allocated resource. (The use of `_var` types in this way is one example of Stroustrup's "Resource Acquisition is Initialization" idiom.)

9.11.2 Avoiding Illegal User Exceptions

An operation implementation in a servant can only throw those user exceptions that appear in the corresponding IDL `raises` clause. This is because the virtual function generated into the skeleton has a `throw` specification that explicitly lists all legal user exceptions. If you throw a user exception that is not in the `throw` specification, you will end up in the C++ `unexpected` function (which, by default, terminates the process). You can easily make this mistake if you invoke other operations as part of an operation's implementation. For example:

```
interface EmployeeFinder {
    struct Details { /* ... */ };
    exception BadEmployee { /* ... */ };
    Details get_details(in string name) raises(BadEmployee);
    // ...
};

interface ReportGenerator {
    exception BadDepartment { /* ... */ };
    void show_employees(in string department) raises(BadDepartment);
    // ...
};
```

The following implementation of `show_employees` illustrates the problem:

```
void
ReportGenerator_impl::
show_employees(const char * department)
throw(CORBA::SystemException, ReportGenerator::BadDepartment)
```

```

{
    EmployeeFinder_var ef = ...;

    // Locate department and get list of employee names...
    for (each emp_name in list) {
        Details_var d = ef->get_details(emp_name); // Dubious!
        // Show employee's details...
    }
}

```

The problem with this code is that if the employee list is not accurate for some reason, it will call `get_details` with an invalid name. In turn, `get_details` will throw a `BadEmployee` exception, which, according to the exception specification of `show_employees`, is illegal and results in process termination.

This kind of problem can crop up whenever you invoke other functions that may throw exceptions. Note that this mistake can also bite you if you call a non-CORBA function. If the called function throws a C++ exception (such as `std::ios_base::failure`), your program suffers the same fate.

The way around the problem is to catch exceptions in at least a generic manner if one of the functions you call may throw exceptions. For example:

```

void
ReportGenerator_impl::
show_employees(const char * department)
throw(CORBA::SystemException, ReportGenerator::BadDepartment)
{
    try {
        EmployeeFinder_var ef = ...;

        // Locate department and get list of employee names...
        for (each emp_name in list) {
            try {
                Details_var d = ef->get_details(emp_name);
                // Show employee's details...
            } catch (const EmployeeFinder::BadEmployee &) {
                // Ignore bad employee and try next one...
            }
        }
    } catch (const CORBA::Exception &) {
        // Other CORBA exceptions are dealt with higher up.
        throw;
    } catch (...) { // This really is an assertion failure
        // because it indicates a bug in the code
        write_error_log_report();
        throw CORBA::INTERNAL();
    }
}

```

Here, the code protects itself against user exceptions from `get_details` by skipping unknown employee names and keeps going, and deals with non-CORBA exceptions by logging them.

Tie Classes

The C++ mapping offers an alternative way to implement servants.

A tie class replaces inheritance from the skeleton with delegation:

```
class Thermometer_impl {    // NO inheritance here!
public:
    // Usual CORBA operation implementation here...
};

// ...
```

```
Thermometer_impl * servantp = new Thermometer_impl;
POA_CCS::Thermometer_tie<Thermometer_impl> tie(servantp);
CCS::Thermometer_var = tie._this();
```

The tie instance delegates each call to the implementation instance, so the implementation instance does not have to inherit from a skeleton.

The IDL compiler generates ties with the `--tie` option.



9.12 Tie Classes

The C++ mapping offers an alternative way to implement servants via tie classes. The basic idea is to replace inheritance from the skeleton class with delegation from a generated tie class. That way, the implementation class does not need to inherit from any base class. Occasionally, this can be useful, for example, if you must implement your implementation classes by using a class framework that forces you to extensively use inheritance; the tie approach can help to avoid complex and difficult to understand inheritance hierarchies in such a case.

The generated tie class has the name of the skeleton class with a `_tie` suffix. The tie class is a template; the actual implementation class becomes the template parameter when you instantiate a tie. The constructor of a tie requires the address of the implementation class; that pointer is stored in a private data member of the tie so it can delegate invocations to the implementation class.

The tie approach has a number of disadvantages. For one, it is possible to navigate from a tie to its implementation instance, but it is not possible to navigate the relationship in the opposite direction. However, as we will see in Section 12.25, navigation in the opposite direction is required if you want to support life cycle operations on your objects, so you must store a back-pointer to the tie in the implementation class. Second, ties can create a number of problems with respect to thread safety, requiring extra locking around creation and destruction of servants.

Overall, other features of the POA have made the tie approach largely redundant; we recommend that you avoid ties unless you have an overriding reason to use them. We will not cover ties further in this course. (See Henning & Vinoski if you require further details.)

Clean Server Shutdown

The **ORB** object contains **shutdown** and **destroy** operations that permit clean server shutdown:

```
interface ORB {
    void shutdown(in boolean wait_for_completion);
    void destroy();
    // ...
};
```

- With a false parameter, **shutdown** initiates ORB shutdown and returns immediately.
- With a true parameter, **shutdown** initiates ORB shutdown and does not return until shutdown is complete.

ORB shutdown stops accepting new requests, allows requests in progress to complete, and destroys all object adapters.

You *must* call **destroy** before leaving **main**!



9.13 Clean Server Shutdown

The example server main program we saw on page 9-12 had no way to terminate cleanly because `ORB::run` does not return; the only way to terminate such a server is to kill it with a signal. For simple servers, this is acceptable. However, more complex servers must usually perform some finalization tasks before they can exit, so we require a way to cause `ORB::run` to return the main thread of control.

A call to `ORB::shutdown` initiates ORB shutdown. The single `wait_for_completion` parameter determines whether `shutdown` should wait for shutdown to complete: if `wait_for_completion` is true, `shutdown` waits until all current requests have completed and all object adapters have been destroyed before returning control to the caller; if `wait_for_completion` is false, `shutdown` initiates shutdown and returns without blocking the caller. In the latter case, you can follow a call to `shutdown(false)` (which initiates shutdown) with a later call to `shutdown(true)`. If, by the time the second call is made, shutdown has completed, the second call returns immediately; otherwise, it blocks and returns when shutdown is complete.

Once `run` has returned (because of a call to `shutdown`), you must call `ORB::destroy` before leaving `main`. Failure to do so may cause resource leaks, depending on the environment.⁴ The effect of calling `destroy` is to destroy the ORB object, which, in turn, reclaims all resources that were associated with the ORB. (See Section 12.26 for a discussion of the cleanup actions.) Once you have called `destroy`, you must not invoke any operations you invoke on the ORB reference raise `OBJECT_NOT_EXIST`. Once the ORB is shutdown, the only legal CORBA operations you

4. Under UNIX, the kernel will clean up and reclaim all resources. However, for other environments, such as Windows or embedded systems, that is not the case and failure to call `destroy` may cause permanent resource leaks.

can invoke are `_duplicate`, `CORBA::release`, `CORBA::is_nil`, `ORB::destroy`, and `ORB_init`. These rules allow you to use global `_var` references without running the risk of something going wrong in their destructors. In addition, these rules make it possible for you to destroy an ORB and to create a new one by calling `ORB_init`.

Armed with this knowledge, we can revisit the server code on page 9-12 and make it “shutdown safe”:

```
#include <iostream.h>
#include <OB/CORBA.h>
#include "Age.h"

// Servant class definition here...

CORBA::ORB_var orb;      // Global, OK!

int
main(int argc, char * argv[])
{
    int status = 0;      // Return value from main()

    try {
        // Initialize ORB
        orb = CORBA::ORB_init(argc, argv);

        // Get Root POA, etc., and initialize application...

        // Accept requests
        orb->run();      // orb->shutdown(false) may be called
                        // from elsewhere, such as another
                        // thread, a signal handler, or as
                        // part of an operation.
    } catch (...) {
        status = 1;
    }

    // Don't invoke CORBA operations from here on, it won't work!

    if (!CORBA::is_nil(orb)) { // If we created an ORB...
        try {
            orb->destroy();    // Wait for shutdown to complete
                               // and destroy ORB
        } catch (const CORBA::Exception &) {
            status = 1;
        }
    }

    // Do application-specific cleanup here...

    return status;
}
```

Note that we have added some exception handling to this example to ensure that `main` returns a failure status to the operating system if something goes wrong. The pertinent points of this code are:

- `ORB::run` is called from the main thread. This is necessary for maximum portability. (Depending on the underlying thread support, calling `run` from a thread other than the main thread may not work.)
- If `shutdown(false)` is called while the run loop is dispatching requests, `run` returns and the code calls `ORB::destroy` before it leaves `main`. Again, this is necessary for portability. (Depending on the underlying thread support, calling `destroy` after `main` returns, for example, from a global constructor, may not work.)
- `destroy` does not return until after all executing requests have completed and all resources have been correctly reclaimed. This not only ensures resource recovery, but also provides a point at which your application can perform its own clean-up work.

You should write all your server code to follow this pattern.

Handling Signals (UNIX)

To react to signals and terminate cleanly, call `shutdown` from within the signal handler:

```
extern "C"
void handler(int)
{
    try {
        if (!CORBA::is_nil(orb))
            orb->shutdown(false);
    } catch (...) {
        // Can't throw here...
    }
}
```

You can install the signal handler on entry to `main`.

You should handle at least `SIGINT`, `SIGHUP`, and `SIGTERM`.

Do not call `shutdown(true)` or `destroy` from within a signal handler!



21
Server-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



9.14 Handling Signals (UNIX)

The `shutdown` function is guaranteed safe for calling from within a signal handler.⁵ We can exploit this fact to achieve clean termination of a server on receipt of a signal. (You should make it a habit to always handle `SIGINT`, `SIGHUP`, and `SIGTERM` because these signals are typically used to achieve clean process shutdown and are sent when a system administrator wants to bring down the machine.)

You can establish the handler on entry to `main`, using the `sigaction` call (in preference to the old `signal` call, which suffers from race conditions and does not allow you to achieve restarting interrupted system calls):

```
struct sigaction sa; // New signal state

sa.sa_handler = handler; // Set handler function
sigfillset(&sa.sa_mask); // Mask all other signals
// while handler runs
sa.sa_flags = 0 | SA_RESTART; // Restart interrupted syscalls

if (sigaction(SIGINT, &sa, (struct sigaction *)0) == -1)
    abort();
if (sigaction(SIGHUP, &sa, (struct sigaction *)0) == -1)
    abort();
```

5. At least in ORBacus. Unfortunately, the standard does not say anything about signal handling.

```

if (sigaction(SIGTERM, &sa, (struct sigaction *)0) == -1)
    abort();

// Initialize ORB, etc...

```

The handler itself should set the caught signals to be ignored. This is useful to prevent signals that are sent to a process in rapid succession from triggering the handler a second time:

```

extern "C"
void handler(int)
{
    // Ignore further signals
    struct sigaction ignore;
    ignore.sa_handler = SIG_IGN;
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGTERM, &ignore, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGHUP, &ignore, (struct sigaction *)0) == -1)
        abort();

    // Terminate event loop
    try {
        if (!CORBA::is_nil(orb))
            orb->shutdown(false);
    } catch (...) {
        // Can't throw here...
    }
}

```

Note that you must not call `shutdown(true)` from a signal handler. This is because `shutdown(true)` will cause actions inside the ORB that are not signal-safe. For the same reason, you must not call `destroy` from within a signal handler.⁶

6. Note that if you want to cause shutdown from within an operation implementation, you must use `shutdown(false)`. Calling `shutdown(true)` or `destroy` from within an executing operation raises `BAD_INV_ORDER`.

Handling Signals (Windows)

For Windows, use the following signal handler:

```

BOOL
handler(DWORD)
{
    // Inform JTC of presence of new thread
    JTCAdoptCurrentThread adopt;

    // Terminate event loop
    try {
        if (!CORBA::is_nil(orb))
            orb->shutdown(false);
    } catch (...) {
        // Can't throw here...
    }
    return TRUE;
}

```



22
Server-Side C++ Mapping
Copyright 2000–2001 IONA Technologies



9.15 Handling Signals (Windows)

For Windows operating systems, you can use a signal handler as shown above. You must instantiate a `JTCAdoptCurrentThread` instance inside the handler because console events are handled in a separate thread.

To install the handler, use the following code in `main`:

```

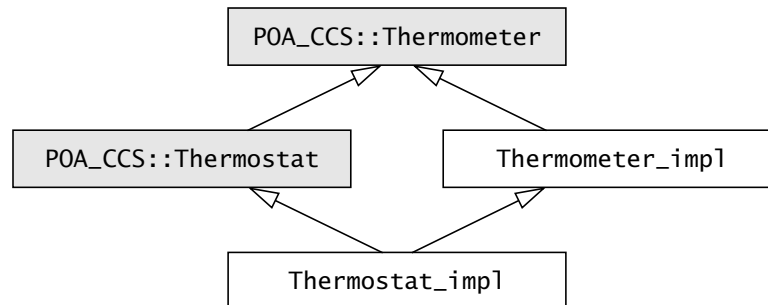
BOOL rc = SetConsoleCtrlHandler((PHANDLER_ROUTINE)handler, TRUE);
if (!rc) {
    // Could not install handler
    abort();
}

```

`SetConsoleCtrlHandler` installs a handler for the following events: `CTRL_C_EVENT`, `CTRL_BREAK_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, and `CTRL_SHUTDOWN_EVENT`.

Implementation Inheritance

If you are implementing base and derived interfaces in the same server, you can use implementation inheritance:



Thermometer_impl implements pure virtual functions inherited from **POA_CCS::Thermometer**, and **Thermostat_impl** implements pure virtual functions inherited from **POA_CCS::Thermostat**.



9.16 Implementation Inheritance

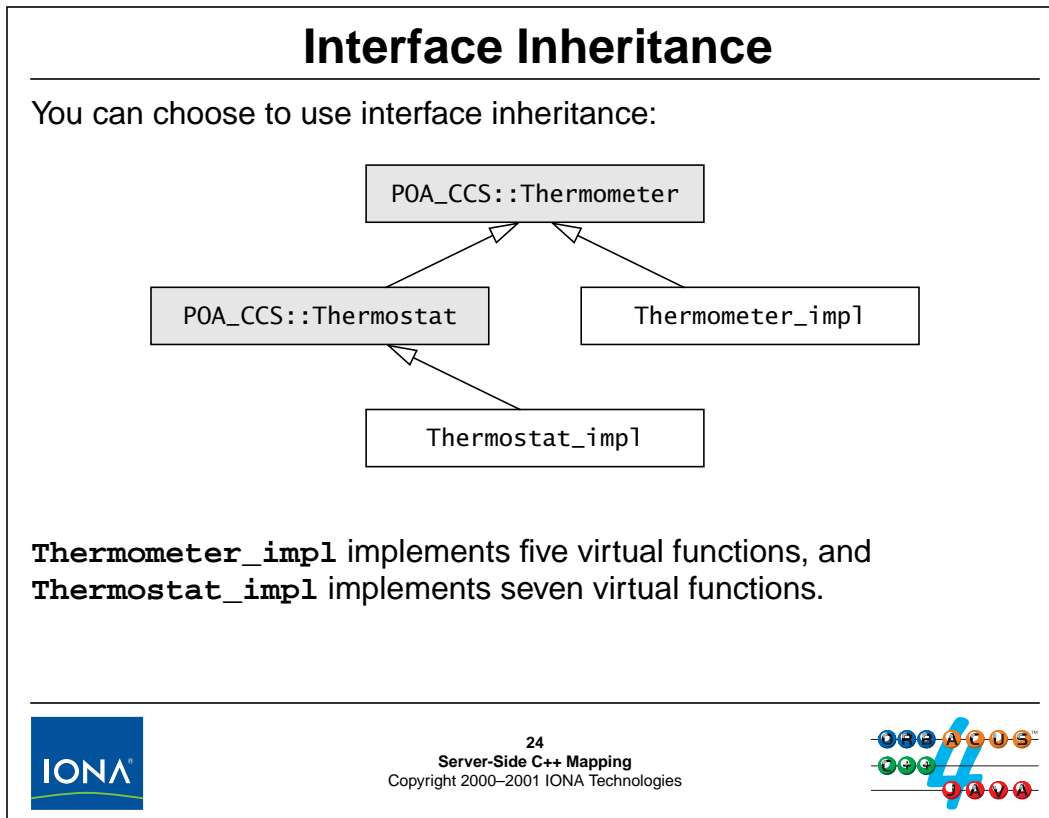
The above diagram shows how to implement derived interfaces using implementation inheritance. The C++ definitions of the implementation classes are as follows:

```

class Thermometer_impl : public virtual POA_CCS::Thermometer {
// ...
};

class Thermostat_impl : public virtual POA_CCS::Thermostat,
                        public virtual Thermometer_impl {
// ...
}
  
```

Obviously, this approach is possible only if the servants for both base and derived interfaces are implemented in the same process. Using implementation inheritance, the derived **Thermostat_impl** servant must implement only the `get_nominal` and `set_nominal` operations defined in interface **Thermostat** and need not implement the four attributes it inherits from interface **Thermometer**.



9.17 Interface Inheritance

You need not use implementation inheritance to implement the servant for a derived interface. Instead, by omitting the inheritance from the base servant class, you can choose to use a completely separate implementation for the servant of the derived interface:

```

class Thermometer_impl : public virtual POA_CCS::Thermometer {
// ...
};

class Thermostat_impl : public virtual POA_CCS::Thermostat {
// ...
}
  
```

Naturally, using this implementation technique, you must implement seven virtual functions, five to provide implementations for the four attributes in interface **Thermometer**, and two to provide implementations for the two operations in interface **Thermostat**.

Note that this technique can also be used to simulate implementation inheritance if the base and derived interfaces are provided by different servers. For example, if you have two servers, one that implements thermometers and one that implements thermostats, you can simulate implementation inheritance for the base part of thermometers with delegation. In that case, a **Thermostat** servant would store an object reference to its **Thermometer** base instance and delegate invocations on the base part of a **Thermostat** via that reference.

Compiling and Linking

To create a server executable, you must compile the application code, skeleton code, and the stub code. Typical compile commands are:

```
c++ -I. -I/opt/OB/include -c server.cc
c++ -I. -I/opt/OB/include -c MyIDL_skel.cpp
c++ -I. -I/opt/OB/include -c MyIDL.cpp
```

The exact flags and compile command vary with the platform.

To link the server, you must link against libOB:

```
c++ -o server server.o MyIDL_skel.o MyIDL.o -L/opt/OB/lib -lOB
```

If you are using JThreads/C++, you also need to link against the JTC library and the native threads library. For example:

```
c++ -o server server.o MyIDL_skel.o MyIDL.o -L/opt/OB/lib \
-lOB -lJTC -lpthread
```



9.18 Compiling and Linking

Compiling and linking a server is almost identical to compiling and linking a client (see page 6-69), except that a server must also contain the generated skeleton code. Note that the exact commands required vary with your operating system and compiler.

10. Exercise: Writing a Server

Summary

In this unit, you will develop a server that implements the CCS IDL we presented in Unit 4.

Objectives

By the completion of this unit, you will have a detailed understanding of how to implement servant classes, accept requests, process parameters, and throw exceptions on the server side.

10.1 Source Files and Build Environment

You will find this exercise in your `server` directory. The files in this directory are the same as for Unit 7. The `client.cpp` file contains the solution to the exercise presented in Unit 8. You will use this program as the test harness for the changes you make to the server source code.

10.2 Server Operation

The server implements a single controller object and a fixed number of thermometers and thermostats. The server uses stack-based servants for now. (We will change it to use proper reference-counted servants in Unit 13.)

The server uses a simulated Instrument Control Protocol (ICP) to access hypothetical devices on a network. The protocol simulator is implemented in the `icp` directory. The ICP API is very simple and can be found in `icp/icp.h`. (See Henning & Vinoski, pages 390–393 for details.) For this exercise, the simulated network is non-persistent, so every time the server shuts down, all updates you made to the network are lost. You will not need to change the implementation of the ICP simulator, but you should at least briefly review the description of the simulator in Henning & Vinoski so you have some idea how the server accesses and updates state.

The IDL operations in the server read and write to the ICP network using helper functions defined in each class. For example, `Thermometer_impl` contains a helper function called `get_model` that reads the model string for a device from the network, and `Thermostat_impl` contains a helper function `set_nominal_temp` that updates a thermostat on the network with a new temperature setting. You will implement all operations in the server using these helper functions.

The single controller object in the server is implemented by the `Controller_impl` class. This class contains a private member variable and two public helper functions:

```
class Controller_impl : public virtual POA_CCS::Controller {
public:
    // ...

    // Helper functions to allow thermometers and
    // thermostats to add themselves to the m_assets map
    // and to remove themselves again.
    void add_impl(CCS::AssetType anum, Thermometer_impl * tip);
    void remove_impl(CCS::AssetType anum);

private:
    // Map of known servants
    typedef map<CCS::AssetType, Thermometer_impl *> AssetMap;
    AssetMap m_assets;

    // ...
};
```

The `m_assets` member maps asset numbers to `Thermometer_impl` pointers. Because `Thermostat_impl` is derived from `Thermometer_impl`, this means that both thermometer and thermostat servant pointers can be stored in this map. The map is used by the controller servant to implement functionality such as `list` and `find`. The constructor of `Thermometer_impl` calls `m_ctrl->add_impl(anum, this)` to add itself to the

controller's map, and the destructor calls `m_ctrl->remove_impl` to remove itself from the map. This ensures that the controller servant has an accurate view of what devices exist. The `m_ctrl` variable is a static member variable of `Thermometer_impl` (initialized in the `run` function), so `thermometer` and `thermostat` servants can get at the controller servant.

10.3 What You Need to Do

Step 1

Read the description of the ICP API in Henning & Vinoski, pages 390–393.

Step 2

Study the contents of the `server.h` file. Make sure that you understand how the class hierarchy works and what member functions and member variables are used in each class. Note that `m_anum` is a protected member in `Thermometer_impl` so it is accessible in the derived `Thermostat_impl` class. Also have a look at how the `StrFinder` function object is implemented. This function object is used for the implementation of `find` and searches the `m_assets` map for devices that match a specific location or model string.

Step 3

Have a look at the `main` function in `server.cpp`. Make sure you understand how the signal handler works and how `main` is structured. The code for the server is mostly implemented in the `run` function. Look at what the `run` function does and make sure you understand how it works. Look through the actions of the constructors for `Thermometer_impl` and `Thermostat_impl` and follow the logic to make sure you understand how the controller servant keeps track of devices.

Step 4

The `get_loc` helper function and the member function to read the IDL `location` attribute are empty. Implement these functions. Be sure to add some trace to your code so you can see what the server is doing when it runs. Use the provided client to test your changes.

Step 5

The `set_nominal_temp` helper function is only partially implemented. The code to throw an exception if the nominal temperature is out of range is missing. Add the missing code. (Note: the ICP API returns the minimum and maximum temperature for attributes named `MIN_TEMP` and `MAX_TEMP`.)

Step 6

Implement the body of the `list` operation in the controller. You can loop over the contents of the `m_assets` map with loop such as:

```
AssetMap::iterator i;
for (i = m_assets.begin(); i != m_assets.end(); i++)
    // ...
```

The expression `i->first` evaluates to the asset number, and `i->second` evaluates to the servant pointer.

Step 7

Implement the body of the `change` operation in the controller. Hint: an easy way to implement this operation is to instantiate an exception at the beginning, just in case it is needed. Then loop over the input sequence and attempt to set the temperature on each thermostat. If the attempt fails, add an element to the sequence in the exception you have previously allocated. When the loop terminates, check if the sequence inside the exception has non-zero length. If so, throw the exception.)

Step 8 (Difficult)

Implement the body of the `find` operation in the controller. To search for an asset number that matches, you can use an expression such as:

```
AssetMap::iterator where;  
where = m_assets.find(28);  
if (where != m_assets.end())  
    // Found it, where points at map entry
```

To search for devices by location or asset number, you can use a search such as:

```
AssetMap::iterator where;    // Iterator for asset map  
where = find_if(  
    m_assets.begin(), m_assets.end(),  
    StrFinder(CCS::Controller::LOCATION, "some_string")  
);  
if (where != m_assets.end())  
    // Found it...
```

11. Solution: Writing a Server

11.1 Solution

Step 4

```
// Helper function to read the location from a device.

CCS::LocType
Thermometer_impl::
get_loc()
{
    char buf[32];
    if (ICP_get(m_anum, "location", buf, sizeof(buf)) != 0)
        abort();
    return CORBA::string_dup(buf);
}

// IDL location attribute accessor.

CCS::LocType
Thermometer_impl::
location() throw(CORBA::SystemException)
{
    return get_loc();
}
```

Step 5

```
// Helper function to set a thermostat's nominal temperature.

CCS::TempType
Thermostat_impl::
set_nominal_temp(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    short old_temp;

    // We need to return the previous nominal temperature,
    // so we first read the current nominal temperature before
    // changing it.
    if (ICP_get(
        m_anum, "nominal_temp", &old_temp, sizeof(old_temp)
    ) != 0) {
        abort();
    }

    // Now set the nominal temperature to the new value.
    if (ICP_set(m_anum, "nominal_temp", &new_temp) != 0) {

        // If ICP_set() failed, read this thermostat's
        // minimum and maximum so we can initialize the
```



```

        // BadTemp exception.
        CCS::Thermostat::BtData btd;
        ICP_get(
            m_anum, "MIN_TEMP",
            &btd.min_permitted, sizeof(btd.min_permitted)
        );
        ICP_get(
            m_anum, "MAX_TEMP",
            &btd.max_permitted, sizeof(btd.max_permitted)
        );
        btd.requested = new_temp;
        btd.error_msg = CORBA::string_dup(
            new_temp > btd.max_permitted ? "Too hot" : "Too cold"
        );
        throw CCS::Thermostat::BadTemp(btd);
    }
    return old_temp;
}

```

Step 6

```

// IDL list operation.

CCS::Controller::ThermometerSeq *
Controller_impl::
list() throw(CORBA::SystemException)
{
    // Create a new thermometer sequence. Because we know
    // the number of elements we will put onto the sequence,
    // we use the maximum constructor.
    CCS::Controller::ThermometerSeq_var listv
        = new CCS::Controller::ThermometerSeq(m_assets.size());
    listv->length(m_assets.size());

    // Loop over the m_assets map and create a
    // reference for each device.
    CORBA::ULong count = 0;
    AssetMap::iterator i;
    for (i = m_assets.begin(); i != m_assets.end(); ++i)
        listv[count++] = i->second->_this();
    return listv._retn();
}

```

Step 7

```

// IDL change operation.

void
Controller_impl::
change(

```

```

    const CCS::Controller::ThermostatSeq & tlist,
          CORBA::Short delta
) throw(CORBA::SystemException, CCS::Controller::EChange)
{
    CCS::Controller::EChange ec;    // Just in case we need it

    // We cannot add a delta value to a thermostat's temperature
    // directly, so for each thermostat, we read the nominal
    // temperature, add the delta value to it, and write
    // it back again.
    CORBA::ULong i;
    for (i = 0; i < tlist.length(); ++i) {
        if (CORBA::is_nil(tlist[i]))
            continue;                // Skip nil references

        // Read nominal temp and update it.
        CCS::TempType tnom = tlist[i]->get_nominal();
        tnom += delta;
        try {
            tlist[i]->set_nominal(tnom);
        }
        catch (const CCS::Thermostat::BadTemp & bt) {
            // If the update failed because the temperature
            // is out of range, we add the thermostat's info
            // to the errors sequence.
            CORBA::ULong len = ec.errors.length();
            ec.errors.length(len + 1);
            ec.errors[len].tmstat_ref = tlist[i];
            ec.errors[len].info = bt.details;
        }
    }

    // If we encountered errors in the above loop,
    // we will have added elements to the errors sequence.
    if (ec.errors.length() != 0)
        throw ec;
}

```

Step 8

```

// IDL find operation

void
Controller_impl::
find(CCS::Controller::SearchSeq & slist)
throw(CORBA::SystemException)
{
    // Loop over input list and look up each device.
    CORBA::ULong listlen = slist.length();
    CORBA::ULong i;

```

```

for (i = 0; i < listlen; ++i) {

    AssetMap::iterator where;    // Iterator for asset map
    int num_found = 0;          // Num matched per iteration

    // Assume we will not find a matching device.
    slist[i].device = CCS::Thermometer::_nil();

    // Work out whether we are searching by asset,
    // model, or location.
    CCS::Controller::SearchCriterion sc = slist[i].key._d();
    if (sc == CCS::Controller::ASSET) {
        // Search for matching asset number.
        where = m_assets.find(slist[i].key.asset_num());
        if (where != m_assets.end())
            slist[i].device = where->second->_this();
    } else {
        // Search for model or location string.
        const char * search_str;
        if (sc == CCS::Controller::LOCATION)
            search_str = slist[i].key.loc();
        else
            search_str = slist[i].key.model_desc();

        // Find first matching device (if any).
        where = find_if(
            m_assets.begin(), m_assets.end(),
            StrFinder(sc, search_str)
        );

        // While there are matches...
        while (where != m_assets.end()) {
            if (num_found == 0) {
                // First match overwrites reference
                // in search record.
                slist[i].device = where->second->_this();
            } else {
                // Each further match appends a new
                // element to the search sequence.
                CORBA::ULong len = slist.length();
                slist.length(len + 1);
                slist[len].key = slist[i].key;
                slist[len].device = where->second->_this();
            }
            ++num_found;

            // Find next matching device with this key.
            where = find_if(
                ++where, m_assets.end(),
                StrFinder(sc, search_str)
            );
        }
    }
}

```

```

        );
    }
}

```

11.2 The server.h File

```

#ifndef server_HH_
#define server_HH_

#include <map>

#ifdef HAVE_STDLIB_H
# include <stdlib.h>
#endif

#include "CCS_skel.h"

#ifdef _MSC_VER
using namespace std;
#endif

class Controller_impl;

class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    // IDL attributes
    virtual CCS::ModelType model()
        throw(CORBA::SystemException);
    virtual CCS::AssetType asset_num()
        throw(CORBA::SystemException);
    virtual CCS::TempType temperature()
        throw(CORBA::SystemException);
    virtual CCS::LocType location()
        throw(CORBA::SystemException);
    virtual void location(const char * loc)
        throw(CORBA::SystemException);

    // Constructor and destructor
    Thermometer_impl(CCS::AssetType anum, const char * location);
    virtual ~Thermometer_impl();

    static Controller_impl * m_ctrl; // My controller

protected:
    const CCS::AssetType m_anum; // My asset number

private:

```

```

    // Helper functions
    CCS::ModelType  get_model();
    CCS::TempType   get_temp();
    CCS::LocType    get_loc();
    void            set_loc(const char * new_loc);

    // Copy and assignment not supported
    Thermometer_impl(const Thermometer_impl &);
    void operator=(const Thermometer_impl &);
};

class Thermostat_impl :
    public virtual POA_CCS::Thermostat,
    public virtual Thermometer_impl {
public:
    // IDL operations
    virtual CCS::TempType  get_nominal()
        throw(CORBA::SystemException);

    virtual CCS::TempType  set_nominal(
        CCS::TempType new_temp
    ) throw(
        CORBA::SystemException,
        CCS::Thermostat::BadTemp
    );

    // Constructor and destructor
    Thermostat_impl(
        CCS::AssetType  anum,
        const char *    location,
        CCS::TempType   nominal_temp
    );
    virtual ~Thermostat_impl() {}

private:
    // Helper functions
    CCS::TempType  get_nominal_temp();
    CCS::TempType  set_nominal_temp(CCS::TempType new_temp)
        throw(CCS::Thermostat::BadTemp);

    // Copy and assignment not supported
    Thermostat_impl(const Thermostat_impl &);
    void operator=(const Thermostat_impl &);
};

class Controller_impl : public virtual POA_CCS::Controller {
public:
    // IDL operations
    virtual CCS::Controller::ThermometerSeq *
        list() throw(CORBA::SystemException);
    virtual void

```

```

        find(CCS::Controller::SearchSeq & slist)
            throw(CORBA::SystemException);
virtual void
    change(
        const CCS::Controller::ThermostatSeq & tlist,
        CORBA::Short                                delta
    ) throw(
        CORBA::SystemException,
        CCS::Controller::EChange
    );

// Constructor and destructor
Controller_impl() {}
virtual ~Controller_impl() {}

// Helper functions to allow thermometers and
// thermostats to add themselves to the m_assets map
// and to remove themselves again.
void add_impl(CCS::AssetType anum, Thermometer_impl * tip);
void remove_impl(CCS::AssetType anum);

private:
// Map of known servants
typedef map<CCS::AssetType, Thermometer_impl *> AssetMap;
AssetMap m_assets;

// Copy and assignment not supported
Controller_impl(const Controller_impl &);
void operator=(const Controller_impl &);

// Function object for the find_if algorithm to search for
// devices by location and model string.
class StrFinder {
public:
    StrFinder(
        CCS::Controller::SearchCriterion    sc,
        const char *                        str
    ) : m_sc(sc), m_str(str) {}
    bool operator()(
        pair<const CCS::AssetType, Thermometer_impl *> & p
    ) const
    {
        switch (m_sc) {
        case CCS::Controller::LOCATION:
            return strcmp(p.second->location(), m_str) == 0;
            break;
        case CCS::Controller::MODEL:
            return strcmp(p.second->model(), m_str) == 0;
            break;
        default:

```

```

        abort();    // Precondition violation
    }
    return 0;      // Stops compiler warning
}
private:
    CCS::Controller::SearchCriterion    m_sc;
    const char *                        m_str;
};

#endif

```

11.3 The server.cpp File

```

#include    <OB/CORBA.h>

#include    <algorithm>
#include    <signal.h>
#include    <string>
#include    <fstream>

#if defined(HAVE_STD_Iostream) || defined(HAVE_STD_STL)
using namespace std;
#endif

#include    "icp.h"
#include    "server.h"

//-----

// Helper function to write a stringified reference to a file.

void
write_ref(const char * filename, const char * reference)
{
    ofstream file(filename);
    if (!file) {
        string msg("Error opening ");
        msg += filename;
        throw msg.c_str();
    }
    file << reference << endl;
    if (!file) {
        string msg("Error writing ");
        msg += filename;
        throw msg.c_str();
    }
    file.close();
    if (!file) {

```

```
        string msg("Error closing ");
        msg += filename;
        throw msg.c_str();
    }
}

//-----

Controller_impl * Thermometer_impl::m_ctrl; // static member

// Helper function to read the model string from a device.

CCS::ModelType
Thermometer_impl::
get_model()
{
    char buf[32];
    if (ICP_get(m_anum, "model", buf, sizeof(buf)) != 0)
        abort();
    return CORBA::string_dup(buf);
}

// Helper function to read the temperature from a device.

CCS::TempType
Thermometer_impl::
get_temp()
{
    short temp;
    if (ICP_get(m_anum, "temperature", &temp, sizeof(temp)) != 0)
        abort();
    return temp;
}

// Helper function to read the location from a device.

CCS::LocType
Thermometer_impl::
get_loc()
{
    char buf[32];
    if (ICP_get(m_anum, "location", buf, sizeof(buf)) != 0)
        abort();
    return CORBA::string_dup(buf);
}

// Helper function to set the location of a device.

void
Thermometer_impl::
```



```
set_loc(const char * loc)
{
    if (ICP_set(m_anum, "location", loc) != 0)
        abort();
}

// Constructor.

Thermometer_impl::
Thermometer_impl(
    CCS::AssetType      anum,
    const char *       location
) : m_anum(anum)
{
    if (ICP_online(anum) != 0)    // Mark device as on-line
        abort();
    set_loc(location);           // Set its location
    m_ctrl->add_impl(anum, this); // Add self to controller's map
}

// Destructor.

Thermometer_impl::
~Thermometer_impl()
{
    try {
        m_ctrl->remove_impl(m_anum); // Remove self from map
        ICP_offline(m_anum);        // Mark device as off-line
    } catch (...) {
        abort();                    // Prevent exceptions from escaping
    }
}

// IDL model attribute.

CCS::ModelType
Thermometer_impl::
model() throw(CORBA::SystemException)
{
    return get_model();
}

// IDL asset_num attribute.

CCS::AssetType
Thermometer_impl::
asset_num() throw(CORBA::SystemException)
{
    return m_anum;
}
```

```

// IDL temperature attribute.

CCS::TempType
Thermometer_impl::
temperature() throw(CORBA::SystemException)
{
    return get_temp();
}

// IDL location attribute accessor.

CCS::LocType
Thermometer_impl::
location() throw(CORBA::SystemException)
{
    return get_loc();
}

// IDL location attribute modifier.

void
Thermometer_impl::
location(const char * loc) throw(CORBA::SystemException)
{
    set_loc(loc);
}

//-----

// Helper function to get a thermostat's nominal temperature.

CCS::TempType
Thermostat_impl::
get_nominal_temp()
{
    short temp;
    if (ICP_get(m_anum, "nominal_temp", &temp, sizeof(temp)) != 0)
        abort();
    return temp;
}

// Helper function to set a thermostat's nominal temperature.

CCS::TempType
Thermostat_impl::
set_nominal_temp(CCS::TempType new_temp)
throw(CCS::Thermostat::BadTemp)
{
    short old_temp;

```

```

// We need to return the previous nominal temperature,
// so we first read the current nominal temperature before
// changing it.
if (ICP_get(
    m_anum, "nominal_temp", &old_temp, sizeof(old_temp)
) != 0) {
    abort();
}

// Now set the nominal temperature to the new value.
if (ICP_set(m_anum, "nominal_temp", &new_temp) != 0) {

    // If ICP_set() failed, read this thermostat's
    // minimum and maximum so we can initialize the
    // BadTemp exception.
    CCS::Thermostat::BtData btd;
    ICP_get(
        m_anum, "MIN_TEMP",
        &btd.min_permitted, sizeof(btd.min_permitted)
    );
    ICP_get(
        m_anum, "MAX_TEMP",
        &btd.max_permitted, sizeof(btd.max_permitted)
    );
    btd.requested = new_temp;
    btd.error_msg = CORBA::string_dup(
        new_temp > btd.max_permitted ? "Too hot" : "Too cold"
    );
    throw CCS::Thermostat::BadTemp(btd);
}
return old_temp;
}

// Constructor.

Thermostat_impl::
Thermostat_impl(
    CCS::AssetType      anum,
    const char *        location,
    CCS::TempType       nominal_temp
) : Thermometer_impl(anum, location)
{
    // Base Thermometer_impl constructor does most of the
    // work, so we need only set the nominal temperature here.
    set_nominal_temp(nominal_temp);
}

// IDL get_nominal operation.

```

```

CCS::TempType
Thermostat_impl::
get_nominal() throw(CORBA::SystemException)
{
    return get_nominal_temp();
}

// IDL set_nominal operation.

CCS::TempType
Thermostat_impl::
set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    return set_nominal_temp(new_temp);
}

//-----

// Helper function for thermometers and thermostats to
// add themselves to the m_assets map.

void
Controller_impl::
add_impl(CCS::AssetType anum, Thermometer_impl * tip)
{
    m_assets[anum] = tip;
}

// Helper function for thermometers and thermostats to
// remove themselves from the m_assets map.

void
Controller_impl::
remove_impl(CCS::AssetType anum)
{
    m_assets.erase(anum);
}

// IDL list operation.

CCS::Controller::ThermometerSeq *
Controller_impl::
list() throw(CORBA::SystemException)
{
    // Create a new thermometer sequence. Because we know
    // the number of elements we will put onto the sequence,
    // we use the maximum constructor.
    CCS::Controller::ThermometerSeq_var listv
        = new CCS::Controller::ThermometerSeq(m_assets.size());
}

```

```

    listv->length(m_assets.size());

    // Loop over the m_assets map and create a
    // reference for each device.
    CORBA::ULong count = 0;
    AssetMap::iterator i;
    for (i = m_assets.begin(); i != m_assets.end(); ++i)
        listv[count++] = i->second->_this();
    return listv._retn();
}

// IDL change operation.

void
Controller_impl::
change(
    const CCS::Controller::ThermostatSeq & tlist,
    CORBA::Short delta
) throw(CORBA::SystemException, CCS::Controller::EChange)
{
    CCS::Controller::EChange ec;    // Just in case we need it

    // We cannot add a delta value to a thermostat's temperature
    // directly, so for each thermostat, we read the nominal
    // temperature, add the delta value to it, and write
    // it back again.
    CORBA::ULong i;
    for (i = 0; i < tlist.length(); ++i) {
        if (CORBA::is_nil(tlist[i]))
            continue;                // Skip nil references

        // Read nominal temp and update it.
        CCS::TempType tnom = tlist[i]->get_nominal();
        tnom += delta;
        try {
            tlist[i]->set_nominal(tnom);
        }
        catch (const CCS::Thermostat::BadTemp & bt) {
            // If the update failed because the temperature
            // is out of range, we add the thermostat's info
            // to the errors sequence.
            CORBA::ULong len = ec.errors.length();
            ec.errors.length(len + 1);
            ec.errors[len].tmstat_ref = tlist[i];
            ec.errors[len].info = bt.details;
        }
    }

    // If we encountered errors in the above loop,
    // we will have added elements to the errors sequence.

```

```

        if (ec.errors.length() != 0)
            throw ec;
    }

    // IDL find operation

    void
    Controller_impl::
    find(CCS::Controller::SearchSeq & slist)
    throw(CORBA::SystemException)
    {
        // Loop over input list and look up each device.
        CORBA::ULong listlen = slist.length();
        CORBA::ULong i;
        for (i = 0; i < listlen; ++i) {

            AssetMap::iterator where;    // Iterator for asset map
            int num_found = 0;          // Num matched per iteration

            // Assume we will not find a matching device.
            slist[i].device = CCS::Thermometer::_nil();

            // Work out whether we are searching by asset,
            // model, or location.
            CCS::Controller::SearchCriterion sc = slist[i].key._d();
            if (sc == CCS::Controller::ASSET) {
                // Search for matching asset number.
                where = m_assets.find(slist[i].key.asset_num());
                if (where != m_assets.end())
                    slist[i].device = where->second->_this();
            } else {
                // Search for model or location string.
                const char * search_str;
                if (sc == CCS::Controller::LOCATION)
                    search_str = slist[i].key.loc();
                else
                    search_str = slist[i].key.model_desc();

                // Find first matching device (if any).
                where = find_if(
                    m_assets.begin(), m_assets.end(),
                    StrFinder(sc, search_str)
                );

                // While there are matches...
                while (where != m_assets.end()) {
                    if (num_found == 0) {
                        // First match overwrites reference
                        // in search record.
                        slist[i].device = where->second->_this();
                    }
                }
            }
        }
    }

```

```
        } else {
            // Each further match appends a new
            // element to the search sequence.
            CORBA::ULong len = slist.length();
            slist.length(len + 1);
            slist[len].key = slist[i].key;
            slist[len].device = where->second->_this();
        }
        ++num_found;

        // Find next matching device with this key.
        where = find_if(
            ++where, m_assets.end(),
            StrFinder(sc, search_str)
        );
    }
}
}
```

//-----

```
void
run(CORBA::ORB_ptr orb)
{
    // Get reference to Root POA.
    CORBA::Object_var obj
        = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa
        = PortableServer::POA::_narrow(obj);

    // Create a controller and set static m_ctrl member
    // for thermostats and thermometers.
    Controller_impl ctrl_servant;
    Thermometer_impl::m_ctrl = &ctrl_servant;

    // Write controller stringified reference to ctrl.ref.
    CCS::Controller_var ctrl = ctrl_servant._this();
    CORBA::String_var str = orb->object_to_string(ctrl);
    write_ref("ctrl.ref", str);

    // Create a few devices. (Thermometers have odd asset
    // numbers, thermostats have even asset numbers.)
    Thermometer_impl thermo1(2029, "Deep Thought");
    Thermometer_impl thermo2(8053, "HAL");
    Thermometer_impl thermo3(1027, "ENIAC");

    Thermostat_impl tmstat1(3032, "Colossus", 68);
    Thermostat_impl tmstat2(4026, "ENIAC", 60);
    Thermostat_impl tmstat3(4088, "ENIAC", 50);
}
```

```

    Thermostat_impl tmstat4(8042, "HAL", 40);

    // Write a thermostat reference to tmstat.ref.
    CCS::Thermostat_var tmstat = tmstat1._this();
    str = orb->object_to_string(tmstat);
    write_ref("tmstat.ref", str);

    // Activate POA manager
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Accept requests
    orb->run();
}

//-----

static CORBA::ORB_var orb; // Global, so handler can see it.

//-----

#ifdef WIN32
BOOL
handler(DWORD)
{
    // Inform JTC of presence of new thread
    JTCAdoptCurrentThread adopt;

    // Terminate event loop
    try {
        if (!CORBA::is_nil(orb))
            orb->shutdown(false);
    } catch (...) {
        // Can't throw here...
    }
    return TRUE;
}
#else
extern "C"
void handler(int)
{
    // Ignore further signals
    struct sigaction ignore;
    ignore.sa_handler = SIG_IGN;
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGTERM, &ignore, (struct sigaction *)0) == -1)
        abort();
}

```



```

        if (sigaction(SIGHUP, &ignore, (struct sigaction *)0) == -1)
            abort();

        // Terminate event loop
        try {
            if (!CORBA::is_nil(orb))
                orb->shutdown(false);
        } catch (...) {
            // Can't throw here...
        }
    }
#endif

//-----

int
main(int argc, char* argv[])
{
    // Install signal handler for cleanup
#ifdef WIN32
    BOOL rc = SetConsoleCtrlHandler((PHANDLER_ROUTINE)handler, TR
UE);
    if (!rc)
        abort();
#else
    struct sigaction sa;           // New signal state
    sa.sa_handler = handler;      // Set handler function
    sigfillset(&sa.sa_mask);     // Mask all other signals
                                // while handler runs
    sa.sa_flags = 0 | SA_RESTART; // Restart interrupted syscal
ls

    if (sigaction(SIGINT, &sa, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGHUP, &sa, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGTERM, &sa, (struct sigaction *)0) == -1)
        abort();
#endif

    // Initialize the ORB and start working...
    int status = 0;
    try {
        orb = CORBA::ORB_init(argc, argv);
        run(orb);
    } catch (const CORBA::Exception & ex) {
        cerr << "Uncaught CORBA exception: " << ex << endl;
        status = 1;
    } catch (...) {
        cerr << "Uncaught non-CORBA exception" << endl;
    }
}

```

```
        status = 1;
    }

    // Destroy the ORB.
    if (!CORBA::is_nil(orb)) {
        try {
            orb->destroy();
        } catch (const CORBA::Exception & ex) {
            cerr << "Cannot destroy ORB: " << ex << endl;
            status = 1;
        }
    }

    return status;
}
```

12. The Portable Object Adapter (POA)

Summary

This unit presents the Portable Object Adapter (POA) in detail and covers most of the functionality of the POA interfaces. It explains how to create persistent objects and how to link database state of objects to object references and servants. In addition, this unit covers how to support life cycle operations for CORBA objects.

Objectives

By the completion of this unit, you will be able to create servers that permit clients to create and destroy objects and that offer objects whose state is persistent. In addition, you will have a thorough understanding of the functionality of the POA, including how to control request flow, initialization, finalization, and memory management techniques.

Interface Overview

The Portable Object Adapter provides a number of core interfaces, all part of the **PortableServer** module:

- **POA**
- **POAManager**
- **Servant**
- POA Policies (seven interfaces)
- Servant Managers (three interfaces)
- **POACurrent**
- **AdapterActivator**

Of these, the first five are used regularly in almost every server; **POACurrent** and **AdapterActivator** support advanced or unusual implementation techniques.



1
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.1 Interface Overview

The interfaces to the POA are defined in IDL in the **PortableServer** module:

- POA

The POA interface is the central server-side interface and contains quite a large number of operations. POAs are concerned with tasks such as keeping track of which servants are currently instantiated and their addresses in memory, the activation and deactivation of servants, the creation of object references, and various other life cycle issues (such as permitting a servant to be deleted at a time when no operation invocation is in progress in that servant).

- POAManager

Conceptually a POA manager represents a transport endpoint that is used by one or more POAs. POA managers control the flow of requests into POAs.

- Servant

The IDL Servant type is defined in the specification as follows:

```
module PortableServer {
    native Servant;
    // ...
};
```

`native` is an IDL keyword that may be used only by OMG-defined specifications and ORB vendors. The `native` keyword indicates that the corresponding IDL construct is highly dependent on the target programming language and therefore does not have an IDL interface;

instead, each language mapping must specify how the native type is represented as programming-language artifacts for a specific implementation language.

The `native` keyword was added to IDL after earlier attempts to specify the interface for servants were unsuccessful—as it turns out, to get elegant language mappings, servant implementations must use features that are specific to each programming language and cannot be expressed in IDL. (This is not surprising when you consider that servants straddle the boundary between language-independent IDL definitions and language-specific implementations.)

- POA Policies (seven interfaces)

Each POA has seven policies that are associated with that POA when it is created (and remain in effect without change for the life time of each POA). The policies control aspects of the implementation techniques that are used by servants using that POA, such as the threading model and whether object references are persistent or transient.

- Servant Managers (three interfaces)

Servant managers permit lazy instantiation of servants. Instead of requiring a server to keep a separate C++ object instantiated in memory for each CORBA object, servant managers allow servers to be written such that C++ instances are created on demand for only those servants that are actually used.

- POACurrent

POACurrent is an object that provides information about a currently executing operation to the operation's implementation. This information is useful mainly for interceptors (which are used to implement functionality required by services such as the Transaction and Security Service).

- AdapterActivator

An adapter activator is a callback object that permits you to create an object adapter on demand, when a request arrives for it, instead of forcing you keep all adapters active in memory at all times. Adapter activators are useful mainly to implement optimistic caching schemes, where entire groups of objects are instantiated in memory when a request for any one of the objects in the group is received.

Functions of a POA

Each POA forms a namespace for servants.

All servants that use the same POA share common implementation characteristics, determined by the POA's policies. (The Root POA has a fixed set of policies.)

Each servant has exactly one POA, but many servants may share the same POA.

The POA tracks the relationship between object references, object IDs, and servants (and so is intimately involved in their life cycle).

POAs map between object references and the associated object ID and servants and map an incoming request onto the correct servant that incarnates the corresponding CORBA object.



2
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.2 Functions of a POA

The main purpose of a POA is to bridge the gap between the abstract notion of a CORBA object and the concrete representation of that object's behavior in form of a servant. In other words, POAs can be seen as a mapping mechanism that associates incoming requests with the correct C++ object in the server's memory.

A server can contain any number of POAs besides the Root POA (which is always present). Each POA, when it is created, is associated with a set of seven policies. These policies remain with the POA for its life time (that is, they become immutable once the POA is created). The policies determine the implementation characteristics of the servants associated with the POA, as well as aspects of object references (such as whether references are transient or persistent).

A POA can have any number of servants, but each servant belongs to exactly one POA.¹

1. The specification (at least in theory) permits a single servant to be associated with more than one POA at a time. However, this must be considered a defect because it creates a number of semantic conflicts; we *strongly* recommend that you never use the same servant with more than one POA.

Functions of a POA Manager

A POA manager is associated with a POA when the POA is created. Thereafter, the POA manager for a POA cannot be changed.

A POA manager controls the flow of requests into one or more POAs.

A POA manager is associated with a POA when the POA is created. Thereafter, the POA manager for a POA cannot be changed.

A POA manager is in one of four possible states:

- **Active:** Requests are processed normally
- **Holding:** Requests are queued
- **Discarding:** Requests are rejected with **TRANSIENT**
- **Inactive:** Requests are rejected; clients may be redirected to a different server instance to try again.



3
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.3 Functions of a POA Manager

A POA manager acts as a gate that controls the flow of requests to one or more associated POAs. Conceptually, a POA manager represents a transport endpoint (such as a host–port pair for TCP/IP). A POA is associated with its POA manager when the POA is created; thereafter, the POA manager for a POA cannot be changed.

A POA manager is in one of four possible states:

- **Active**

This is the normal state in which the POA manager passes an incoming request to the target POA, which in turn passes the request to the correct servant.

- **Holding**

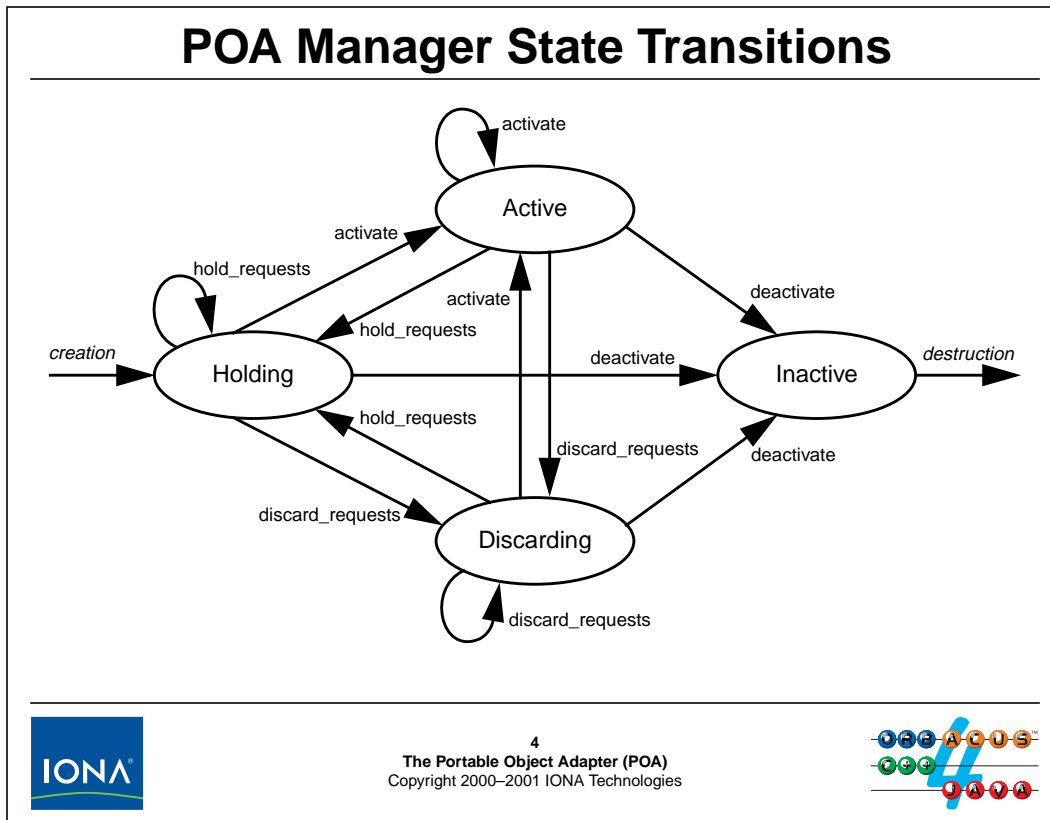
In this state, the POA manager holds requests in a queue. Once the POA manager enters the active state, it passes the requests to their destination POAs.

- **Discarding**

Incoming requests are rejected with a **TRANSIENT** exception. This exception indicates to the client that the request cannot be delivered right now, but that it may work if retransmitted again later.

- **Inactive**

Requests are rejected; however, instead of raising an exception, the POA manager indicates to the client that the connection to the server is no longer usable. Depending on how the client is configured, this may result in an attempt by the client to locate a new instance of the server.



12.4 POA Manager State Transitions

The above diagram shows the possible state transitions. The arcs in the diagram are labeled with the corresponding IDL operation name. Initially, when it is first created, a POA manager starts out in the holding state. Before the ORB delivers requests to POAs associated with that POA manager, you must transition to the active state (see page 9-14).

Note that, once the POA manager enters the inactive state, it cannot be reactivated again and the only remaining transition is the destruction of the POA manager. POA managers are not destroyed explicitly; instead, a POA manager is destroyed once the last of its POAs is destroyed. You can freely transition among the remaining states by invoking the corresponding transition operation.

The IDL for the POAManager interface is as follows:

```
module PortableServer {
    // ...
    interface POAManager {
        exception AdapterInactive {};

        enum State { HOLDING, ACTIVE, DISCARDING, INACTIVE };

        State    get_state();
        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);
};
};

```

State get_state()

The `get_state` operation returns the current state of the of the POA manager as an enumerated value.

void activate() raises(AdapterInactive)

The `activate` operation transitions the POA manager into the active state. If the POA manager was previously in the holding state, the queued requests are dispatched in the order in which they were received. Attempts to activate an inactive POA manager raise `AdapterInactive`.

void hold_requests(in boolean wait_for_completion) raises(AdapterInactive)

The `hold_requests` operation transitions the POA manager into the holding state. Incoming requests are queued up to some implementation-dependent limit.² If `wait_for_completion` is false, the operation returns immediately; otherwise, it queues incoming requests but waits until all currently executing requests have completed before returning control to the caller. If you call this operation with `wait_for_completion` set to true from within a servant that has a POA that is controlled by this POA manager, the operation raises `BAD_INV_ORDER` (because it would deadlock otherwise). Attempts to invoke this operation on an inactive POA manager raise `AdapterInactive`.

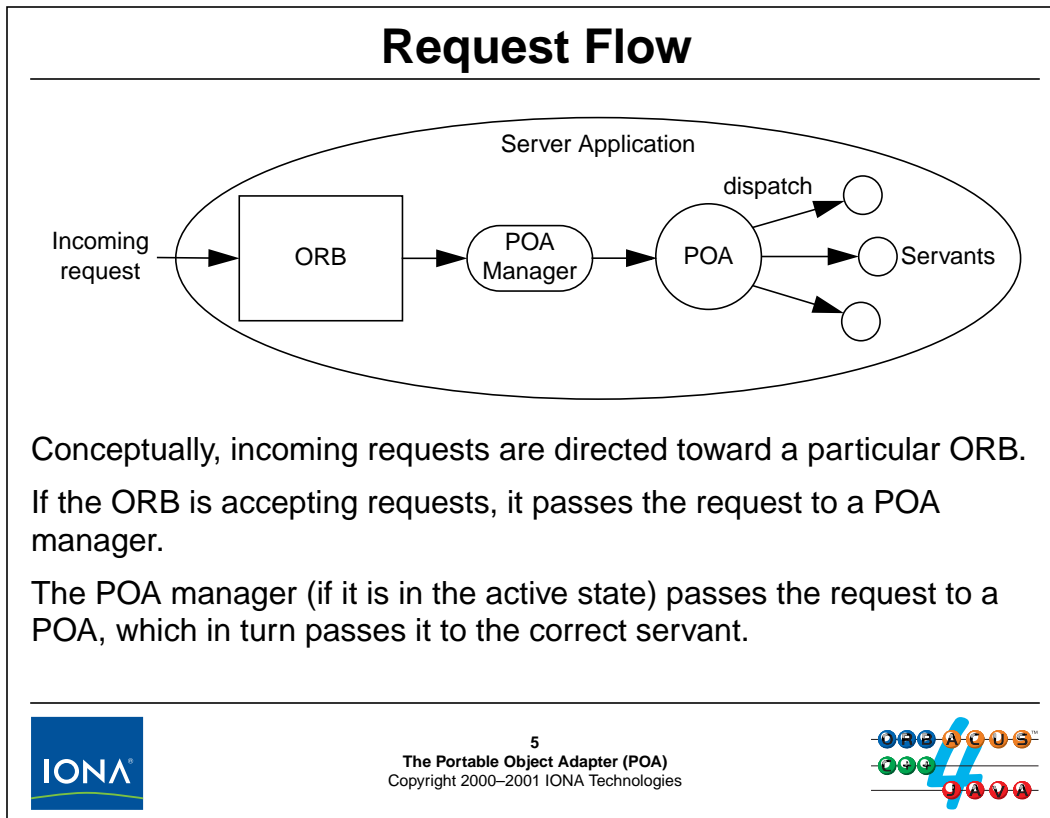
void discard_requests(in boolean wait_for_completion) raises(AdapterInactive)

The `discard_requests` operation transitions the POA manager into the discarding state. Incoming requests are rejected with a `TRANSIENT` exception. The `wait_for_completion` parameter has the same semantics as for `hold_requests`. Attempts to invoke this operation on an inactive POA manager raise `AdapterInactive`.

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

The `deactivate` operation transitions the POA manager into the inactive state. Incoming requests are faced with a closed connection; the behavior that is visible to the client in this case depends on the type of object reference (transient or persistent) and the rebinding policy of the client. The `wait_for_completion` parameter has the same semantics as for `discard_requests`. The `etherealize_objects` parameter determines whether or not servant activators will be asked to destroy existing servants. (See page 15-4.) Attempts to invoke this operation on an inactive POA manager raise `AdapterInactive`.

2. In ORBacus, the underlying transport is used as the queueing mechanism. This means that, due to TCP/IP flow control, leaving a POA manager in the holding state may cause flow control to affect the client (if transport buffers fill up completely) and cause the client to block in an operation until the POA manager transitions out of the holding state.



12.5 Request Flow

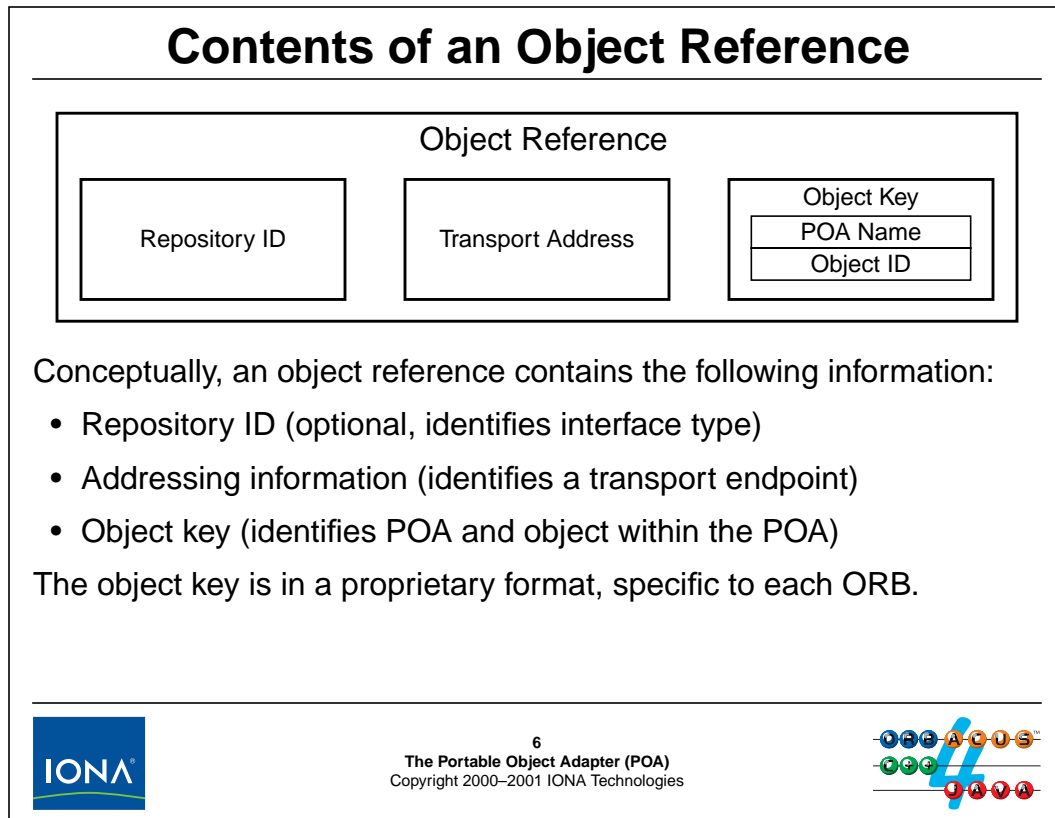
The general request flow into a server is shown above. Note that the diagram represents a conceptual view only. In the implementation, requests are not physically passed in this way for efficiency reasons.

Conceptually, the request is directed toward a particular ORB within a server.³ If the ORB is processing requests (that is, has created a dispatch loop by calling `ORB::run` or is dispatching requests explicitly via `ORB::work_pending` and `ORB::perform_work`), the request is passed to the POA manager.

The POA manager determines whether the request is queued, discarded, or passed on. If the POA manager is in the active state, the request is passed to the correct POA.

The POA determines the relationship between the CORBA reference that was used to make the call (and, therefore, the CORBA object represented by that reference) and the servant, and passes the request to the correct servant.

3. It is possible to instantiate multiple ORBs by calling `ORB_init` more than once with different ORB IDs. This is useful if you, for example, require different dispatch policies to be used for different objects.



12.6 Contents of an Object Reference

For a server to correctly dispatch incoming requests to the correct servant, and for a client to correctly connect to the a server, an object reference must contain a minimum amount of information. In particular, it must contain an address (such as a host–port pair) that the client can use to contact the server, and it must contain information that, once a request is passed to the server, identifies the particular target object for an invocation.

As shown above, an object reference contains exactly that. The transport information, which (at least for IIOP) is standardized, enables the client to connect to the correct server. When a client sends an invocation to a particular server, it sends the object key with the request. The object key, internally, contains both a POA name and an object ID. The POA name enables the receiving ORB to identify the correct POA to pass the request to. In turn, the POA uses the object ID part of the object key to identify the specific servant that must handle the request.

Note that the object key is in a proprietary format, specific to each ORB vendor, and is never looked at except by the server that created it. Other clients and servers in a CORBA system treat the object key as an opaque blob of data.

Policies

Each POA is associated with a set of seven policies when it is created. Policies control implementation characteristics of object references and servants.

The **CORBA** module provides a **Policy** abstract base interface:

```

module CORBA {
    typedef unsigned long PolicyType;

    interface Policy {
        readonly attribute PolicyType policy_type;
        Policy copy();
        void destroy();
    };
    typedef sequence<Policy> PolicyList;
    // ...
};

```



7
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.7 Policies

Each POA is associated with a set of seven policies when the POA is created. The policies determine implementation characteristics of object references created by that POA, as well as of servants that are associated with that POA, such as the life time of references or the threading model to be used for request dispatch.

Policies are used in contexts other than the POA.⁴ For that reason, the CORBA module provides an abstract base interface for policies from which all concrete policies are derived. The `Policy` interface provides only the basic features that are common to all policies. The `policy_type` attribute identifies the specific kind of policy. (Policy numbers are assigned by the OMG.)

The `copy` operation returns a (polymorphic) copy of a policy, and the `destroy` operation destroys a policy object. The specification requires you to call `destroy` on a policy object you have created before releasing its last reference.⁵

4. CORBA is using policies as a general abstraction for a quality-of-service (QoS) framework. For example, the real-time and messaging specifications both use policies to control various operational aspects of an ORB.

5. This is a rather useless requirement because policy objects are locality constrained (implemented as library objects) and the ORB can reclaim their resources automatically, when the last reference to a policy object is released. As a result, all ORBs we are aware of implement `destroy` as a no-op, so you don't suffer a resource leak if you do not call `destroy` before releasing a policy reference.

POA Policies

The **PortableServer** module contains seven interfaces that are derived from the **CORBA::Policy** interface:

- **LifespanPolicy**
- **IdAssignmentPolicy**
- **IdUniquenessPolicy**
- **ImplicitActivationPolicy**
- **RequestProcessingPolicy**
- **ServantRetentionPolicy**
- **ThreadPolicy**



8
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.8 POA Policies

The seven POA policies are all derived from the **CORBA::Policy** base interface. Each controls a different aspect of the implementation of an object. We briefly describe the purpose of each policy here and discuss it in more detail as we present the relevant implementation techniques.

- **LifespanPolicy**

The life span policy controls whether a reference is transient or persistent. A transient reference works only for as long as its POA remains in existence and then becomes permanently non-functional. Therefore, transient references do not survive server shut-down. Persistent references continue to denote the same object even if the server is shut down and restarted.

- **IdAssignmentPolicy**

The ID assignment policy controls whether the object ID that is part of the object key of every reference is created by the ORB or is provided by the application. Transient references usually use IDs that are created by the ORB, whereas persistent reference usually use IDs that are provided by the application.

- **IdUniquenessPolicy**

The ID uniqueness policy determines how object references are mapped to C++ servants. You can choose to use one servant for each CORBA object that is provided by a server, or you can choose to incarnate multiple CORBA objects with the same C++ servant.

- **ImplicitActivationPolicy**

The implicit activation policy determines whether a newly instantiated servant must be explicitly activated (registered with the ORB) or will be activated automatically when you first

create a reference for the servant. Transient references usually use implicitly activated servants, whereas persistent references must use explicitly activated servants.

- **RequestProcessingPolicy**

The request processing policy controls whether the POA maintains the object ID-to-servant associations for you (either to multiple servants or a single servant). You can also choose to maintain these associations yourself. Doing so is more work, but provides more powerful implementation choices.

- **ServantRetentionPolicy**

The servant retention policy controls whether you keep your servants in memory at all times or instantiate them on demand, as requests arrive from clients.

- **ThreadPolicy**

The thread policy controls whether requests are dispatched on a single thread or multiple threads.

POA Creation

The POA interface allows you to create other POAs:

```
module PortableServer {
    interface POAManager;

    exception AdapterAlreadyExists {};
    exception InvalidPolicy { unsigned short index; };
    interface POA {
        POA create_POA(
            in string          adapter_name,
            in POAManager      manager,
            in CORBA::PolicyList policies;
        ) raises(AdapterAlreadyExists, InvalidPolicy);
        readonly attribute POAManager the_POAManager;
        // ...
    };
    // ...
};
```



9
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.9 POA Creation

The POA interface provides an operation that creates POAs. (This is an example of the factory pattern, which we will examine in more detail in Section 12.24.) Initially, the only POA you have access to is the Root POA, returned by `resolve_initial_references`. In order to create other POAs, you call the `create_POA` operation on the Root POA or, once you have created other POAs, on a POA other than the Root POA.

The newly created POA becomes a child of the POA on which you invoke `create_POA`. In other words, if you have multiple POAs in a server, they are arranged into a hierarchy with the Root POA at the top. You control the shape of the hierarchy by choosing the POA on which you call `create_POA`.

Each POA has a name, controlled by setting the `adapter_name` parameter. You can choose any name you deem suitable, but you must ensure that no other sibling POA has the same name; otherwise, `create_POA` raises an `AdapterAlreadyExists` exception. As with a directory tree, the name of a POA must be unique only within the context of its parent, so you can have several POAs with the same name, as long as they have different parent POAs.

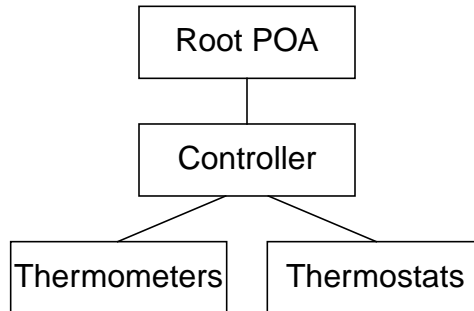
The `manager` parameter controls whether the new POA will use a separate POA manager or share a POA manager with other POAs: if you pass a `nil` reference, a new POA manager will be created for this POA; otherwise, you can pass a reference to an existing POA manager⁶ and the new POA will be added to the list of POAs controlled by that manager.

The `policies` parameter sets the policies to be applied to the new POA. The policy list can contain up to seven distinct POA policies. If you supply a value for the same policy more than

6. The `the_POAManager` read-only attribute on the POA interface returns the POA manager reference for a POA.

once, or if one of the policies does not apply to the POA, the `create_POA` raises `InvalidPolicy`; the `index` member of the exception indicates the first policy that was found to be in error. You can create a POA with an empty policy sequence. If you do, each of the seven policies gets a default value.

For now, let us look at a simple example. The code that follows creates the following POA hierarchy:



For now, we will use the simplest way to create this hierarchy, using the default policies for all POAs, and using a separate POA manager for each POA.

```
// Initialize ORB and get Root POA
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
CORBA::Object_var obj =
    orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa =
    PortableServer::POA::_narrow(obj);
assert(!CORBA::is_nil(root_poa));

// Create empty policy list
CORBA::PolicyList policy_list;

// Create Controller POA
PortableServer::POA_var ctrl_poa = root_poa->create_POA(
    "Controller",
    PortableServer::POAManager::_nil(),
    policy_list);

// Create Thermometer POA as a child of the Controller POA
PortableServer::POA_var thermometer_poa = ctrl_poa->create_POA(
    "Thermometers",
    PortableServer::POAManager::_nil(),
    policy_list);

// Create Thermostat POA as a child of the Controller POA
PortableServer::POA_var thermostat_poa = ctrl_poa->create_POA(
    "Thermostats",
    PortableServer::POAManager::_nil(),
    policy_list);
```

Because the code passes a nil reference as the `manager` parameter, each POA ends up with its own, separate POA manager; because the code passes an empty policy sequence as the `policies` parameter, each POA gets created with the default policies.

If we wanted to use the same POA manager for all four POAs, we could write the code as follows:

```
// Initialize ORB and get Root POA
PortableServer::POA_var root_poa = ...;

// Create empty policy list
CORBA::PolicyList policy_list;

// Get the Root POA manager
PortableServer::POAManager_var mgr = root_poa->the_POAManager();

// Create Controller POA, using the Root POA's manager
PortableServer::POA_var ctrl_poa = root_poa->create_POA(
    "Controller",
    mgr,
    policy_list);

// Create Thermometer POA as a child of the Controller POA,
// using the Root POA's manager
PortableServer::POA_var thermometer_poa = ctrl_poa->create_POA(
    "Thermometers",
    mgr,
    policy_list);

// Create Thermostat POA as a child of the Controller POA,
// using the Root POA's manager
PortableServer::POA_var thermostat_poa = ctrl_poa->create_POA(
    "Thermostats",
    mgr,
    policy_list);
```

This code is almost identical to the preceding example. The only difference is that the code first gets a reference to the Root POA's manager by reading the `the_POAManager` attribute of the Root POA, and then passes that manager's reference to the three `create_POA` calls.

The Life Span Policy

The life span policy controls whether references are transient or persistent. The default is **TRANSIENT**.

```
enum LifespanPolicyValue { TRANSIENT, PERSISTENT };
```

```
interface LifespanPolicy : CORBA::Policy {
    readonly attribute LifespanPolicyValue value;
};
```

You should combine **PERSISTENT** with:

- **ImplicitActivationPolicy: NO_IMPLICIT_ACTIVATION**
- **IdAssignmentPolicy: USER_ID**

You should combine **TRANSIENT** with:

- **ImplicitActivationPolicy: IMPLICIT_ACTIVATION**
- **IDAssignmentPolicy: SYSTEM_ID**



11
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.11 The Life Span Policy

The life span policy controls whether references are transient or persistent. Transient references are usually created for objects that only exist temporarily, to support short-lived client–server interactions. On the other hand, persistent references are usually created if a server acts as a front end to some form of persistent store, such as a document retrieval service, which makes it desirable to pass out references to clients that can survive server shut-down.

Although the specification does not require it, you should combine the **PERSISTENT** life span policy with an implicit activation policy value of **NO_IMPLICIT_ACTIVATION**, and an ID assignment policy value of **USER_ID**.⁷

Note that, to create persistent objects, you must do a few things other than using the **PERSISTENT** life span policy. We discuss these details in Section 12.21.

7. While the other two combinations are legal, they do not have realistic use cases.

The ID Assignment Policy

The ID assignment policy controls whether object IDs are created by the ORB or by the application. The default is **SYSTEM_ID**.

```
enum IdAssignmentPolicyValue { USER_ID, SYSTEM_ID };
```

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

You should combine **USER_ID** with:

- **ImplicitActivationPolicy: NO_IMPLICIT_ACTIVATION**
- **LifespanPolicy: PERSISTENT**

You should combine **SYSTEM_ID** with:

- **ImplicitActivationPolicy: IMPLICIT_ACTIVATION**
- **LifespanPolicy: TRANSIENT**



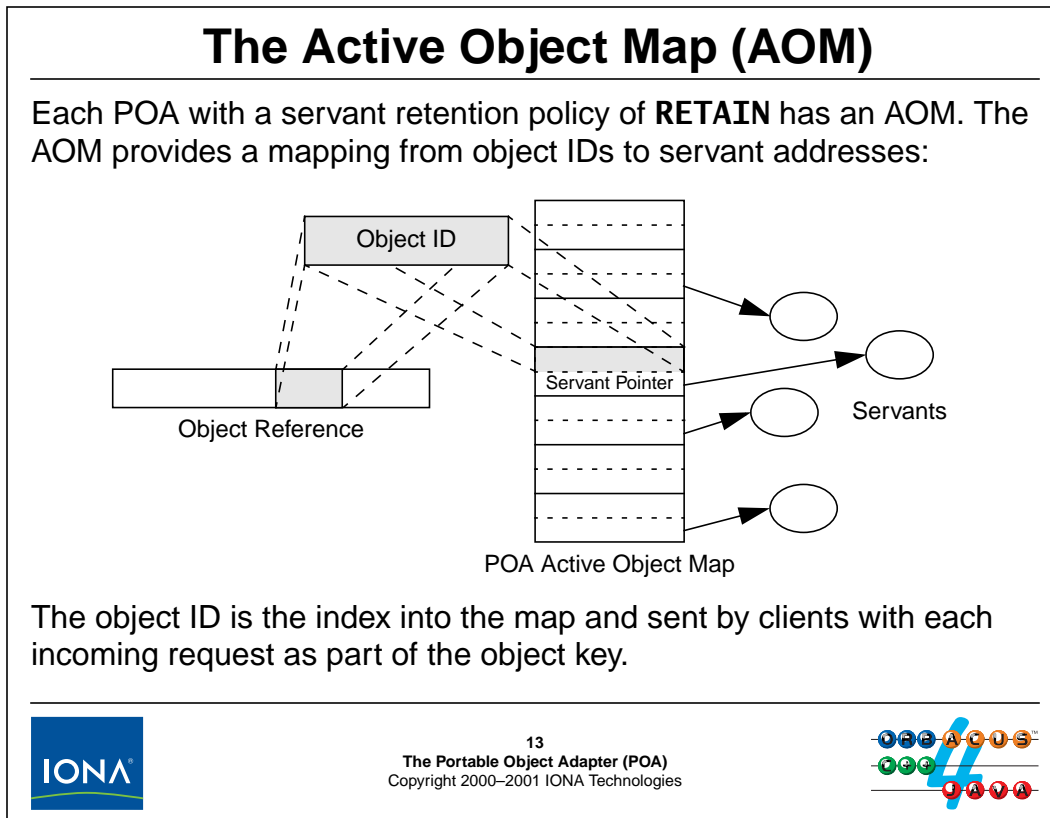
12
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.12 The ID Assignment Policy

The ID assignment policy controls whether object IDs (which end up being embedded into the object key inside references) are generated by the POA or supplied explicitly by the application. As we saw on page 12-9, the object ID ultimately identifies which servant is to handle an incoming request. This means that each ID denotes exactly one servant at a time.

If the ID assignment policy is **SYSTEM_ID**, the POA automatically creates unique identifiers. If the policy is **USER_ID**, the POA rejects attempts to use the same ID a second time.



12.13 The Active Object Map (AOM)

The POA maintains a lookup table known as the Active Object Map (AOM) that associates each object ID with the address of the corresponding servant in memory.⁸ This means that each object ID must uniquely identify a servant; otherwise, the POA could end up with a single object ID designating two servants simultaneously and would not know which servant to give the request to.

8. You can change the setting of the servant retention policy to `NON_RETAIN` in order to provide your own AOM.

The ID Uniqueness Policy

The ID uniqueness policy controls whether a single servant can represent more than one CORBA object. The default is **UNIQUE_ID**):

```
enum IdUniquenessPolicyValue { UNIQUE_ID, MULTIPLE_ID };
```

```
interface IdUniquenessPolicy : CORBA::Policy {
    readonly attribute IdUniquenessPolicyValue value;
};
```

- **UNIQUE_ID** enforces that no servant can appear in the AOM more than once.
- **MULTIPLE_ID** permits the same servant to be pointed at by more than one entry in the AOM.

For **MULTIPLE_ID**, an operation implementation can ask its POA for the object ID for the current invocation.



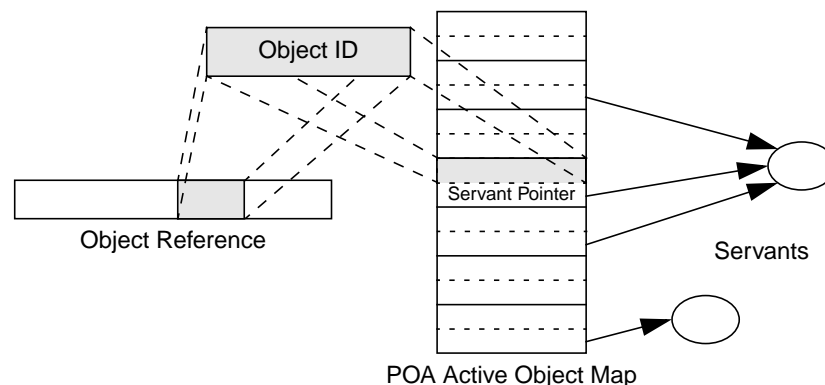
14
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.14 The ID Uniqueness Policy

You can choose to provide a separate C++ servant for each CORBA object by setting the ID uniqueness policy to **UNIQUE_ID**. This setting enforces that no servant can appear more than once in the AOM, so each CORBA object is incarnated by a separate servant. (This corresponds to the diagram shown on page 12-20.)

If you set the policy to **MULTIPLE_ID**, a single servant can incarnate more than one CORBA object simultaneously:



MULTIPLE_ID is useful if a server must provide access to a large number of CORBA objects with limited memory footprint. The cost of this increased scalability is that the identity of the CORBA object for a request is no longer implicit in the particular servant instance. Instead, the implementation of each operation must associate the object ID with the correct object state at run time.

The Servant Retention Policy

The servant retention policy controls whether a POA has an AOM. (The default is **RETAIN**).

```
enum ServantRetentionPolicyValue { RETAIN, NON_RETAIN };
```

```
interface ServantRetentionPolicy : CORBA::Policy {
    readonly attribute ServantRetentionPolicyValue value;
};
```

NON_RETAIN requires a request processing policy of **USE_DEFAULT_SERVANT** or **USE_SERVANT_MANAGER**.

With **NON_RETAIN** and **USE_DEFAULT_SERVANT**, the POA maps incoming requests to a nominated default servant.

With **NON_RETAIN** and **USE_SERVANT_MANAGER**, the POA calls back into the server application code for each incoming request to map the request to a particular servant.



15
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.15 The Servant Retention Policy

The servant retention policy controls whether an AOM is present (**RETAIN**) or absent (**NON_RETAIN**). Obviously, for the **NON_RETAIN** case, this deprives the POA of automatically mapping the object ID for an incoming request to the correct C++ servant. Depending on the setting of the request processing policy, the ORB either maps all requests to a nominated default servant (**USER_DEFAULT_SERVANT**) or it calls back into the application code to supply it with a servant for the request (**USE_SERVANT_MANAGER**).

Most servers that must provide access to a large number of CORBA objects simultaneously use the **NON_RETAIN** policy to limit the number of servants that must be in memory simultaneously.

The Request Processing Policy

The request processing policy controls whether a POA uses static AOM, a default servant, or instantiates servants on demand. (The default is **USE_ACTIVE_OBJECT_MAP_ONLY**.)

```
enum RequestProcessingPolicyValue {
    USE_ACTIVE_OBJECT_MAP_ONLY,
    USE_DEFAULT_SERVANT,
    USE_SERVANT_MANAGER
};

interface RequestProcessingPolicy : CORBA::Policy {
    readonly attribute RequestProcessingPolicyValue value;
};
```

USE_DEFAULT_SERVANT requires **MULTIPLE_ID**.

USE_ACTIVE_OBJECT_MAP_ONLY requires **RETAIN**.



16
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.16 The Request Processing Policy

The most simple approach to implementing a server is to use a request processing policy of **USE_ACTIVE_OBJECT_MAP_ONLY** together with a servant retention policy of **RETAIN**. This combination uses a separate servant for each CORBA object (see page 12-20), and all servants are permanently in memory. If a request arrives for an object ID that is not in the AOM, the request raises **OBJECT_NOT_EXIST** in the client.

The **USE_DEFAULT_SERVANT** policy can be combined with both **RETAIN** and **NON_RETAIN** policies:

- If used with **NON_RETAIN**, the POA passes all incoming requests to a nominated default servant (established by calling the **set_servant** operation on the POA).
- If used with **RETAIN**, the POA first looks for an instantiated servant with the given object ID. If one is found in the AOM, the request is passed to that servant; otherwise, the request is passed to the default servant.

USE_SERVANT_MANAGER can be used with either **RETAIN** or **NON_RETAIN**:

- If used with **RETAIN** and a request arrives for which no entry can be found in the AOM, the ORB makes a callback to an application-provided servant manager that is asked to instantiate a servant for the request. If the servant manager instantiates such a server, that servant is added to the AOM and the request is passed to the new servant; otherwise, the operation raises **OBJECT_NOT_EXIST** in the client.
- If used with **NON_RETAIN**, the ORB also calls back to an application-provided servant manager and dispatches the request if the servant manager returns a servant. However, the association between CORBA object and the servant is effective for only a single request.

The Implicit Activation Policy

The implicit activation policy controls whether a servant can be activated implicitly or must be activated explicitly. (The default is **NO_IMPLICIT_ACTIVATION**).

```
enum ImplicitActivationPolicyValue {
    IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION
};

interface ImplicitActivationPolicy : CORBA::Policy {
    readonly attribute ImplicitActivationPolicyValue value;
};
```

- For **IMPLICIT_ACTIVATION** (which requires **RETAIN** and **SYSTEM_ID**), servants are added to AOM by calling **_this**.
- For **NO_IMPLICIT_ACTIVATION**, servants must be activated with a separate API call before you can obtain their object reference.



17
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.17 The Implicit Activation Policy

The code examples we have seen so far simply call **_this** on a newly instantiated servant in order to create a reference for the corresponding CORBA object. This technique works because the Root POA always uses the **IMPLICIT_ACTIVATION** policy. The first call to **_this** generates a new unique ID for the servant and adds the servant to the AOM. (The Root POA uses **SYSTEM_ID** and **RETAIN**).

However, as we will see in Section 12.22, **IMPLICIT_ACTIVATION** is useful only for transient objects. For persistent objects, you must use **NO_IMPLICIT_ACTIVATION** (because persistent objects almost always use **USER_ID**, for which **IMPLICIT_ACTIVATION** is illegal).

The Thread Policy

The thread policy controls whether requests are dispatched single-threaded (are serialized) or whether the ORB chooses a threading model for request dispatch. The default is **ORB_CTRL_MODEL**.

```
enum ThreadPolicyValue {
    ORB_CTRL_MODEL, SINGLE_THREAD_MODEL
};

interface ThreadPolicy : CORBA::Policy {
    readonly attribute ThreadPolicyValue value;
};
```

- **ORB_CTRL_MODEL** allows the ORB to choose a threading model. (Different ORBs will exhibit different behavior.)
- **SINGLE_THREAD_MODEL** serializes all requests on a per-POA basis.



18
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.18 The Thread Policy

The thread policy controls whether requests are dispatched on a single thread or multiple threads per POA. If you choose **SINGLE_THREAD_MODEL**, all invocations on that POA are serialized. If you choose **ORB_CTRL_MODEL**, the ORB is free to implement any threading strategy it prefers (including single-threaded dispatch).

Unfortunately, the specification is rather weak when it comes to controlling the threading model of a server, so ORBs from different vendors exhibit different behavior with respect to threading. For ORBacus, additional policies control a server's concurrency model with more precision. We cover these in Unit 25.

The Root POA Policies

The Root POA has a fixed set of policies:

Life Span Policy	TRANSIENT
ID Assignment Policy	SYSTEM_ID
ID Uniqueness Policy	UNIQUE_ID
Servant Retention Policy	RETAIN
Request Processing Policy	USE_ACTIVE_OBJECT_MAP_ONLY
Implicit Activation Policy	IMPLICIT_ACTIVATION
Thread Policy	ORB_CTRL_MODEL

Note that the implicit activation policy does *not* have the default value.

The Root POA is useful for transient objects only.

If you want to create persistent objects or use more sophisticated implementation techniques, you must create your own POAs.



19
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.19 The Root POA Policies

The Root POA always has the policies shown above. The policies for the Root POA have the default values, except for the implicit activation policy (which has a default value of `NO_IMPLICIT_ACTIVATION`).

The Root POA uses `TRANSIENT` and `SYSTEM_ID`, so it is useful only for creation of transient references. You should therefore restrict use of the Root POA to short-lived temporary objects.

Policy Creation

The POA interface provides a factory operation for each policy.

Each factory operation returns a policy with the requested value, for example:

```
module PortableServer {
  // ...
  interface POA {
    // ...
    LifespanPolicy create_lifespan_policy(
                        in LifespanPolicyValue value
                    );
    // ...
  };
};
```

You must call **destroy** on the returned object reference before you release it.



20
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.20 Policy Creation

The POA offers one factory operation for each of the seven policies:

```
module PortableServer {
  // ...
  interface POA {
    LifespanPolicy
      create_lifespan_policy(
          in LifespanPolicyValue value
      );

    IdAssignmentPolicy
      create_id_assignment_policy(
          in IdAssignmentPolicyValue value
      );

    IdUniquenessPolicy
      create_id_uniqueness_policy(
          in IdUniquenessPolicyValue value
      );

    ImplicitActivationPolicy
      create_implicit_activation_policy(
          in ImplicitActivationPolicyValue value
      );
  };
};
```

```

    RequestProcessingPolicy
        create_request_processing_policy(
            in RequestProcessingPolicyValue value
        );

    ServantRetentionPolicy
        create_servant_retention_policy(
            in ServantRetentionPolicyValue value
        );

    ThreadPolicy
        create_thread_policy(
            in ThreadPolicyValue value
        );
    // ...
};
};

```

To create a new POA, you first create the required policies, add them to a policy list, and then call the `create_POA` operation with the policy list. Here is an example that creates a POA with the `PERSISTENT`, `USER_ID`, and `SINGLE_THREAD_MODEL` policy values, leaving the remaining policies at their defaults:

```

PortableServer::POA_var root_poa = ...;    // Get Root POA

CORBA::PolicyList pl;
pl.length(3);

pl[0] = root_poa->create_lifespan_policy(
    PortableServer::PERSISTENT
);
pl[1] = root_poa->create_id_assignment_policy(
    PortableServer::USER_ID
);
pl[2] = root_poa->create_thread_policy(
    PortableServer::SINGLE_THREAD_MODEL
);

PortableServer::POA_var CCS_poa =
    root_poa->create_POA("CCS", nil_mgr, pl);

pl[0]->destroy();
pl[1]->destroy();
pl[2]->destroy();

```

Note that policies are copied when they are passed to `create_POA`, so destroying the policy objects after creating a POA does not affect the created POA. (Of course, if you need to create several POAs, you can keep the policy list around and reuse it for different calls to `create_POA`.)

NOTE: The above code is somewhat tedious if written in-line, so we suggest that you write a simple helper function that you can use to simplify your POA creation.

Creating Persistent Objects

Persistent objects have references that continue to work across server shut-down and re-start.

To create persistent references, you must:

- use **PERSISTENT**, **USER_ID**, and **NO_IMPLICIT_ACTIVATION**
- use the same POA name and object ID for each persistent object
- link the object IDs to the objects' identity and persistent state
- explicitly activate each servant
- allow the server to be found by clients by
 - either specifying a port number (direct binding)
 - or using the implementation repository (IMR)

It sounds complicated, but it is easy!



21
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.21 Creating Persistent Objects

The server we developed in Unit 10 created transient references because we used the Root POA for all its objects. Obviously, this is useless if the server is to retain any state across shut-down and re-start. Frequently, we need to develop servers that offer objects whose state is stored in a database or a network, such that the server acts as a CORBA front end to the persistent state. (This is a very common scenario when adding legacy applications to a CORBA system.)

To create persistent references, you must take care of the steps shown above, which we will examine over the next few slides.

Creating a Simple Persistent Server

- Use a separate POA for each interface.
- Create your POAs with the **PERSISTENT**, **USER_ID**, and **NO_IMPLICIT_ACTIVATION** policies.
- Override the `_default_POA` method on your servant class. (*Always* do this for all POAs other than the Root POA. If you have multiple ORBs, do this even for the Root POA on non-default ORBs.)
- Explicitly activate each servant with `activate_object_with_id`.
- Ensure that servants have unique IDs per POA. Use some part of each servant's state as the unique ID (the identity).
- Use the identity of each servant as its database key.



22
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.22 Creating a Simple Persistent Server

The above list presents a “cook book” approach to implementing a simple server for persistent objects. Note that, for now, we restrict ourselves to a simple implementation model where each CORBA object has a separate C++ servant that is permanently in memory.

12.22.1 Creating Persistent POAs

The first step is to create persistent POAs. A simple approach is to use a separate POA for each interface that is supported by the server.⁹ In the CCS system, we have three different interfaces and the controller object is a collection manager for thermometers and thermostats. We can reflect this relationship by creating three POAs, with the POA for the controller as the parent of the POAs for thermometers and thermostats:

```
PortableServer::POA_ptr
create_persistent_POA(
    const char *          name,
    PortableServer::POA_ptr parent)
{
    // Create policy list for simple persistence
    CORBA::PolicyList pl;
    pl.length(3);
    pl[0] = parent->create_lifespan_policy(
        PortableServer::PERSISTENT
```

9. For servers that keep all servants permanently in memory, a single persistent POA can be sufficient. However, once you use servant managers, having separate POAs simplifies servant instantiation and avoids object ID name clashes.

```

        );
    pl[1] = parent->create_id_assignment_policy(
        PortableServer::USER_ID
    );
    pl[2] = parent->create_thread_policy(
        PortableServer::SINGLE_THREAD_MODEL
    );

    // Get parent POA's POA manager
    PortableServer::POAManager_var pmanager
        = parent->the_POAManager();

    // Create new POA
    PortableServer::POA_var poa =
        parent->create_POA(name, pmanager, pl);

    // Clean up
    for (CORBA::ULong i = 0; i < pl.length(); ++i)
        pl[i]->destroy();

    return poa._retn();
}

int
main(int argc, char * argv[])
{
    // ...

    PortableServer POA_var root_poa = ...; // Get Root POA

    // Create POAs for controller, thermometers, and thermostats.
    // The controller POA becomes the parent of the thermometer
    // and thermostat POAs.
    PortableServer::POA_var ctrl_poa
        = create_persistent_POA("Controller", root_poa);
    PortableServer::POA_var thermo_poa
        = create_persistent_POA("Thermometers", ctrl_poa);
    PortableServer::POA_var tstat_poa
        = create_persistent_POA("Thermostats", ctrl_poa);

    // Create servants...

    // Activate POA manager
    PortableServer::POAManager_var mgr
        = root_poa->the_POAManager();
    mgr->activate();

    // ...
}

```

Creating a Simple Persistent Server (cont.)

`PortableServer::ServantBase` (which is the ancestor of all servants) provides a default implementation of the `_default_POA` function.

The default implementation of `_default_POA` always returns the Root POA.

If you use POAs other than the Root POA, you *must* override `_default_POA` in the servant class to return the correct POA for the servant.

If you forget to override `_default_POA`, calls to `_this` and several other functions will return incorrect object references and implicitly register the servant with the Root POA as a transient object.

Always override `_default_POA` for servants that do not use the Root POA! If you use multiple ORBs, override it for *all* servants!



23
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.22.2 Overriding `_default_POA`

If you use POAs other than the Root POA, you must override the `_default_POA` operation inherited from `ServantBase`. An easy way to do this is to use a private static class member in your servant class, together with an modifier and an accessor. For example:

```
class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    Thermometer_impl(/* ... */)
    {
        if (CORBA::is_nil(m_poa))
            throw "Thermometer_impl::m_poa not set!"
        // ...
    }

    static void
    poa(PortableServer::POA_ptr poa)
    {
        if (!CORBA::is_nil(m_poa))
            throw "Thermometer_impl::m_poa set more than once!"
        m_poa = PortableServer::POA::_duplicate(poa);
    }

    static PortableServer::POA_ptr
    poa()

```

```

    {
        if (CORBA::is_nil(m_poa))
            throw "Thermometer_impl::m_poa not set!"
        return m_poa;    // Note: no call to _duplicate() here!
    }

    virtual PortableServer::POA_ptr
    _default_POA()
    {
        return PortableServer::POA::_duplicate(m_poa);
    }
private:
    static PortableServer::POA_var m_poa;
    // ...
};

int
main(int argc, char * argv[])
{
    // ...
    PortableServer::POA_var thermo_poa
        = create_persistent_POA("Thermometers", ctrl_poa);
    Thermometer_impl::poa(thermo_poa);
    // ...
    PortableServer::POAManager_var mgr
        = root_poa->the_POAManager();
    mgr->activate();
    // ...
}

```

Note that this technique ensures that the POA for a servant class is set only once and that you cannot instantiate a servant before the POA has been set. The static `poa` accessor is useful if you want to get at the servant's POA before you have a servant instance, for example, when creating references with `create_reference_with_id` (see page 15-16). Note that this accessor does not call `_duplicate` on the returned reference, so you can make a call such as

```
Thermometer_impl::poa()->create_reference_with_id(...);
```

without having to release the returned reference.

Creating a Simple Persistent Server (cont.)

To explicitly activate a servant and create an entry for the servant in the AOM, call `activate_object_with_id`:

```
typedef sequence<octet> ObjectId;
// ...
interface POA {
    exception ObjectAlreadyActive {};
    exception ServantAlreadyActive {};
    exception WrongPolicy {};
    void activate_object_with_id(
        in ObjectId id, in Servant p_servant
    ) raises(
        ObjectAlreadyActive,
        ServantAlreadyActive,
        WrongPolicy
    );
    // ...
};
```



24
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.23 Explicit Servant Activation

For POAs with the `USER_ID` policy, we must explicitly add servants to the POA's AOM by calling `activate_object_with_id`. The object ID is passed to the call as a sequence of octets, which allows us to use IDs of any length and data type. However, because dealing with octet sequences directly and because, in practice, IDs are frequently string values, the C++ mapping provides four helper functions to make it easier to convert object IDs to strings and vice versa:

```
namespace PortableServer {
    // ...
    char *      ObjectId_to_string(const ObjectId &);
    CORBA::WChar * ObjectId_to_wstring(const ObjectId &);

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

Note that `ObjectId_to_string` and `ObjectId_to_wstring` will throw a `BAD_PARAM` exception if called with an object ID that is considered malformed by the ORB. (For example, for POAs with the `SYSTEM_ID` policy, object IDs usually must conform to an internal format.)

You can use any value as the object ID that uniquely (within its POA) identifies the target object. In addition, you must ensure that you use the same ID for the same logical CORBA object whenever you activate that object. For that reason, the best choice for an ID value is whatever bit of object state provides the object's identity. It might be a database row identifier, a social security

number, or, in case of the CCS, an asset number. One convenient place to do this is a servant's constructor:

```
class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    Thermometer_impl(CCS::AssetType anum /* , ... */);
    // ...
};

Thermometer_impl::
Thermometer_impl(CCS::AssetType anum /* , ... */)
{
    // ...
    ostrstream tmp;
    tmp << anum << ends;
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(tmp.str());
    tmp.rdbuf()->freeze(0);
    m_poa->activate_object_with_id(oid, this);
}
```

Merely instantiating the servant is then sufficient to ensure that it is activated correctly:

```
Thermometer_impl * t1 = new Thermometer_impl(22);
```

Of course, you can also activate the servant without doing this from a servant's constructor. (The advantage of this is that the POA policies and the way the servant is activated are unknown to the servant.)

NOTE: For multi-threaded servers, the POA dispatches requests as soon as you call `activate_object_with_id`. This means that, to avoid race conditions, you must call `activate_object_with_id` only once all other initialization for the servant is complete; otherwise, you end up with a race condition that can permit an incoming request to be dispatched before you have fully initialized the servant.

Creating a Simple Persistent Server (cont.)

A servant's object ID acts as a key that links an object reference to the persistent state for the object.

- For read access to the object, you can retrieve the state of the servant in the constructor, or use lazy retrieval (to spread out initialization cost).
- For write access, you can write the updated state back immediately, or when the servant is destroyed, or when the server shuts down.

When to retrieve and update persistent state is up to your application.

The persistence mechanism can be anything you like, such as a database, text file, mapped memory, etc.



25
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.23.1 Storing Persistent Servant State

Obviously, CORBA objects cannot be persistent unless you have some kind of database to hold their state. The object ID acts as the key that permits you to associate an object reference with a CORBA object and its servant, and the object ID acts as the database key to the object's state.

Exactly when and how to read and update the state for an object depends on your application. You can choose to hold all of a servant's state in memory (for example, as private data members of the servant) and to initialize all servants on construction, or you can use more sophisticated techniques, such as lazy initialization, depending on the space-time trade-offs for your application.

Similarly, for updates, you can choose to update the database immediately as soon as an update is made, or to cache updates until some timer expires, the servant is destroyed, or the server shuts down. The exact strategy as to how to update persistent servant state is largely determined by how much you are willing to sacrifice performance in order to reduce the risk of data loss in case of a crash.

As far as the ORB is concerned, the persistence mechanism you use is entirely up to you. The ORB merely enables you to correctly associate an incoming request with the persistent state of an object; the ORB does not provide persistence for the objects you create.

Creating a Simple Persistent Server (cont.)

POAs using the **PERSISTENT** policy write addressing information into each object reference.

You must ensure that the same server keeps using the same address and port; otherwise, references created by a previous server instance dangle:

- The host name written into the each IOR is obtained automatically.
- You can assign a specific port number using the **-OApport** option.

If you do not assign a port number, the server determines a port number dynamically (which is likely to change every time the server starts up).

If you do not have a working DNS, use **-OAnumeric**. This forces dotted-decimal addresses to be written into IORs.



26
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.23.2 Fixing a Port Number

Because persistent references contain addressing information, you must ensure that a server for a set of persistent objects starts on the same machine every time, and uses a fixed port number; otherwise, references created by a previous run of the same server are no longer valid.

The **-OApport** option allows you to set a port number when you start a server. By keeping the port number constant for each server execution, you can ensure that references continue to denote the same objects across server start-up and shut-down.¹⁰

Occasionally, you may find that incorrectly configured DNS servers prevent clients from resolving a domain name that is embedded in an IOR. For such cases, you can use the **-OAnumeric** option to override the domain name with a dotted-decimal IP address.

NOTE: Options beginning with **-OA** are ORBacus-specific and processed by `ORB_init`.

If you use **-OApport**, it only affects the Root POA manager, so **-OApport** works correctly if all persistent POAs use the that POA manager. There is no portable way to assign a port to other POA managers because POA managers do not have a separate identity (such as a name) that you could use to associate the port number with. If you need to control the port number for multiple POA managers, you can use the (ORBacus-specific) `ORB::POAManagerFactory` interface to create named POA managers. You can then use configuration properties to attach a different port number to each named POA manager. (See the ORBacus manual for details.)

¹⁰For large installations with many servers, manual administration of port numbers becomes a burden. For such installations, you can use the Implementation Repository (IMR), which permits you (among other things) to have port numbers assigned dynamically without breaking existing references. (See Unit 22 for details.)

Object Creation

Clients create new objects by invoking operations on a factory. The factory operation returns the reference to the new object. For example:

```
exception DuplicateAsset {};
```

```
interface ThermometerFactory {
    Thermometer create(in AssetType n) raises(DuplicateAsset);
};
```

```
interface ThermostatFactory {
    Thermostat create(in AssetType n, in TempType t)
        raises(DuplicateAsset, Thermostat::BadTemp);
};
```

As far as the ORB is concerned, object creation is just another operation invocation.



27
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.24 Object Creation

If we want to permit clients to create new objects (instead of offering a fixed set of objects in the server), we can add one or more object factories to the system. An object factory contains one or more operations that return an object reference. As far as the ORB is concerned, nothing special is actually happening—the client simply invokes an operation that returns an object reference. However, the implementation of a factory operation creates a new CORBA object as a side effect (possibly creating a new record in a database for the object).

There are many ways to create factory interfaces. For example, you can have a single factory that offers a separate creation operation for each type of object, you can add factory operations to a collection manager interface (such as the `Controller` interface), or you can add creation operations that create more than one object at a time.

It is important that your creation operations completely initialize the new object. Designs that create an object with one operation and then initialize it with another are poor (because you permit objects to be created that are not fully initialized, and therefore run the risk of a client using an uninitialized object).

Implementing a create operation is almost trivial. You must return an object reference, so you must call `_this` on an instantiated servant.¹¹ That servant can be persistent or transient, as appropriate for your situation. Initialization of persistent object state (if any) is up to the factory implementation. For example:

¹¹. See Section 15.2 for how to avoid the cost of instantiating a servant immediately.

```
CCS::Thermometer_ptr
ThermometerFactory_impl::
create(CCS::AssetType n)
throw(CORBA::SystemException, CCS::DuplicateAsset)
{
    // Create database record for the new servant (if needed)
    // ...

    // Instantiate a new servant on the heap
    Thermometer_impl * thermo_p = new Thermometer_impl(n);

    // Activate the servant if it is persistent (and
    // activation is not done by the constructor)
    // ...

    return thermo_p->_this();
}
```

The main thing to note here is that you *must* instantiate the new servant on the heap by calling `new`. If you use a stack-based servant, you will leave a dangling pointer in the POA's AOM, with an eventual crash the most likely outcome.¹²

NOTE: The preceding code example drops the return value from `new`. This need not cause a memory leak, depending on how you write your code to shut down the server. (Recall that, if the POA uses the `RETAIN` policy, it uses an AOM, so the servant pointer is not lost, but stored in the AOM after registration of the servant.)

Of course, you can also choose to store the servant pointer in a data structure and explicitly delete the servant again later.

¹²Note that there are several other reasons why servants should be instantiated on the heap. Rather than elaborate on these here, we suggest that you simply make it a habit to use heap-instantiated servants as a matter of principle.

Destroying CORBA Objects

To permit clients to destroy a CORBA object, add a **destroy** operation to the interface:

```
interface Thermometer {
    // ...
    void destroy();
};
```

The implementation of **destroy** deactivates the servant and permanently removes its persistent state (if any).

Further invocations on the destroyed object raise **OBJECT_NOT_EXIST**.

As far as the ORB is concerned, **destroy** is an ordinary operation with no special significance.



28
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.25 Destroying CORBA Objects

To permit clients to destroy a CORBA object, simply add a **destroy** operation to the object's interface, as shown above.

Note that you could also add a **destroy** operation to the object's factory:

```
interface ThermometerFactory {
    Thermometer create(in AssetType n);
    void destroy(in AssetType n); // Not recommended!
};
```

There are two fundamental problems with this IDL:

- Placing the **destroy** operation on the object's factory means that you must specify the asset number of the object to be destroyed. However, this creates an additional error scenario because it allows a client to supply the asset number of a non-existent thermometer. You can deal with this by throwing a user exception, but it makes the design more complex than necessary. (Note that you could not use **OBJECT_NOT_EXIST** to indicate an invalid asset number because then the client would conclude that the *factory* does not exist instead of the object to be destroyed, so you would probably use **BAD_PARAM**.)
- Placing **destroy** on the factory places an additional burden on clients because, for each object, they have to remember which factory was used to create the object. For large systems with large numbers of factories and objects, this approach can rapidly become very unwieldy.

Destroying CORBA Objects (cont.)

The POA holds a pointer to each servant in the AOM. You remove the AOM entry for a servant by calling **deactivate_object**:

```
interface POA {
    // ...
    void deactivate_object(in ObjectId oid)
        raises(ObjectNotActive, WrongPolicy);
};
```

Once deactivated, further requests for the same object raise **OBJECT_NOT_EXIST** (because no entry can be found in the AOM).

Once the association between the reference and the servant is removed, you can delete the servant.

deactivate_object does not remove the AOM entry immediately, but waits until all operations on the servant have completed.

Never call `delete this`; from inside `destroy`!



29
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.25.1 Object Deactivation

In order to inform the ORB that an object is destroyed, you must remove its entry from the Active Object Map by calling `deactivate_object`. Once the entry for the servant is removed, further requests raise **OBJECT_NOT_EXIST** in the client because no entry with the corresponding object ID can be found in the AOM.

However, `deactivate_object` does *not* remove the entry for the specified object ID immediately. Instead, it marks the entry as to be removed once all operations (including `destroy` itself) have finished. This means that you *cannot* implement `destroy` like this:

```
void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    ostream tmp;
    tmp << m_anum << ends;
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(tmp.str());
    tmp.rdbuf()->freeze(0);
    m_poa->deactivate_object(oid);           // Fine
    delete this;                           // Disaster!!!
}
```

Calling `delete this`; is wrong because the POA invokes operations on `ServantBase` after the operation completes, which results in accessing deleted memory.

Destroying CORBA Objects (cont.)

For multi-threaded servers, you must wait for all invocations to complete before you can physically destroy the servant.

To make this easier, the ORB provides a reference-counting mix-in class for servants:

```
class RefCountServantBase : public virtual ServantBase {
public:
    ~RefCountServantBase();
    virtual void    _add_ref();
    virtual void    _remove_ref();
protected:
    RefCountServantBase();
};
```

The ORB keeps a reference count for servants and calls `delete` on the servant once the reference count drops to zero.



30
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.25.2 Reference-Counted Servants

Because we cannot call `delete this;` from inside `destroy`, we end up with a problem: when and from where should we call `delete` on the servant after it is deactivated? (There is no convenient place where we could do this.) In addition, if a server is multi-threaded, we must worry about the fact that multiple operation invocations may be in the servant simultaneously and that we cannot delete the servant until after all these invocations have finished.

To get around this, the `PortableServer` namespace contains the `RefCountServantBase` reference-counting mix-in class. To use it, you simply inherit from it:

```
class Thermometer_impl :
    public virtual POA_CCS::Thermometer,
    public virtual PortableServer::RefCountServantBase {
// ...
};
```

Note that the constructor is protected, so you can only instantiate classes that are derived from `RefCountServantBase`. The constructor initializes a reference count for the servant to 1. The `_add_ref` and `_remove_ref` operations increment and decrement the reference count, respectively. In addition, `_remove_ref` calls `delete this;` once the reference count drops to zero.

By inheriting from `RefCountServantBase`, we can implement our `destroy` operation as follows, without having to artificially find a point of control at which it is safe to call `delete` (and without having to worry about still-executing operations in that servant):

```

CCS::Thermometer_ptr
ThermometerFactory_impl::
create(AssetType n) throw(CORBA::SystemException)
{
    CCS::Thermometer_impl * thermo_p = new Thermometer_impl(...);
    // ...

    m_poa->activate_object_with_id(oid, thermo_p);
    thermo_p->_remove_ref();    // Drop ref count
    return thermo_p->_this();
}

void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    ostringstream tmp;
    tmp << m_anum << ends;
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(tmp.str());
    tmp.rdbuf()->freeze(0);
    m_poa->deactivate_object(oid);    // Fine
}

```

The trick here is to realize that, in the factory `create` operation, we allocate the servant by calling `new`. This sets the reference count for the servant to 1. Calling `activate_object_with_id` increments the reference count a second time, so now it is 2. The call to `_remove_ref` decrements the reference count again, so now the only thing that keeps the servant alive in memory is its entry in the AOM. Every time the ORB invokes an operation, it calls `_add_ref` (which increments the reference count) and every time an operation completes, the ORB calls `_remove_ref` (which drops the reference count again). This means that, for as long as the CORBA object exists, the reference count on the servant will be at least 1, and greater than 1 during operation invocations.

When `destroy` is called by a client, the reference count for the servant is 2 on entry to the operation (assuming no other threads are executing inside the servant). `destroy` breaks the reference-to-servant association by calling `deactivate_object`. (Remember that this does not remove the servant's entry from AOM immediately, but only when all requests for the servant have completed.) `deactivate_object` calls `_remove_ref`, which drops the reference count to 1. Once `destroy` returns control to the ORB, the ORB calls `_remove_ref` to balance the call to `_add_ref` it made when it invoked `destroy`. This drops the reference count to 0, removes the servant's entry from the AOM, and calls `delete` on the servant.

NOTE: We strongly encourage you to use `RefCountServantBase` for your servants if you support life cycle operations. This is of paramount importance for threaded servers, where you cannot otherwise be sure when it is safe to delete a servant. Reference-counted servants require you to use heap allocation, but you should be using heap-allocated servants anyway. The main reason for preferring heap allocation is that it is also required for more advanced implementation techniques that instantiate servants on demand. (See Section 15.2.)

Destroying CORBA Objects (cont.)

Destroying a persistent object implies destroying its persistent state.

Generally, you cannot remove persistent state as part of **destroy** (because other operations executing in parallel may still need it):

- It is best to destroy the persistent state from the servant's destructor.
- The servant destructor also runs when the server is shut down, so take care to destroy the persistent state only after a previous call to **destroy**.
- Use a boolean member variable to remember a previous call to **destroy**.



31
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.25.3 Destroying the Persistent State of an Object

For a single-threaded server, it is OK to destroy servant state immediately from within **destroy**. However, this technique does not generalize to multi-threaded servants because other requests may still be executing inside the servant in parallel and require the persistent state to remain. This means that it is best to destroy persistent servant state from the servant's destructor, when you can be sure that no other requests are still active in the servant.

However, take care not to destroy persistent state unconditionally. If you do, you will end up destroying the persistent state for every object when the server shuts down. This, of course, would be wrong because, for persistent objects, server shut-down does not imply that the server's CORBA objects are destroyed, only that their implementation is temporarily unavailable. An easy way to deal with this is to add a boolean member variable to each servant:

```
class Thermometer_impl :
    public virtual POA_CCS::Thermometer,
    public virtual PortableServer::RefCountServantBase {
public:
    Thermometer_impl(AssetType anum) :
        m_anum(anum), m_removed(false) { /* ... */ }
    ~Thermometer_impl();
    // ...
protected:
    AssetType m_anum;
    bool m_removed;
};
```


In the `destroy` member function, set the `m_removed` member to true and then check it in the destructor:

```
Thermometer_impl::
~Thermometer_impl()
{
    if (m_removed) {
        // Destroy persistent state for this object...
    }
    // Release whatever other resources (not related to
    // persistent state) were used...
}
```

NOTE: `deactivate_object` does not immediately remove the AOM entry for the servant. Instead, the entry is marked for deletion and removed once the servant becomes idle. Until this happens, new request that arrive for the same servant are transparently delayed until the servant is deactivated. Therefore, you do not have to worry about concurrent activation and deactivation of the same servant.

Deactivation and Servant Destruction

The POA offers a **destroy** operation:

```
interface POA {
    // ...
    void destroy(
        in boolean etherealize_objects,
        in boolean wait_for_completion
    );
};
```

Destroying a POA recursively destroys its descendant POAs.

If **wait_for_completion** is true, the call returns after all current requests have completed and all POAs are destroyed. Otherwise, the POA is destroyed immediately.

Calling **ORB::shutdown** or **ORB::destroy** implicitly calls **POA::destroy** on the Root POA.



32
The Portable Object Adapter (POA)
Copyright 2000–2001 IONA Technologies



12.26 Deactivation and Servant Destruction

You can explicitly destroy a POA by calling its **destroy** operation. Doing this recursively destroys any descendent POAs before destroying the parent POA. In other words, **destroy** does a depth-first traversal of the POA hierarchy; the order in which siblings are destroyed is undefined.

The **wait_for_completion** flag determines whether the invocation waits until current requests have finished executing and destruction is complete. We strongly recommend that you do not use a **wait_for_completion** flag of false unless you are prepared to have currently executing operations fail in unpredictable ways (because all their POA-related calls will raise **OBJECT_NOT_EXIST** if you do this).

The **etherealize_objects** parameter is relevant to servant activators. (See Section 15.3.) It determines whether servants will be etherealized as part of the call or not.

If you call **ORB::shutdown** or **ORB::destroy**, the ORB makes an implicit call to **POA::destroy** on the Root POA. (**ORB::shutdown** passes the value of its **wait_for_completion** flag through to **POA::destroy**.)

Note that calling **destroy** on a POA implicitly invokes **_remove_ref** on every servant in the AOM. This means that each servant's destructor runs once operations have drained out of the servant.

13. Exercise: Writing a Persistent Server

Summary

In this unit, you will add persistence to a server and implement life cycle operations.

Objectives

By the completion of this unit, you will know how to create persistent object references, how to maintain state in a database, and how to implement life cycle operations.

13.1 Source Files and Build Environment

You will find this exercise in your `persistent` directory. The files in this directory are the same as for Unit 10.

13.2 Server Operation

This server differs markedly from the one we presented in Unit 11. The following are the major differences:

- The ICP simulator is persistent for this exercise, so state changes from run to run are remembered.
- The server maintains a list of asset numbers on disk.
- The server uses reference-counted servants that inherit from `RefCountServantBase`.
- The server uses three separate persistent POAs, one for each type of object.
- The controller supports two factory operations to create thermometers and thermostats, and thermometers offer a `destroy` operation.

13.2.1 The Persistent ICP Simulator

The persistent version of the ICP simulator works as follows. The ICP library creates a static object that, in its constructor, uploads the database into memory and, in its destructor, writes the contents of the database to disk. The database file is called `CCS_DB` and is created in the current directory. A consequence of this design is that the database contents will not be written to disk if the server terminates abnormally (because then the destructor of the global object will not run). Clearly, this is not a realistic approach for a real application but will be good enough for the purposes of this exercise.

The format of the `CCS_DB` file is documented in Henning & Vinoski, pages 1026–1027. Please have a look at the description there for the details.

When a device is created by the ICP simulator, it uses the asset number to decide what type the device should have. If the asset number is odd, the device is a thermometer, and if the asset number is even, the device is a thermostat. (This is not an in-built limitation of our system, but simply a reflection of the fact that we do not have real hardware devices that we could interrogate for their model string, so we must infer the model some other way. With real hardware, the model string would be read out of ROM.) Keep this limitation in mind during testing of your server because attempts to create a device with the wrong asset number for its type will fail.

13.2.2 Persistent Asset Numbers

The ICP simulator does not offer a way to discover which devices are on the network. This presents a problem for the controller, which must know which devices exist. To get around this, the constructor of the controller reads a list of asset number from the file `CCS_assets`, and the destructor writes the current asset list into the file when the server shuts down. Note that the notion of what devices exist must match in the `CCS_DB` and `CCS_assets` file—if they go out of sync, the server will get confused. You can easily ensure that the two files match by hand-editing them (or simply clearing them).

13.2.3 Reference-Counted Servants

Servants for this server are allocated on the heap and inherit from `RefCountServantBase`. In addition, the server uses persistent POAs with the `RETAIN` and `USER_ID` policies, so servants must

be explicitly entered into the AOM with `activate_object_with_id`. Note that `activate_object_with_id` calls `_add_ref` on the servant. This means that you can call `_remove_ref` as soon as you have added a new servant to the AOM. (You can look at the constructor of the controller servant to see how this works.)

13.2.4 Separate POAs

The server uses a separate POA for each type of device. (The POAs all have the same policies.) Note that servants override the `_default_POA` method to ensure that object references are created with the correct POA identifier. Each servant class has a private `m_poa` member that is used to hold the POA for all servants of the same type.

13.2.5 Life Cycle Operations

The Controller interface contains two new operations, `create_thermometer` and `create_thermostat`. In addition, the Thermometer interface contains a `destroy` operation. You will implement these operations as part of this exercise.

13.3 What You Need to Do

Step 1

Have a look at the `CCS.idl` file to see how the life cycle operations affect the interfaces.

Step 2

Study the contents of the `server.h` file. Make sure that you understand how the `_default_POA` function is implemented. Have a look at the overloaded static `poa` member functions. These functions are useful when you need to get at the POA for an interface without actually having a servant instance. Note that you must *not* release the return value from the `poa` accessor.

Also note that the controller contains an `exists` helper function to determine whether a particular device exists already. This helper function is useful for the implementation of the factory operations.

Step 3

Look at the `server main` function. Note that an extra `catch` clause for `const char *` has been added. This is useful if you encounter a fatal error condition. You can simply throw a string constant in order to cleanly terminate the program with a message. For example:

```
throw "Fatal error: Cannot open DB file";
```

Step 4

Look at the body of the `run` function. It creates three POAs using the `create_persistent_POA` helper function and sets the static `m_poa` member in each servant class. The body of `create_persistent_POA` is empty. Add the required code.

Step 5

Implement the body of `create_thermometer`. You can look at the implementation of the constructor for the controller to get a general idea of the actions you need to take. Note that the `make_oid` helper function is provided so you can easily convert an asset number to an octet sequence. Remember that you must not only update the ICP network, but also update the

controllers notion of what devices exist by calling `add_impl`. Modify the provided client source code to test your implementation.

Step 6

Implement the body of the `destroy` operation and test your implementation.

Step 7

Implement the body of the `create_thermostat` operation.

Step 8

Test your implementation. You should be able to reuse the controller IOR produced by a previous run of the server to reach a newly-created server process.

14. Solution: Writing a Persistent Server

14.1 Solution

Step 4

```
// Create a new POA named 'name' and with 'parent' as its
// ancestor. The new POA shares its POA manager with
// its parent.

static PortableServer::POA_ptr
create_persistent_POA(
    const char *          name,
    PortableServer::POA_ptr parent)
{
    // Create policy list for simple persistence
    CORBA::PolicyList pl;
    CORBA::ULong len = pl.length();
    pl.length(len + 1);
    pl[len++] = parent->create_lifespan_policy(
        PortableServer::PERSISTENT
    );
    pl.length(len + 1);
    pl[len++] = parent->create_id_assignment_policy(
        PortableServer::USER_ID
    );
    pl.length(len + 1);
    pl[len++] = parent->create_thread_policy(
        PortableServer::SINGLE_THREAD_MODEL
    );
    pl.length(len + 1);
    pl[len++] = parent->create_implicit_activation_policy(
        PortableServer::NO_IMPLICIT_ACTIVATION
    );

    // Get parent POA's POA manager
    PortableServer::POAManager_var pmanager
    = parent->the_POAManager();

    // Create new POA
    PortableServer::POA_var poa =
        parent->create_POA(name, pmanager, pl);

    // Clean up
    for (CORBA::ULong i = 0; i < len; ++i)
        pl[i]->destroy();

    return poa._retn();
}
```


Step 5

```

CCS::Thermometer_ptr
Controller_impl::
create_thermometer(CCS::AssetType anum, const char * loc)
throw(CORBA::SystemException, CCS::Controller::DuplicateAsset)
{
    if (exists(anum))
        throw CCS::Controller::DuplicateAsset();
    if (anum % 2 == 0)
        throw CORBA::BAD_PARAM(); // ICS limitation
    if (ICP_online(anum) != 0)
        abort();
    if (ICP_set(anum, "location", loc) != 0)
        abort();
    Thermometer_impl * t = new Thermometer_impl(anum);
    PortableServer::ObjectId_var oid = make_oid(anum);
    Thermometer_impl::poa()->activate_object_with_id(oid, t);
    t->_remove_ref();

    return t->_this();
}

```

Step 6

```

// IDL destroy operation.

void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    m_ctrl->remove_impl(m_anum);
    if (ICP_offline(m_anum) != 0)
        abort();
    PortableServer::ObjectId_var oid = make_oid(m_anum);
    PortableServer::POA_var poa = _default_POA();
    poa->deactivate_object(oid);
}

```

Step 7

```

CCS::Thermostat_ptr
Controller_impl::
create_thermostat(
    CCS::AssetType  anum,
    const char*    loc,
    CCS::TempType  temp)
throw(
    CORBA::SystemException,
    CCS::Controller::DuplicateAsset,
    CCS::Thermostat::BadTemp)

```

```

{
    if (exists(anum))
        throw CCS::Controller::DuplicateAsset();
    if (anum % 2)
        throw CORBA::BAD_PARAM(); // ICS limitation
    if (ICP_online(anum) != 0)
        abort();
    if (ICP_set(anum, "location", loc) != 0)
        abort();
    // Set the nominal temperature.
    if (ICP_set(anum, "nominal_temp", &temp) != 0) {

        // If ICP_set() failed, read this thermostat's
        // minimum and maximum so we can initialize the
        // BadTemp exception.
        CCS::Thermostat::BtData btd;
        ICP_get(
            anum, "MIN_TEMP",
            &btd.min_permitted, sizeof(btd.min_permitted)
        );
        ICP_get(
            anum, "MAX_TEMP",
            &btd.max_permitted, sizeof(btd.max_permitted)
        );
        btd.requested = temp;
        btd.error_msg = CORBA::string_dup(
            temp > btd.max_permitted ? "Too hot" : "Too cold"
        );
        ICP_offline(anum);
        throw CCS::Thermostat::BadTemp(btd);
    }

    Thermostat_impl * t = new Thermostat_impl(anum);
    PortableServer::ObjectId_var oid = make_oid(anum);
    Thermostat_impl::poa()->activate_object_with_id(oid, t);
    t->_remove_ref();

    return t->_this();
}

```

Step 8

```
./server -OApport 7819 >ctrl.ref
```

15. Advanced Uses of the POA

Summary

This unit covers advanced aspects of the POA that permit you to exercise tight control over the trade-offs in performance, scalability, and memory consumption of a server.

Objectives

By the completion of this unit, you will be able to create sophisticated servers that scale to unlimited numbers of objects. In addition, you will have an appreciation of advanced caching techniques, such as eviction of servants and optimistic caching.

Pre-Loading of Objects

The `USE_ACTIVE_OBJECT_MAP_ONLY` requires one servant per CORBA object, and requires all servants to be in memory at all times.

This forces the server to pre-instantiate all servants prior to entering its dispatch loop:

```
int main(int argc, char * argv[])
{
    // Initialize ORB, create POAs, etc...
    // Instantiate one servant per CORBA object:
    while (database_records_remain) {
        // Fetch record for a servant...
        // Instantiate and activate servant...
    }
    // ...
    orb->run();    // Start dispatch loop
    // ...
}
```



1
Advanced Uses of the POA
Copyright 2000–2001 IONA Technologies



15.1 Pre-Loading of Objects

So far, all the server code we have seen used the `USE_ACTIVE_OBJECT_MAP_ONLY` policy for its POAs. As a result, we end up with a design that requires a separate servant for each CORBA object and forces us to keep all servants in memory at all times. This is a perfectly acceptable and sensible implementation choice, provide that you can afford it. This will be the case if:

- The number of objects is small enough to fit into memory without increasing the working set unacceptably. Generally, this means that you can either have a fairly small number of large objects or a fairly large number of small objects.
- The time taken to iterate over the database and to instantiate all the servants is acceptable. Generally, this means that initialization of each servant has to be fast, otherwise the server may take minutes or hours to enter its dispatch loop, which is usually unacceptable (especially if servers are started automatically as requests arrive—see Unit 22).

To get around this problem, the POA offers servant managers.

Servant Managers

Servant managers permit you to load servants into memory on demand, when they are needed.

Servant managers come in two flavors:

- **ServantActivator** (requires the **RETAIN** policy)

The ORB makes a callback the first time a requests arrives for an object that is not in the AOM. The callback returns the servant to the ORB, and the ORB adds it to the AOM.

- **ServantLocator** (requires the **NON_RETAIN** policy)

The ORB makes a callback every time a request arrives. The callback returns a servant for the request. Another callback is made once the request is complete. The association between request and servant is in effect only for the duration of single request.



15.2 Servant Managers

Servant managers allows you to bring a servant into memory when a request for it arrives, instead of having to keep all servants in memory at all times, just in case they are needed. Servant managers come in two flavors, both derived from a common base interface:

```
module PortableServer {
    // ...

    interface ServantManager {};

    interface ServantActivator : ServantManager {
        // ...
    };

    interface ServantLocator : ServantManager {
        // ...
    };
};
```

Servant activators require the **RETAIN** policy to bring servants into memory on demand (and leave them in the AOM thereafter). Servant locators require the **NON_RETAIN** policy and only provide the association between a request and its servant for the duration of a single request.

Servant locators require more implementation work from the application than servant activators, but are more powerful and offer more aggressive scalability options.

Both servant activators and servant locators also require the **USE_SERVANT_MANAGER** policy.

Servant Activators

```

exception ForwardRequest {
    Object forward_reference;
};
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
    );
};

```



3
Advanced Uses of the POA
Copyright 2000–2001 IONA Technologies



15.3 Servant Activators

Servant activators have the IDL interface shown above. As the server programmer, you create an implementation of this interface and register it with a POA. (See page 15-9.) The corresponding POA must use the RETAIN and USE_SERVANT_MANAGER policies.

Once an activator is registered, if a request comes in for which the ORB cannot locate an entry in the AOM, instead of raising OBJECT_NOT_EXIST in the client, it first calls the `incarnate` operation, passing the object ID and the POA for the request. Now `incarnate` is in control and can use the information passed to it to locate the state of a servant for the request and instantiate the servant:

- If `incarnate` can instantiate a servant, it returns the servant to the ORB, which then adds the object ID and servant to its AOM and dispatches the request as usual. (This is, of course, completely transparent to the client.)
- If `incarnate` cannot locate the state for a servant with the given object ID (probably because the corresponding object was destroyed previously), it simply raises OBJECT_NOT_EXIST, which is propagated back to the client.

The Request exception presents a third option to return control to the ORB. If `incarnate` raises this exception, it returns an object reference. This indicates to the ORB that the request could not be handled by this POA, but that retrying the request at the returned IOR may work. The ORB returns the IOR to the client, and the client-side ORB (transparently to the application) then tries to dispatch the request using the new IOR. This mechanism can be used to support object migration (albeit at the cost of slower request dispatch).

The `etherealize` operation is invoked by the ORB to instruct you that it no longer requires the servant and gives you a chance to reclaim the servant.

The ORB invokes `etherealize` in the following circumstances:

- `deactivate_object` was called for an object represented by the servant
- `POAManager::deactivate` was called on a POA manager with active servants
- `POA::destroy` was called with `etherealize_objects` set to true
- `ORB::shutdown` or `ORB::destroy` were called

The `cleanup_in_progress` parameter is true if the call to `etherealize` resulted because of a call to `POAManager::deactivate`, `POA::destroy`, `ORB::shutdown`, or `ORB::destroy`; otherwise, the parameter is false. This allows you to distinguish between normal deactivation and application (or POA) shut-down.

The `remaining_activations` parameter is false if this servant is used to only represent a single CORBA object; otherwise, if true, the parameter indicates that this servant still represents other CORBA objects and therefore should not be physically deleted. (See page 15-18 for how to map multiple CORBA objects onto a single C++ servant.)

NOTE: The ORB removes the entry for the passed object ID from the AOM *before* it calls `etherealize`.

Implementing a Servant Activator

The implementation of **incarnate** is usually very similar to a factory operation:

1. Use the object ID to locate the persistent state for the servant.
2. If the object ID does not exist, throw **OBJECT_NOT_EXIST**.
3. Instantiate a servant using the retrieved persistent state.
4. Return a pointer to the servant.

The implementation of **etherealize** gets rid of the servant:

1. Write the persistent state of the servant to the DB (unless you are using write-through).
2. If **remaining_activations** is false, call `_remove_ref` (or call `delete`, if the servant is not reference-counted).



4
Advanced Uses of the POA
Copyright 2000–2001 IONA Technologies



15.4 Implementing a Servant Activator

To implement a servant activator, you must derive your implementation from `PortableServer::ServantActivator`. This means your servant activator is itself a servant for the `ServantActivator` interface:

```
class Activator_impl :
    public virtual POA_PortableServer::ServantActivator {
public:
    virtual PortableServer::Servant
    incarnate(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr          poa
    ) throw(CORBA::SystemException,
           PortableServer::ForwardRequest);

    virtual void
    etherealize(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr          poa,
        PortableServer::Servant          serv,
        CORBA::Boolean                   cleanup_in_progress,
        CORBA::Boolean                   remaining_activations
    ) throw(CORBA::SystemException);
};
```


The implementation of `incarnate` can be outlined as follows:

```
PortableServer::Servant
Activator_impl::
incarnate(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr          poa
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    // Turn the OID into a string
    CORBA::String_var oid_str;
    try {
        oid_string = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM &) {
        throw CORBA::OBJECT_NOT_EXIST(); // Malformed OID
    }

    // Use OID to look in the DB for the persistent state...
    if (object_not_found)
        throw CORBA::OBJECT_NOT_EXIST();

    // Use the state retrieved from the database to
    // instantiate a servant. The type of the servant may be
    // implicit in the POA, the object ID, or the database state.
    AssetType anum = ...;
    return new Thermometer_impl(anum, /* ... */);
}
```

The implementation of `etherealize` is usually very simple. If your servants are implemented to write updates directly to the database with each update operation, there is no persistent state to be finalized. Otherwise, if updates are held in memory and written to the database only when the servant is destroyed, `etherealize` must write the database state before destroying the servant:

```
void
Activator_impl::
etherealize(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr          poa,
    PortableServer::Servant          serv,
    CORBA::Boolean                   cleanup_in_progress,
    CORBA::Boolean                   remaining_activations
) throw(CORBA::SystemException)
{
    // Write updates (if any) for this object to database and
    // clean up any resources that may still be used by the
    // servant (or do this from the servant destructor)...
    if (!remaining_activations)
        serv->_remove_ref(); // Or delete serv, if not ref-counted
}
```

Use Cases for Servant Activators

Use servant activators if:

- you cannot afford to instantiate all servants up-front because it takes too long

A servant activator distributes the cost of initialization over many calls, so the server can start up quicker.

- clients tend to be interested in only a small number of servants over the period the server is up

If all objects provided by the server are eventually touched by clients, all servants end up in memory, so there is no saving in that case.

Servant activators are of interest mainly for servers that are started on demand.



5
Advanced Uses of the POA
Copyright 2000–2001 IONA Technologies



15.5 Use Cases for Servant Activators

Servant activators are useful mainly because they incur the cost of servant initialization only for those servants that are actually used. In addition, the initialization cost is distributed over many calls instead of incurred up-front, so the server starts up more quickly. This is important if you arrange for servers to be started by an implementation repository when a client sends a request (see Unit 22), which works well only if the server comes up quickly.

If only a subset of the server's objects are touched by clients, servant activators offer some saving in memory consumption because only those servants that are actually used need be instantiated. However, if a server runs for a long time and clients, over time, end up using every object offered by the server, all servants eventually end up in memory, so there is no saving.

NOTE: You can call `deactivate_object` to get rid of unused servants in order to limit the memory consumption of the server. This idea generalizes to that of a servant cache and is described in more detail in Henning & Vinoski as the Evictor Pattern. However, such servant caches are best implemented with servant locators instead of servant activators.

Servant Manager Registration

You must register a servant manager with the POA before activating the POA's POA manager:

```
interface POA {
    // ...
    void          set_servant_manager(in ServantManager mgr)
                  raises(WrongPolicy);
    ServantManager get_servant_manager() raises(WrongPolicy);
};
```

If you pass a servant activator, to **set_servant_manager**, the POA must use **USE_SERVANT_MANAGER** and **RETAIN**.

You can register the same servant manager with more than one POA.

You can set the servant manager only once; it remains attached to the POA until the POA is destroyed.

get_servant_manager returns the servant manager for a POA.



15.6 Servant Manager Registration

You must register a servant manager with a POA before you activate the POA's manager. This is necessary because, otherwise, clients will get OBJ_ADAPTER exceptions for their requests. (If you use the same servant manager for multiple POAs and later create another POA that will use the same servant manager, you must temporarily put the POA manager into the holding state and activate it again once you have created the new POA and set its servant manager.)

In C++, instantiation and registration of a servant manager is trivial:

```
Activator_impl * ap = new Activator_impl;
PortableServer::ServantManager_var mgr_ref = ap->_this();
some_poa->set_servant_manager(mgr_ref);
```

You cannot change the servant manager of a POA once you have assigned it. Calling **set_servant_manager** a second time raises OBJ_ADAPTER.

Type Issues with Servant Managers

How does a servant manager know which type of interface is needed?
Some options:

- Use a separate POA and separate servant manager class for each interface. The interface is implicit in the servant manager that is called.
- Use a separate POA for each interface but share the servant manager. Infer the interface from the POA name by reading the **the_name** attribute on the POA.
- Use a single POA and servant manager and add a type marker to the object ID. Use that type marker to infer which interface is needed.
- Store a type marker in the database with the persistent state.

The second option is usually easiest.



15.7 Type Issues with Servant Managers

When a POA calls `incarnate`, it passes the POA and object ID for the servant to be instantiated. `incarnate` is not told by the POA which type of interface (for example, a thermometer or a thermostat) is required. In fact, the POA itself does not know what type of interface a request is for until after it gets a servant for the request from the application. (The type of interface that eventually handles a request is not known at compile time and not transmitted over the wire because of late binding.) This means that a servant manager must know what type of interface to instantiate purely from the arguments it receives. The above slide presents the options for how you can deal with multiple interface types and servant managers.

The easiest thing is to simply use a separate POA and separate servant manager *class* for each interface type. That way, the decision which interface is needed is hard-coded into each servant manager class and made at compile time.

Another option is to share a servant manager among several POAs, one POA for each interface type. In that case, the servant manager can use the POA name (available by reading the `the_name` attribute on the POA). This approach allows you to use a single servant manager class for each interface type and decides what type to instantiate at run time. This approach is useful if instantiation of different servant types is substantially similar in nature (such as for thermometers and thermostats, where a lot of the code is identical).

You can use a type marker that becomes part of the object ID of each servant. For example, you could use a “t” prefix on the object ID for thermostats, and an “m” for thermometers. This approach also works well, but is maintenance intensive if the interface types supported by a server grow over time (and it adds a small size penalty to object references). Alternatively, as a variation on this idea, you can keep the type marker in the database with the persistent state of each object.

Servant Locators

```

native Cookie;
interface ServantLocator : ServantManager {
    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           serv
    );
};

```



15.8 Servant Locators

Servant locators provide a more powerful alternative to servant activators. They require the `NON_RETAIN` policy and `USE_SERVANT_MANAGER` policies on the POA. One consequence of `NON_RETAIN` is that the POA no longer maintains an Active Object Map. Instead, the association between a request and its servant remains in effect only for the duration of the request and is forgotten by the POA as soon as a request completes.

If you use servant locators, *every* incoming request causes the POA to first call `preinvoke`. Like `incarnate`, `preinvoke` can either throw an `OBJECT_NOT_EXIST` exception or return a servant. If `preinvoke` returns a servant, the request is given to the servant. As soon as the servant has carried out the request, the POA calls `postinvoke`, which permits you to clean up. Once `postinvoke` has finished, the POA forgets the association between the request and the servant. Another request for the same object will call `preinvoke` again (and may be carried out by the same servant or a different one).

Note that, in contrast to servant activators, servant locators are told which operation (or attribute) is being invoked via the `operation` parameter (`CORBA::Identifier` is an unbounded string). This permits you to select a servant based on the operation name as well as the POA and object ID.

`preinvoke` can return a value to the ORB in the `the_cookie` parameter. The ORB treats the cookie as an opaque value and never looks at it. However, it guarantees that the cookie that was returned by `preinvoke` will be passed to `postinvoke`. This allows you pass information from `preinvoke` to `postinvoke`, for example, by using the cookie as a key into a data structure.

Implementing Servant Locators

The implementation of **preinvoke** is usually very similar to a factory operation (or **incarnate**):

1. Use the POA, object ID, and operation name to locate the persistent state for the servant.
2. If the object does not exist, throw **OBJECT_NOT_EXIST**.
3. Instantiate a servant using the retrieved persistent state.
4. Return a pointer to the servant.

The implementation of **postinvoke** gets rid of the servant:

1. Write the persistent state of the servant to the DB (unless you are using write-through).
2. Call **_remove_ref** (or call **delete**, if the servant is not reference-counted).



15.9 Implementing Servant Locators

To implement a servant locator, you must derive your implementation from `PortableServer::ServantLocator`:

```
// In the PortableServer namespace:
// typedef void * Cookie;

class Locator_impl :
    public virtual POA_PortableServer::ServantLocator,
    public virtual PortableServer::RefCountServantBase {
public:
    virtual PortableServer::Servant
    preinvoke(
        const PortableServer::ObjectId &          oid,
        PortableServer::POA_ptr                  adapter,
        const char *                              operation,
        PortableServer::ServantLocator::Cookie & the_cookie
    ) throw(CORBA::SystemException,
            PortableServer::ForwardRequest);

    virtual void
    postinvoke(
        const PortableServer::ObjectId &          oid,
        PortableServer::POA_ptr                  adapter,
```

```

        const char *                operation,
        PortableServer::ServantLocator::Cookie the_cookie,
        PortableServer::Servant      the_servant
    ) throw(CORBA::SystemException);
};

```

A simple implementation of `preinvoke` can be identical to an implementation of `incarnate`. Note that the IDL `Cookie` type is mapped to a `void *` in C++, so you can move arbitrary information between `preinvoke` and `postinvoke`.

```

PortableServer::Servant
Locator_impl::
preinvoke(
    const PortableServer::ObjectId &      oid,
    PortableServer::POA_ptr              adapter,
    const char *                          operation,
    PortableServer::ServantLocator::Cookie & the_cookie
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    // Turn the OID into a string
    CORBA::String_var oid_str;
    try {
        oid_string = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM &) {
        throw CORBA::OBJECT_NOT_EXIST(); // Malformed OID
    }

    // Use OID to look in the DB for the persistent state...
    if (object_not_found)
        throw CORBA::OBJECT_NOT_EXIST();

    // Use the state retrieved from the database to
    // instantiate a servant. The type of the servant may be
    // implicit in the POA, the object ID, or the database state.
    AssetType anum = ...;
    return new Thermometer_impl(anum, /* ... */);
}

```

Note that this is not the most efficient implementation of `preinvoke`, mainly because it creates and destroys a servant for each request. However, with a bit of thought, we can come up with designs in which servants are not destroyed by `postinvoke` but are placed into a pool instead, to be reused if another request comes in for the same object. (See Henning & Vinoski for details.)

NOTE: The implementation of `postinvoke` is very similar to an implementation of `etherealize`, so we do not show it here.

Use Cases for Servant Locators

Advantages of servant locators:

- They provide precise control over the memory use of the server, regardless of the number of objects supported.
- **preinvoke** and **postinvoke** bracket every operation call, so you can do work in these operations that must be performed for every operation, for example:
 - initialization and cleanup
 - creation and destruction of network connections or similar
 - acquisition and release of mutexes
- You can implement servant caches that bound the number of servants in memory to the n most recently used ones (Evictor Pattern.)



10
Advanced Uses of the POA
Copyright 2000–2001 IONA Technologies



15.10 Use Cases for Servant Locators

The main use case for servant locators is that they limit memory consumption in the server to the number of objects actually in use at any one time. This means that you can build servers that scale to very large numbers of objects whose state is retrieved from persistent storage on demand. (Naturally, this scalability does not come for free because each invocation does more work, so throughput will be somewhat less.)

Another advantage of servant locators is that you can use the `preinvoke` and `postinvoke` calls to perform work that is common to all operations. For example, if you support life cycle operations and use an `m_removed` member as shown on page 12-46, you test this member in `preinvoke` and throw an `OBJECT_NOT_EXIST` exception if the object no longer exists. (Note that servant activators and servant locators can only throw system exceptions, not user exceptions. Also note that, if you want to access servant members during `preinvoke` or `postinvoke`, you must perform a down-cast from `ServantBase *` to a pointer to the actual type of your servant.)

`preinvoke` and `postinvoke` can also be useful to ensure exclusive access to a critical region by locking and unlocking a mutex before and after every operation (or only some operations—you can make a choice depending on the operation name). This works because the POA guarantees that, in multi-threaded servers, `preinvoke`, the operation body, and `postinvoke` all run in the same thread.

Finally, servant locators are useful to implement the Evictor Pattern, which creates a bounded pool of the most recently used servants. This technique is extremely effective because it exploits the fact that clients usually are interested in a small number of objects for some time before they shift interest to a different group of objects; this locality of reference means that most requests can be satisfied by an already instantiated servant. (See Henning and Vinoski for details.)

Servant Managers and Collections

For operations such as `Controller::list`, you cannot iterate over a list of servants to return references to all objects because not all servants may be in memory.

Instead of instantiating a servant for each object just so you can call `_this` to create a reference, you can create a reference without instantiating a servant:

```
interface POA {
    // ...
    Object create_reference(in CORBA::RepositoryId intf)
        raises(WrongPolicy);

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



11
Advanced Uses of the POA
Copyright 2000–2001 IONA Technologies



15.11 Servant Managers and Collections

In the absence of servant managers, an operation like `list` on the `Controller` interface can be implemented by iterating over a list of servant pointers and calling `_this` on each servant to obtain its reference. However, if servant managers are used, this creates a problem: not all servants are in memory, so iterating over them will not return the complete list of objects. In addition, in order to create a reference, you would have to instantiate the servant first in order to call `_this` on the servant. Of course, this would defeat the purpose of using servant managers in the first place and would make operations such as `list` prohibitively expensive.

To get around this problem, the POA offers `create_reference` and `create_reference_with_id`. Both operations create an object reference in isolation, that is, without the need for an instantiated servant. `create_reference` requires `SYSTEM_ID` and generates a unique object ID for the reference, whereas `create_reference_with_id` requires `USER_ID` and you have supply an object ID explicitly. (This is analogous to the difference between `activate_object` and `activate_object_with_id`.)¹

Both operations require you to supply the repository ID for interface the reference will denote (such as `"IDL:acme.com/CCS/Thermometer:1.0"`).

Once you have created a reference this way, you can return it to clients as usual. The only difference to calling `_this` is that the reference does not have a servant yet. When a client invokes an operation via the reference, a servant manager can take care of bringing the servant into memory as usual.

1. If you use `create_reference`, you do not know what the generated object ID is unless you also call `reference_to_id`. (See page 15-32.)

Because operations such as `list` are strongly typed and return something other than type `Object`, you must call `_narrow` before you can return the reference:

```

CCS::Controller::ThermometerSeq *
Controller_impl::
list() throw(CORBA::SystemException)
{
    CCS::Controller::ThermometerSeq_var return_seq
        = new CCS::Controller::ThermometerSeq;
    CORBA::ULong index = 0;

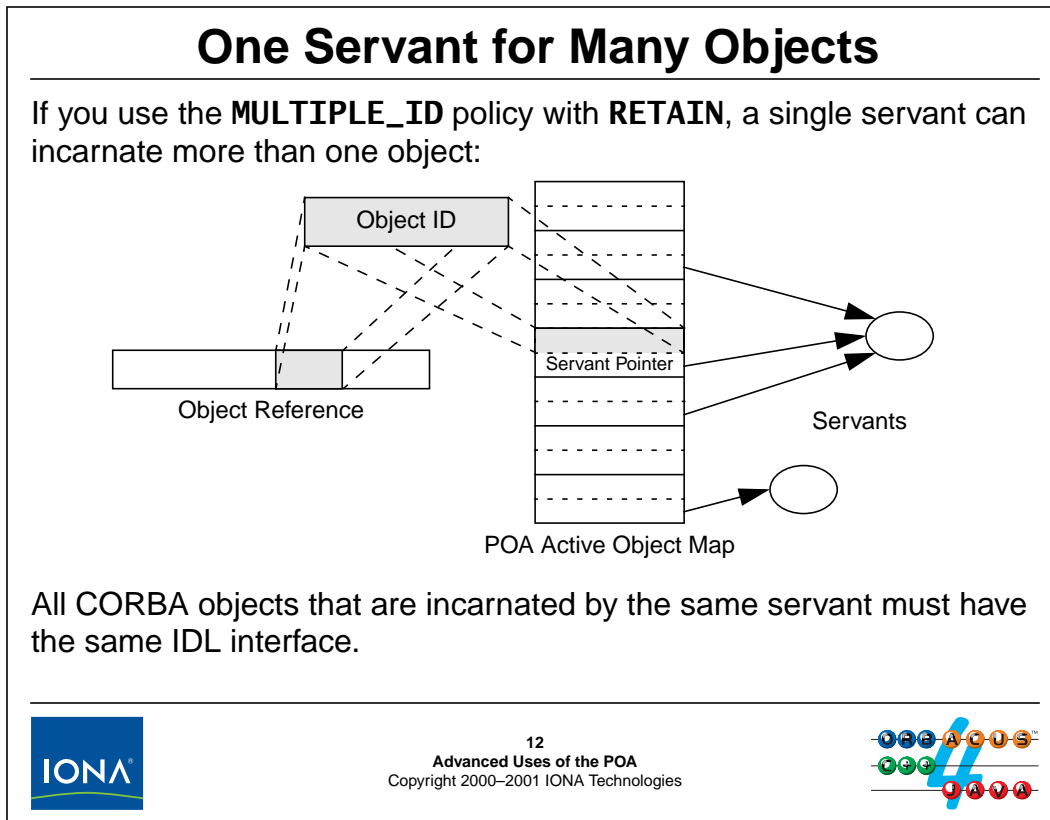
    // Iterate over the database contents (or other list
    // of existing objects) and create reference for each.
    while (more objects remain) {
        // Get asset number from database and convert to OID.
        CCS::AssetType anum = ...;
        ostream ostr;
        ostr << anum << ends;
        PortableServer::ObjectId_var oid =
            PortableServer::string_to_ObjectId(ostr.str());
        ostr.rdbuf()->freeze(0);

        // Use object state to work out which type of device
        // we are dealing with and which POA to use.
        const char * rep_id;
        PortableServer::POA_var poa;
        if (device is a thermometer) {
            rep_id = "IDL:acme.com/CCS/Thermometer:1.0";
            poa = ...; // Thermometer POA
        } else {
            rep_id = "IDL:acme.com/CCS/Thermostat:1.0";
            poa = ...; // Thermostat POA
        }

        // Create reference
        CORBA::Object_var obj =
            poa->create_reference_with_id(oid, rep_id);
        // Narrow and store in our return sequence.
        return_seq->length(index + 1);
        if (device is a thermometer)
            return_seq[index++] = CCS::Thermometer::_narrow(obj);
        else
            return_seq[index++] = CCS::Thermostat::_narrow(obj);
    }
    return return_seq._retn();
}

```

Note that this code relies on an ORB-specific optimization, namely, that a call to `_narrow` with a precise type match is short-cut by the ORB and not sent to the target servant. ORBacus provides this optimization (but other ORBs may not). Without the optimization, this technique is useless.



15.12 One Servant for Many Objects

If you use the **MULTIPLE_ID** policy with **RETAIN**,² you can add the same servant to the AOM several times with different object IDs. This means that there are as many CORBA objects as there are entries in the AOM, but that some of these objects happen to be represented by the same servant. This idea is attractive if we need a server that supports a large number of CORBA objects that are used by clients simultaneously, but must limit its memory footprint.

If you use **MULTIPLE_ID**, you must ensure that, if you register the same servant for multiple objects, all objects support the same interface (have the same repository ID), otherwise you will get unpredictable behavior at run time (such as clients receiving a **BAD_OPERATION** exception, or even silent and potentially fatal run time errors).

Using a single servant for multiple CORBA objects means that we can no longer maintain the object state in the servant, because the identity of the CORBA object for a particular request is not longer implicit in the C++ servant that handles the request. In other words, the same single servant must now pretend to be different CORBA objects for different requests, on a per-request basis. Servants use the `Current` interface to achieve this.

² For **NON_RETAIN** POAs, neither **MULTIPLE_ID** nor **UNIQUE_ID** have any effect because the mapping from object IDs to servants is managed by the application.

The Current Object

The **Current** object provides information about the request context to an operation implementation:

```
module PortableServer {
    // ...
    exception NoContext {};

    interface Current : CORBA::Current {
        POA          get_POA() raises(NoContext);
        ObjectId     get_object_id() raises(NoContext);
    };
};
```

The **get_POA** and **get_object_id** operations must be called from within an executing operation (or attribute access) in a servant.

Note: You must resolve the Root POA before resolving **POACurrent**.



13
Advanced Uses of the POA
Copyright 2000–2001 IONA Technologies



15.13 The Current Object

The **Current** object delivers information about the currently executing request to a servant (and therefore must be invoked only from within the context of an operation on a servant). Calling **get_POA** or **get_object_id** from outside the context of an executing operation raises **NoContext**.

You obtain access to the **Current** object by calling **resolve_initial_references**:

```
CORBA::Object_var obj =
    orb->resolve_initial_references("POACurrent");
```

```
PortableServer::Current_var cur =
    PortableServer::Current::_narrow(obj);
```

You only need to call **resolve_initial_references** once to obtain the **Current** object and, thereafter, you can use that same object reference for all POAs and servants. Invocations on the **Current** object automatically return the correct information for the currently executing request, even in multi-threaded servers. In effect, **Current** is a singleton object that returns thread-specific data.

The implementation of our servants now must use the **Current** object on entry to each operation to find out what its identity should be for the current request, and use that information to act on the correct state. Assuming that we store a reference to the **Current** object in the global variable **poa_current**, we can write a helper function that retrieves the asset number for the current request. (You would usually make this function a private member function of the servant):

```

CCS::AssetType
Thermometer_impl::
get_anum()
throw(CORBA::SystemException)
{
    // Get object ID from Current object
    PortableServer::ObjectId_var oid =
        poa_current->get_object_id();

    // Check that ID is valid
    CORBA::String_var tmp;
    try {
        tmp = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM &) {
        throw CORBA::OBJECT_NOT_EXIST();
    }

    // Turn string into asset number
    istrstream istr(tmp.in());
    CCS::AssetType anum;
    istr >> anum;
    if (str.fail())
        throw CORBA::OBJECT_NOT_EXIST();
    return anum;
}

```

To implement the servant, we call this helper function on entry to every operation to get the asset number, and in turn use the asset number as a key to locate the data (typically, by looking it up in a database or interrogating a network):

```

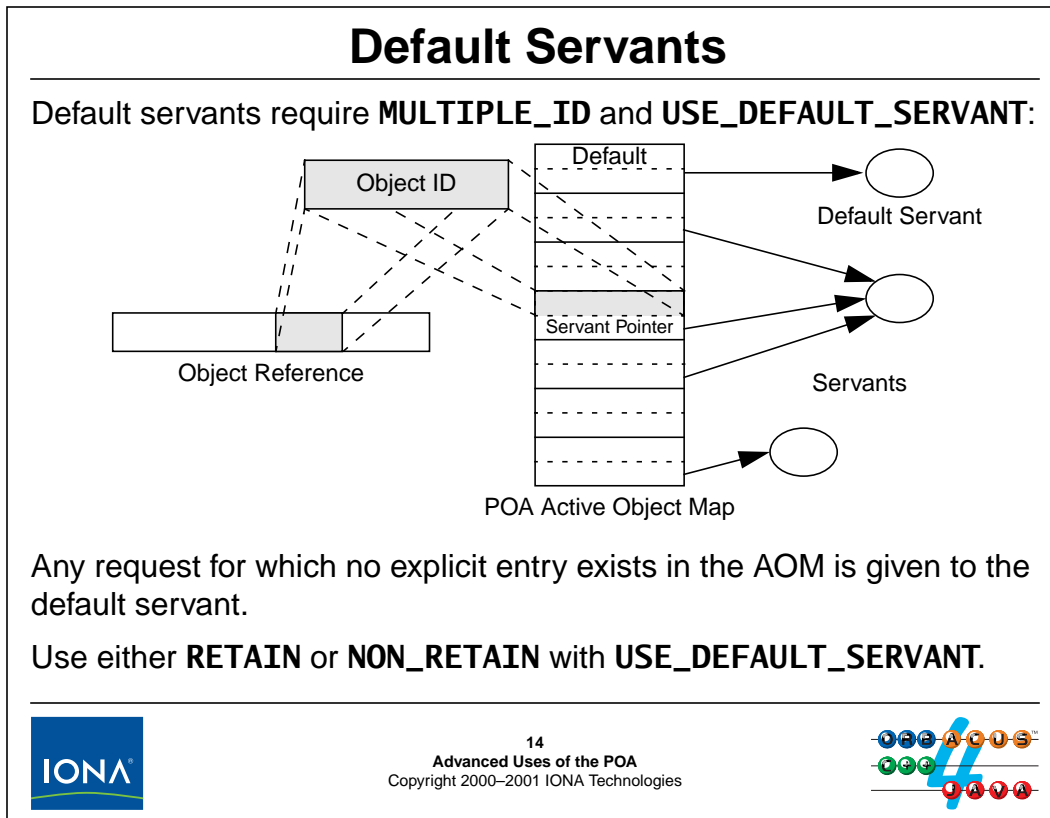
CCS::LocType
Thermometer_impl::
location() throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_anum();    // Who are we?

    // Get location string from the database
    CORBA::String_var loc = db_get_field(anum, "LocationField");
    return loc._retn();
}

void
Thermometer_impl::
location(const char * loc) throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_anum();    // Who are we?

    // Set location string in the database
    db_set_field(anum, "LocationField", loc);
}

```



15.14 Default Servants

If you set the `USE_DEFAULT_SERVANT` policy, the POA allows you to register a default servant:

- If you combine `USE_DEFAULT_SERVANT` with `RETAIN`, the POA first attempts to locate a matching servant in the AOM. If no servant is explicitly for the object ID in the request, the POA passes the request to the default servant. (This situation is shown above.)
- If you combine `USE_DEFAULT_SERVANT` with `NON_RETAIN`, all requests go to the default servant.

The first combination, even though appealing at first, is probably overkill. For example, you can achieve the same effect by using two POAs, one with a default servant and one without. This is not only simple, but also makes call dispatch to the default servant faster because it avoids the two-step process of locating a servant by first looking for an explicitly registered servant and passing the request to the default servant only if no explicitly registered servant is found. For this reason, we recommend that, if you use default servants, you should use POAs with the `NON_RETAIN` policy.

For `USE_DEFAULT_SERVANT` and `NON_RETAIN`, you must use a separate POA for each interface that is implemented by a default servant (because a single servant cannot implement two interfaces simultaneously). For `USE_DEFAULT_SERVANT` and `RETAIN`, you can mix interface types on the same POA, but all requests that are directed to the default servant must be for objects with the same interface.

As for servant managers, you must explicitly register the default servant for a POA:

```
interface POA {
    // ...
    Servant get_servant() raises(NoServant, WrongPolicy);
    void    set_servant(in Servant s) raises(WrongPolicy);
};
```

`get_servant` returns the current default servant and raises `NoServant` if none has been set.

`set_servant` sets the default servant. You can call `set_servant` more than once to change the default servant. (However, it is unlikely that you would ever do this unless you wanted to dynamically replace the code for the default servant at run time.)

The POA calls `_add_ref` on the default servant during the call to `set_servant`. This means that, if you use a reference-counted default servant, you can call `_remove_ref` immediately after calling `set_servant`. If you do this, the default servant will be automatically destroyed when its POA is destroyed. Otherwise, you must destroy the default servant explicitly once it is no longer needed.

Also note that the POA calls `_add_ref` when you call `get_servant`. This means that you must eventually call `_remove_ref` on the returned servant, otherwise the reference count will be left too high. An easy way to ensure this happens is to use a `ServantBase_var` when calling `get_servant`:

```
PortableServer::ServantBase_var servant
    = some_poa->get_servant();

// ServantBase_var destructor calls _remove_ref() eventually...

// If we want the actual type of the servant again, we must
// use a down-cast:
Thermometer_impl * dflt_serv =
    dynamic_cast<Thermometer_impl *>(servant);
```

Trade-Offs for Default Servants

Default servants offer a number of advantages:

- simple implementation
- POA and object ID can be obtained from **Current**
- ideal as a front end to a back-end store
- servant is completely stateless
- infinite scalability!

The downside:

- possibly slow access to servant state



15
Advanced Uses of the POA
Copyright 2000–2001 IONA Technologies



15.15 Trade-Offs for Default Servants

The main motivation for using default servants is scalability because a single C++ instance can be used to simultaneously represent an unlimited number of CORBA objects. The implementation becomes very simple: each operation first retrieves the object ID to identify the CORBA object and then uses the identity of the object to retrieve its state. As a result, the servant itself is completely stateless, which makes this approach ideal if you want to use a CORBA server as a front end to a database that may be updated independently. Provided that the database provides atomic access to servant state, there are no cache coherency issues that could arise through independent updates.

Using default servants, the scalability of a CORBA server is no longer limited by its memory consumption and only depends on the bandwidth to the database and the number of parallel invocations you can afford to support.

The unlimited scalability of default servants comes at a price though: each access to servant state takes longer than if you would use multiple servants that hold their state in member variables and write to the database only occasionally when they are updated. However, if you have accesses with good locality of reference and a database with effective caching, the performance penalty of default servants can be surprisingly small, so the technique is well worth exploring.

POA Activators

You can create POAs on demand, similar to activating servants on demand:

```
module PortableServer {
  // ...
  interface AdapterActivator {
    boolean unknown_adapter(in POA parent, in string name);
  };
};
```

This is a callback interface you provide to the ORB.

If a request for an unknown POA arrives, the ORB invokes the **unknown_adapter** operation to allow you to create the POA.



15.16 POA Activators

The POA offers a mechanism to activate POAs on demand. The idea is similar to those of servant managers: by providing a callback mechanism, it is possible to avoid having all POAs in memory at all times. The main motivation for POA activators is to permit a server to easily implement optimistic caching (also known as pre-fetching). This is important if, for example, a number of objects are always used as a group. By using a separate POA for each group, you can use a POA activator to activate all the servants for the objects in a group at once, so they are available immediately, without further initialization while they are in use. Once you no longer require a group, you can simply invoke `destroy` on the POA to reclaim the resources that were used by the group.

To implement a POA activator, you must derive a servant class from `PortableServer::AdapterActivator`:

```
class POA_Activator_impl :
  public virtual POA_PortableServer::AdapterActivator {
public:
    POA_Activator_impl() {}
    virtual ~POA_Activator_impl() {}
    virtual CORBA::Boolean
      unknown_adapter(
        PortableServer::POA_ptr parent,
        const char * name
      ) throw(CORBA::SystemException);
};
```

Implementing POA Activators

The **parent** parameter allows you to get details of the parent POA (particularly, the name of the parent POA).

The **name** parameter provides the name for the new POA.

While **unknown_adapter** is running, requests for the new adapter are held pending until the activator returns.

The implementation of the activator must decide on a set of policies for the new POA and instantiate it.

If optimistic caching is used, the activator must instantiate the servants for the POA. (If combined with **USE_SERVANT_MANAGER**, a subset of the servants can be instantiated instead.)

On success, the activator must return true to the ORB (which dispatches the request as usual.) A false return value raises **OBJECT_NOT_EXIST** in the client.



15.17 Implementing POA Activators

To implement a POA activator, you only need to create the `unknown_adapter` member function:

```
CORBA::Boolean
POA_Activator_impl::
unknown_adapter(
    PortableServer::POA_ptr parent,
    const char *          name
) throw(CORBA::SystemException)
{
    // Check which adapter is being activated and
    // create appropriate policies. (Might use pre-built
    // policy list here...)
    CORBA::PolicyList policies;
    if (strcmp(name, "Some_adapter") == 0) {
        // Create policies for "Some_adapter"...
    } else if (strcmp(name, "Some_other_adapter") == 0) {
        // Create policies for "Some_other_adapter"...
    } else {
        // Unknown POA name
        return false;
    }
}
```

```
// Select POA manager for new adapter (parent POA
// manager in this example).
PortableServer::POAManager_var mgr = parent->the_POAManager();

// Create new POA.
try {
    PortableServer::POA_var child =
        parent->create_POA(name, mgr, policies);
} catch (const PortableServer::POA::AdapterAlreadyExists &) {
    return false;
} catch (...) {
    return false;
}

// For optimistic caching, activate servants here...

return true;
}
```

Note that the code ensures that false is returned if the POA being activated already exists. This is necessary because (at least for multi-threaded servers), another thread may have already activated the adapter.

Registering POA Activators

An adapter activator must be registered by setting the POA's **the_activator** attribute:

```
interface POA {
    // ...
    attribute AdapterActivator the_activator;
};
```

You can change the adapter activator of an existing POA, including the Root POA.

By attaching an activator to all POAs, a request for a POA that is low in the POA hierarchy will automatically activate all parent POAs that are needed.



15.18 Registering POA Activators

To register a POA activator, you must set the `the_activator` attribute of its parent POA. Note that, by attaching an activator to all POAs (including the Root POA), you can ensure that an incoming request automatically activates the necessary parent POAs as well. For example, the code of the preceding example can be modified such that the POA activator adds itself to the POAs it creates:

```
CORBA::Boolean
POA_Activator_impl::
unknown_adapter(
    PortableServer::POA_ptr parent,
    const char * name
) throw(CORBA::SystemException)
{
    // ...

    // Create new POA.
    try {
        PortableServer::POA_var child =
            parent->create_POA(name, mgr, policies);
        PortableServer::AdapterActivator_var act = _this();
        child->the_activator(act);
    } catch (const PortableServer::POA:AdapterAlreadyExists &) {
        return false;
    }
```

```
    } catch (...) {  
        return false;  
    }  
  
    // ...  
}
```

In main, we use the same adapter activator as the Root POA's activator:

```
// ...  
  
PortableServer::POA_var root_poa = ...;  
  
// Create activator servant.  
POA_Activator_impl act_servant;  
  
// Register activator with Root POA.  
PortableServer::AdapterActivator_var act = act_servant._this();  
root_poa->the_activator(act);  
  
// ...
```

Note that, for POA activators to work, the POA manager for the Root POA must be active if the server uses indirect binding via the implementation repository (see Unit 22); otherwise, the server has no transport endpoint for the incoming request that should result in activation of a new POA. However, it is not necessary for POAs that are dynamically activated to actually use the Root POA manager; you can use any POA manager you like but, for indirect binding, activation will work only when the Root POA manager is active.

If your server does not use the implementation repository, the POA manager of the to-be-activated POA must be in the active state.

Finding POAs

The **find_POA** operation locates a POA:

```
// In module PortableServer: typedef sequence<POA> POAList;
interface POA {
    // ...
    POA find_POA(in string name, in boolean activate_it)
        raises(AdapterNonExistent);
    readonly attribute POAList the_children;
    readonly attribute POA the_parent;
};
```

You must invoke **find_POA** on the correct parent (because POA names are unique only within their parent POA).

If **activate_it** is true and the parent has an adapter activator, **unknown_adapter** will be called to create the child POA.

You can use this to instantiate all your POAs by simply calling **find_POA**.



15.19 Finding POAs

The `the_children` attribute retrieves a list of all the (current) children of a POA.

The `the_parent` attribute returns the parent of a POA. (The parent of the Root POA is a nil reference.)

The `find_POA` operation returns the child POA with the supplied name. If the `activate_it` parameter is true, calls to `find_POA` trigger the parent's POA activator. This is particularly useful if you use adapter activators anyway because you can create new POAs by simply calling `find_POA` from main. This has the advantage that all your POA creation code is kept in one place and that you can centralize the mapping from POA names to policies (for example, in a single lookup table that maps POA names to policy lists).

```
// ...

PortableServer::POA_var root_poa = ...;

// Create activator servant.
POA_Activator_impl act_servant;

// Register activator with Root POA.
PortableServer::AdapterActivator_var act = act_servant._this();
root_poa->the_activator(act);

// Use find_POA to create a POA hierarchy. The POAs will be
// created by the adapter activator.
```



```
PortableServer::POA_var ctrl_poa
    = root_poa->find_POA("Controller", true);
PortableServer::POA_var thermometer_poa
    = ctrl_poa->find_POA("Thermometers", true);
PortableServer::POA_var thermostat_poa
    = ctrl_poa->find_POA("Thermostats", true);

// Activate POAs...
```

Identity Mapping Operations

The POA offers a number of operations to map among object references, object IDs, and servants:

```
interface POA {
    // ...
    ObjectId servant_to_id(in Servant s)
        raises(ServantNotActive, WrongPolicy);
    Object servant_to_reference(in Servant s)
        raises(ServantNotActive, WrongPolicy);
    Servant reference_to_servant(in Object o)
        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);
};
```



20
Advanced Uses of the POA
Copyright 2000–2001 IONA Technologies



15.20 Identity Mapping Operations

The POA provides operations that allow you to map among references, object IDs, and servants.

ObjectId servant_to_id(in Servant s)
raises(ServantNotActive, WrongPolicy)

This operation returns the object ID for a servant. The behavior depends on whether you invoke `servant_to_id` from inside an executing operation on the specified servant, or from the outside:

- If called from inside an executing operation on the specified servant, the operation returns the object ID for the current request (that is, the semantics are the same as calling `get_object_id` on the `Current` object).
- If called from outside an executing operation on the specified servant, the POA must
 - either use the `RETAIN` policy, together with `UNIQUE_ID` or `IMPLICIT_ACTIVATION`,
 - or the `USE_DEFAULT_SERVANT` policy.

`servant_to_id` raises `WrongPolicy` if the policies on the POA do not match these criteria.

The behavior when called from outside an executing operation on the specified servant is as follows:

- If the specified servant is in the AOM and `UNIQUE_ID` is in effect, the operation returns that servant's ID.
- If the POA uses `IMPLICIT_ACTIVATION` (which implies `SYSTEM_ID`) and the servant is not in the AOM, it implicitly activates the servant with a new object ID and returns that ID. This happens whether `UNIQUE_ID` or `MULTIPLE_ID` is in effect and whether you

use `USE_DEFAULT_SERVANT` or `USE_ACTIVE_OBJECT_MAP_ONLY` and is almost certainly not what you want, so we suggest you avoid `IMPLICIT_ACTIVATION`.

- If neither of the preceding conditions holds, the operation raises `ServantNotActive`.

**Object `servant_to_reference(in Servant s)`
raises(`ServantNotActive`, `WrongPolicy`)**

This operation returns the object reference for a servant. The behavior depends on whether you invoke `servant_to_reference` from inside an executing operation on the specified servant, or from the outside:

- If called from inside an executing operation on the specified servant, the operation returns the reference for the current request (that is, the semantics are the same as calling `get_object_id` on the `Current` object and creating a reference with that object ID).
- If called from outside an executing operation on the specified servant, the POA must
 - either use the `RETAIN` policy, together with `UNIQUE_ID` or `IMPLICIT_ACTIVATION`,
 - or the `USE_DEFAULT_SERVANT` policy.

`servant_to_reference` raises `WrongPolicy` if the policies on the POA do not match these criteria.

The behavior when called from outside an executing operation on the specified servant is as follows:

- If the specified servant is in the AOM and `UNIQUE_ID` is in effect, the operation returns that servant's reference.
- If the POA uses `IMPLICIT_ACTIVATION` (which implies `SYSTEM_ID`) and the servant is not in the AOM, it implicitly activates the servant with a new object ID and returns a reference containing that ID. This happens whether `UNIQUE_ID` or `MULTIPLE_ID` is in effect and whether you use `USE_DEFAULT_SERVANT` or `USE_ACTIVE_OBJECT_MAP_ONLY` and is almost certainly not what you want, so we suggest you avoid `IMPLICIT_ACTIVATION`.
- If neither of the preceding conditions holds, the operation raises `ServantNotActive`.

**Servant `reference_to_servant(in Object o)`
raises(`ObjectNotActive`, `WrongAdapter`, `WrongPolicy`)**

Calling `reference_to_servant` on a POA other than the one that created the reference raises `WrongAdapter`.

Calling `reference_to_servant` on a POA that does not use either `RETAIN` or `USE_DEFAULT_SERVANT` raises `WrongPolicy`.

Otherwise, the operation returns the servant for an object reference:

- If the POA uses `RETAIN` and the object denoted by the reference is in the AOM, the operation returns the corresponding servant.
- Otherwise, if the POA uses `USE_DEFAULT_SERVANT`, the operation returns the default servant.
- Otherwise, the operation raises `ObjectNotActive`.

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

Calling `reference_to_id` on a POA other than the one that created the reference raises `WrongAdapter`. Otherwise, `reference_to_id` returns the object ID encapsulated in the reference. (The `WrongPolicy` exception is currently not raised and reserved for future extensions.)

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

Calling `id_to_servant` on a POA that does not have either the `RETAIN` policy or the `USE_DEFAULT_SERVANT` policy raises `WrongPolicy`. Otherwise, the operation behaves as follows:

- If the specified ID is in the AOM, `id_to_servant` returns the corresponding servant.
- Otherwise, if the POA uses `USE_DEFAULT_SERVANT`, `id_to_servant` returns the default servant.
- Otherwise, the operation raises `ObjectNotActive`.

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

Calling `id_to_reference` on a POA that does not use the `RETAIN` policy raises `WrongPolicy`. Otherwise, if an object with the specified ID is currently active, the operation returns an object reference that encapsulates the specified ID. If no object with the specified ID is active, the operation raises `ObjectNotActive`.

16. Exercise: Writing Servant Locators

Summary

In this unit, you will add a servant locator to the server we presented in Unit 14.

Objectives

By the completion of this unit, you will know how to implement servant locators and how to create references on demand without having to instantiate a servant.

16.1 Source Files and Build Environment

You will find this exercise in your `locator` directory. The files in this directory are the same as for Unit 13.

16.2 Server Operation

The server source code provided to you for this exercise implements the solution presented in Unit 14. The purpose of this exercise is to implement servant locators for this server.

With servant locators, not all devices are in memory at a time. This means that, when we return an object reference (such as for the `list` operation), we cannot call simply call `_this` because the servant may not be in memory. This means that all references returned by the server need to be created without servants, using `create_reference_with_id`. The server uses this technique for all operations, including the factory operations and it relies on a servant manager to actually instantiate a servant for a device when it is needed.

Note that, as a result, the constructor of the controller no longer instantiates a servant for each device on start-up. Instead, it simply reads the list of asset numbers from `CCS_assets` to update its notion of what devices exists, without instantiating a servant.

The server uses a single servant locator that is shared between the `Thermometers` and `Thermostats` POAs. This means that, in `preinvoke`, you must make a decision as to what type of servant to instantiate based on the POA name.

16.3 What You Need to Do

Step 1

Read the body of the `run` function. Note that it calls a modified version of `create_persistent_POA` with a defaulted third argument. If no third argument is supplied, the function creates a persistent POA with `RETAIN` as in Unit 14. If a reference to a servant locator is passed as the third argument, the function instead creates a POA that is suitable for use with a servant locator.

The body of `create_persistent_POA` is empty. Implement this function. (Note: much of the body of this function is the same as for Unit 14, so you can save some time by using that as a starting point.)

Step 2

The `make_dref` function is used to create a reference without having to instantiate a servant. Note that, depending on what the type of the actual device on the network is, `make_dref` must create either a thermometer or a thermostat reference (using the correct POA and repository ID). Implement this function. (Note: use the `make_oid` helper function to create the object ID.)

Step 3

The body of `preinvoke` on the servant locator is empty. Implement this operation. (You can gain some insight as to the correct implementation by looking at the other operations in the controller.)

Step 4

The body of `postinvoke` on the servant locator is empty. Implement this operation and test your server. For this simple exercise, deallocate the servant in `postinvoke` instead attempting to keep it around for later requests.

17. Solution: Writing Servant Locators

17.1 Solution

```

static PortableServer::POA_ptr
create_persistent_POA(
    const char *          name,
    PortableServer::POA_ptr parent,
    PortableServer::ServantManager_ptr locator
        = PortableServer::ServantLocator::_nil())
{
    // Create policy list for simple persistence
    CORBA::PolicyList pl;
    CORBA::ULong len = pl.length();
    pl.length(len + 1);
    pl[len++] = parent->create_lifespan_policy(
        PortableServer::PERSISTENT
    );
    pl.length(len + 1);
    pl[len++] = parent->create_id_assignment_policy(
        PortableServer::USER_ID
    );
    pl.length(len + 1);
    pl[len++] = parent->create_thread_policy(
        PortableServer::SINGLE_THREAD_MODEL
    );
    pl.length(len + 1);
    pl[len++] = parent->create_implicit_activation_policy(
        PortableServer::NO_IMPLICIT_ACTIVATION
    );

    // Check if we need to register a servant locator
    if (!CORBA::is_nil(locator)) {
        pl.length(len + 1);
        pl[len++] = parent->create_servant_retention_policy(
            PortableServer::NON_RETAIN
        );
        pl.length(len + 1);
        pl[len++] = parent->create_request_processing_policy(
            PortableServer::USE_SERVANT_MANAGER
        );
    }

    // Get parent POA's POA manager
    PortableServer::POAManager_ptr pmanager
        = parent->the_POAManager();

    // Create new POA
    PortableServer::POA_ptr poa =
        parent->create_POA(name, pmanager, pl);
}

```



```

    // Register servant locator, if required
    if (!CORBA::is_nil(locator))
        poa->set_servant_manager(locator);

    // Clean up
    for (CORBA::ULong i = 0; i < len; ++i)
        pl[i]->destroy();

    return poa._retn();
}

```

Step 2

```

// Helper function to create object references.

CCS::Thermometer_ptr
make_dref(CCS::AssetType anum)
{
    // Convert asset number to OID
    PortableServer::ObjectId_var oid = make_oid(anum);

    // Look at the model via the network to determine
    // the repository ID and the POA.
    char buf[32];
    if (ICP_get(anum, "model", buf, sizeof(buf)) != 0)
        abort();
    const char * rep_id;
    PortableServer::POA_ptr poa;
    if (strcmp(buf, "Sens-A-Temp") == 0) {
        rep_id = "IDL:acme.com/CCS/Thermometer:1.0";
        poa = Thermometer_impl::poa();
        CORBA::Object_var obj
            = poa->create_reference_with_id(oid, rep_id);
        return CCS::Thermometer::_narrow(obj);
    } else {
        rep_id = "IDL:acme.com/CCS/Thermostat:1.0";
        poa = Thermostat_impl::poa();
        CORBA::Object_var obj
            = poa->create_reference_with_id(oid, rep_id);
        return CCS::Thermostat::_narrow(obj);
    }
}

```

Step 3

```

PortableServer::Servant
DeviceLocator::
preinvoke(
    const PortableServer::ObjectId &    oid,
    PortableServer::POA_ptr             adapter,

```

```

        const char *                operation,
        PortableServer::ServantLocator::Cookie & the_cookie
    ) throw(CORBA::SystemException, PortableServer::ForwardRequest)
    {
        CCS::AssetType anum = make_anum(oid);
        if (!Thermometer_impl::m_ctrl->exists(anum))
            throw CORBA::OBJECT_NOT_EXIST();
        CORBA::String_var poa_name = adapter->the_name();
        if (strcmp(poa_name, "Thermometers") == 0)
            return new Thermometer_impl(anum);
        else
            return new Thermostat_impl(anum);
    }

```

Step 4

```

void
DeviceLocator::
postinvoke(
    const PortableServer::ObjectId &        oid,
    PortableServer::POA_ptr              adapter,
    const char *                          operation,
    PortableServer::ServantLocator::Cookie the_cookie,
    PortableServer::Servant               the_servant
) throw(CORBA::SystemException)
{
    the_servant->_remove_ref();
}

```

18. ORBacus Configuration

Summary

This unit covers the configuration mechanism of ORBacus, which controls value-added and non-standard aspects of the ORB.

Objectives

By the completion of this unit, you will know how to control configuration parameters and understand how defaults are applied at various levels to arrive at an effective configuration for a process.

Introduction

Some aspects of behavior for ORBacus are controlled by properties.

Properties are scoped name–value pairs. The name is a variable such as `ooc.orb.client_timeout`. The value of a property is a string.

ORBacus uses properties to change its behavior in some way.

There are properties to control threading models, to control the return value from `resolve_initial_references` for different tokens, to change connection management strategies, etc.

The property configuration mechanism is not standardized and therefore specific to ORBacus.

Property values are read once only, on process start-up. Changing the value of a property has no effect on running processes!



1
ORBacus Configuration
Copyright 2000–2001 IONA Technologies



18.1 Introduction

The specification falls short in a number of areas when it comes to ORB behavior. For example, the specification loses few words about threads, or how to influence the connection management algorithm that is used by clients and servers.

Because the non-standard aspects of ORB behavior are just as important as the standard ones, ORBacus offers a proprietary mechanism to control these aspects, known as properties. A property is a name–value pair. The name is a scoped name such as `ooc.orb.client_timeout` or `ooc.orb.oa.conc_model`. (All ORBacus properties use `ooc` as the first element of their name.) The value of a property is a string which, depending on the property, may be further constrained. For example, some properties require a numeric value and the string you use to set the value must parse as a number.

ORBacus processes property values once only, during initialization. This means that, if you change the value of a property, it only affects processes you start after the change, not processes that are running while you make the change.

Defining Properties

Properties can be defined in a number of places:

1. in a Windows registry key under
`HKEY_LOCAL_MACHINE\Software\OOC\Properties\<name>`
2. in a Windows registry key under
`HKEY_CURRENT_USER\Software\OOC\Properties\<name>`
3. in a configuration file specified by the `ORBACUS_CONFIG` environment variable
4. by setting a property from within the program
5. in a configuration file specified on the command line
6. by using a command-line option

Property values are retrieved using all these means (if applicable).

Higher numbers have higher precedence.



2
ORBacus Configuration
Copyright 2000–2001 IONA Technologies



18.2 Defining Properties

You can define properties in a number of ways, depending on the platform. When ORBacus retrieves property values, it follows the above sequence in order and overrides property values retrieved during earlier steps with values retrieved during later steps. This means that property values supplied on the command line override all other settings because the final step has the highest precedence.

Most properties have a default value; that value is silently applied if the property is not set.

Setting Properties in the Registry

To set a property in the Windows registry, use the property name, replacing the “.” by “\”. For example, the property `ooc.orb.id` can be set by setting the value of:

```
HKEY_LOCAL_MACHINE\Software\OOC\Properties\ooc\orb\id
```

Defaults for properties that affect all processes on the system can be set under `HKEY_LOCAL_MACHINE`.

Defaults for properties that affect only the current user can be set under `HKEY_CURRENT_USER`.



3
ORBacus Configuration
Copyright 2000–2001 IONA Technologies



18.3 Setting Properties in the Registry

Under Windows, you can establish system-wide and per-user default values for properties by setting registry keys under `HKEY_LOCAL_MACHINE` and `HKEY_CURRENT_USER`, respectively.

To determine the path to the key holding a property value, replace the “.” characters in a property name by “\” and prefix the root path depending on whether you want to establish a per-machine or per-user default value. For example, you can set

```
HKEY_CURRENT_USER\Software\OOC\Properties\ooc\orb\oa\thread_pool
```

to 20 if you want to use a default thread pool size of 20 for the current user.

Setting Properties in a Configuration File

You can create a configuration file containing property values. For example:

```
# Default client concurrency model is threaded
ooc.orb.conc_model=threaded

# Default server concurrency model is a pool of 20 threads
ooc.orb.oa.conc_model=thread_pool
ooc.orb.oa.thread_pool=20

# Default naming service is on HostA, port 5000
ooc.orb.service.NameService=corbaloc::HostA:5000/NameService
```

Trailing white space is ignored and is not part of the property.



4
ORBacus Configuration
Copyright 2000–2001 IONA Technologies



18.4 Setting Properties in a Configuration File

You can set properties in a configuration file. Lines beginning with “#” are comments and are ignored. You can insert blank lines as you wish. Note that trailing white space characters are *not* ignored and become part of the property value.

Once you have created a property file, you can set the **ORBACUS_CONFIG** environment variable (UNIX only) to indicate the path to the configuration file. For example, using a Bourne shell or similar:

```
ORBACUS_CONFIG=/home/michi/.ob.config
export ORBACUS_CONFIG
```

Note that it is best to use an absolute pathname. Relative pathnames are legal, but will work only if they are correct relative to the current working directory.

Setting Properties Programmatically

You can set properties from within your program using the `OB::Properties` class and `OBCORBA::ORB_init`.

```
// Get default properties (established by config file)
OB::Properties_var dflt
    = OB::Properties::getDefaultProperties();

// Initialize a property set with the defaults
OB::Properties_var props = new OB::Properties(dflt);

// Set the properties we want
props->setProperty("ooc.orb.conc_model", "threaded");
props->setProperty("ooc.orb.oa.conc_model", "thread_pool");
props->setProperty("ooc.orb.oa.thread_pool", "20");

// Initialize the ORB with the given properties
CORBA::ORB_var orb = OBCORBA::ORB_init(argc, argv, props);
```



5
ORBacus Configuration
Copyright 2000–2001 IONA Technologies



18.5 Setting Properties Programmatically

You can set properties from within your programming using the `OB::Properties` class. The `getDefaultProperties` member returns a property set that is initialized with the defaults that may be present in a configuration file. To set properties explicitly, you must copy-construct a property set from the defaults and then set the properties you want on that copy. Finally, instead of calling `CORBA::ORB_init`, call the ORBacus-specific `OBCORBA::ORB_init` function, which accepts the property set as the third argument.

Setting Properties from the Command Line

You can pass the `-ORBconfig <pathname>` option to a process.

The specified file overrides the defaults that are taken from registry keys or the `ORBACUS_CONFIG` environment variable, and overrides values that are set with `OBCORBA::ORB_init`.

You can also set most properties from the command line directly. For example:

```
./a.out -ORBconfig $HOME/ob.config \  
-ORBthreaded -OAreactive
```

Explicit property values override the defaults in a configuration file, so this process will use `ooc.orb.conc_model=threaded` and `ooc.orb.oa.conc_model=reactive`.

See the manual for a complete list of options.



6
ORBacus Configuration
Copyright 2000–2001 IONA Technologies



18.6 Setting Properties from the Command Line

Most properties can be set using command-line options. The `-ORBconfig` option allows you specify a different configuration file. Note that, if you also have a configuration file specified with the `ORBACUS_CONFIG` environment variable, that configuration file is still read to establish defaults. However, properties that are defined in the configuration file specified with `-ORBconfig` override those defaults.

You can also specify many properties explicitly on the command line. For example, `-ORBthreaded` is the same as setting the `ooc.orb.conc_model` property to `threaded`.

Commonly Used Properties

- **ooc.orb.service.<name>=<IOR>**
Specify the IOR to returned by **resolve_initial_references**.
- **ooc.orb.trace.connections=<level>**
Trace connection establishment and closure at **<level>** (0–2).
- **ooc.orb.trace.retry=<level>**
Trace connection reestablishment attempts at **<level>** (0–2).
- **ooc.orb.oa.numeric={true, false}**
Use a dotted-decimal IP address in IORs instead of a name.
- **ooc.orb.oa.port=<port>**
Set the port number to be embedded in IORs.



18.7 Commonly Used Properties

The above properties are among the ones used more commonly.

- **ooc.orb.service.<name>=<IOR>**
This property controls the object reference returned by **resolve_initial_references** for various tokens. For example, setting **ooc.orb.service.NameService=corbaloc::HostA:5000/NameService** sets the reference returned for the initial naming context. (See Section 19.20.)
You can also use the **-ORBInitRef <name>=<IOR>** command line option to set this property. For example:

```
a.out -ORBInitRef NameService=corbaloc::HostA:5000/NameService
```
- **ooc.orb.trace.connections=<level>**
This property controls the level of tracing for connection establishment and closure. Valid tracing levels are 0, 1, and 2. This property is useful for debugging, for example, if a reference will not bind correctly.
You can also use the **-ORBtrace_connections <level>** command-line option to set this property.

- **ooc.orb.trace.retry=<level>**

This property controls the level of tracing of transparent retry attempts made by clients when connections go down. Valid tracing levels are 0, 1, and 2.

You can also use the **-ORBtrace_rebind <level>** command-line option to set this property.

- **ooc.iiop.numeric={true, false}**

This property, when set to true, causes dotted-decimal IP addresses to be inserted into object references instead of domain names. This is useful if the DNS on the client side is not configured correctly and cannot resolve the domain name of the server.

You can also use the **-OAnumeric** command-line option to set this property.

- **ooc.iiop.port=<port>**

This property controls the port at which the server listens for incoming requests. That port number is inserted into object references created by the server. (See Unit 22.)

You can also use the **-OApport <port>** command-line option to set this property.

19. The Naming Service

Summary

This unit presents the OMG Naming Service in detail and explains how the Naming Service can be used to decouple clients and servers by providing an external reference exchange mechanism. The unit also covers how to solve the bootstrapping problem for clients and servers by controlling their configuration.

Objectives

By the completion of this unit, you will know how to use the Naming Service as a rendezvous point between clients and servers, and how to configure your application components so they agree on the particular Naming Service they should use.

Introduction

Copying stringified references from a server to all its clients is clumsy and does not scale.

The Naming Service provides a way for servers to advertise references under a name, and for clients to retrieve them. The advantages are:

- Clients and servers can use meaningful names instead of having to deal with stringified references.
- By changing a reference in the service without changing its name, you can transparently direct clients to a different object.
- The Naming Service solves the bootstrapping problem because it provides a fixed point for clients and servers to rendezvous.

The Naming Service is much like a white pages phone book. Given a name, it returns an object reference.



1
The Naming Service
Copyright 2000–2001 IONA Technologies



19.1 Introduction

So far, we have used stringified references to allow clients to contact a server. This is a rather clumsy way for clients to gain access to objects because copying stringified references around is cumbersome, error prone, and does not scale. The Naming Service offers a solution to this problem by offering an object reference advertising service. Servers advertise references under a name, and clients use the name to locate the reference in the naming service.

This allows clients to use meaningful names for objects instead of cryptic stringified references. In addition, it provides greater flexibility because of the extra level of indirection it adds. Finally, the Naming Service solves the bootstrapping problem because it provides a fixed point for clients and servers to rendezvous.

Terminology

- A name-to-IOR association is called a *name binding*.
- Each binding identifies exactly one object reference, but an object reference may be bound more than once (have more than one name).
- A *naming context* is an object that contains name bindings. The names within a context must be unique.
- Naming contexts can contain bindings to other naming contexts, so naming contexts can form graphs.
- *Binding* a name to a context means to add a name-IOR pair to a context.
- *Resolving* a name means to look for a name in a context and to obtain the IOR bound under that name.

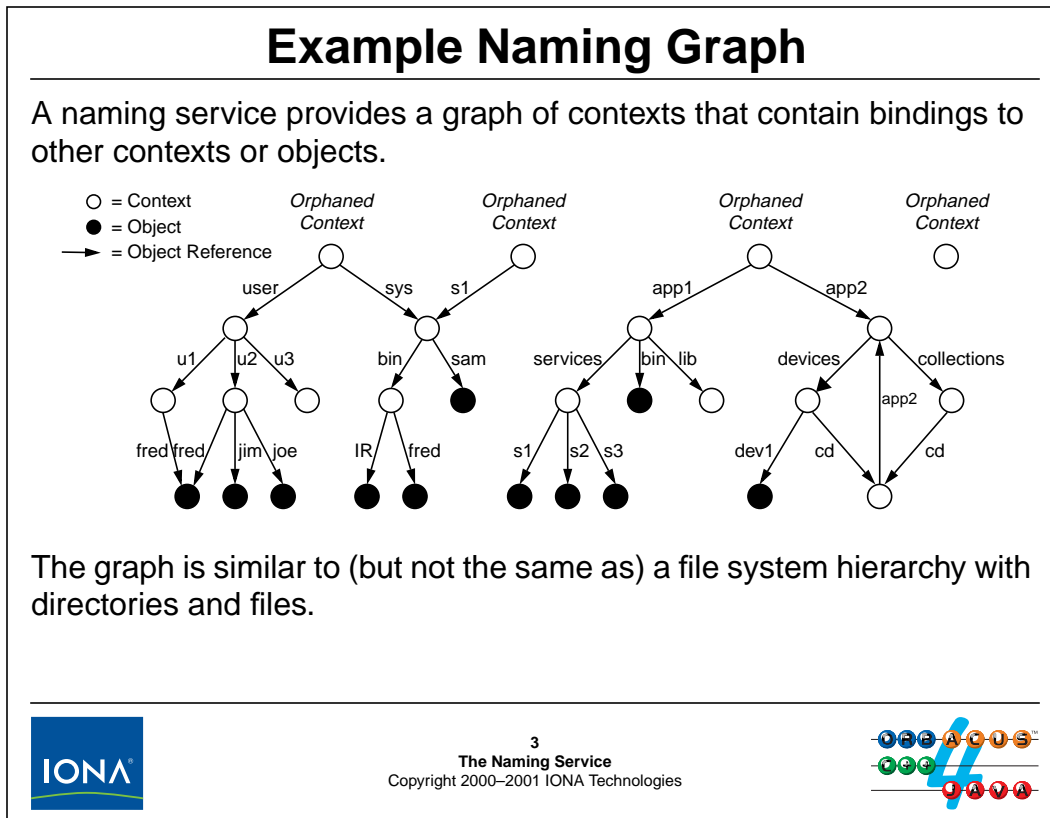


2
The Naming Service
Copyright 2000–2001 IONA Technologies



19.2 Terminology

The above slide shows the terminology that is used for the Naming Service. Conceptually, a Naming Service is similar to a file system, which uses directories and files. Naming contexts are analogous to directories: they hold name-IOR pairs, called bindings. By providing a name, you can obtain an IOR stored in a context under that name. The IOR for a binding may denote another context or an application object; by adding bindings to contexts that denote other contexts, you can connect contexts into arbitrary graphs.



19.3 Example Naming Graph

The above slide shows an example of a small naming graph. Hollow nodes denote contexts, and solid nodes denote application objects. The arcs represent bindings. Each arc is an object reference, labeled with the name under which that reference is bound in its context.

Note that this is similar to a file system hierarchy:

- Within a context, name bindings are unique. Each binding can appear only once within its parent context.
- Given a starting context, you can navigate to a target node by traversing a path from the starting context to the target node. The sequence of bindings used in the traversal forms a pathname that uniquely identifies the target object.
- The same name binding can appear multiple times provided that each binding is in a different parent context. For example, the binding `bin` appears twice in the above graph.
- A single object or context can have multiple names. For example, the sample graph uses the name bindings `sys` and `s1` for the same context. (This corresponds to the concept of multiple links to the same file or directory in a UNIX file system.)

Apart from the similarities, there are differences that distinguish a naming graph from a file system graph:

- It is possible for the graph to have contexts that have no names. Such contexts are known as *orphaned* contexts. (This is different from a normal file system, which requires every file and directory to have a name.)

- A naming graph has one or more distinguished contexts known as *initial naming* contexts. Typically, initial naming contexts are orphaned contexts (but they need not be). Conversely, if a context is orphaned, it is typically also an initial naming context. (Initial naming contexts determine the points at which clients gain access to a naming graph. An initial naming context corresponds to what we think of as the root directory of a file system.)
- A naming graph can have more than one root. Typically, each such root is also configured as an initial naming context.
- A graph can consist of several disconnected subgraphs.
- It is possible for the graph to have loops.

NOTE: Even though loops are legal, we strongly recommend that you avoid them. Loops are legal not because they are desirable but because it is impossible to prevent them. If you have loops in a graph, it is easy to get confused about which bindings are where; in addition, administrative tools become harder to use.

Naming IDL Structure

The IDL for the Naming Service has the following overall structure:

```
//File: CosNaming.idl
#pragma prefix "omg.org"
module CosNaming {
    // Type definitions here...
    interface NamingContext {
        // ...
    };
    interface NamingContextExt : NamingContext {
        // ...
    };
    interface BindingIterator {
        // ...
    };
};
```

Note that all OMG-defined IDL uses the prefix **omg.org**.



4
The Naming Service
Copyright 2000–2001 IONA Technologies



19.4 Naming IDL Structure

The IDL for the Naming Service follows the general structure shown above. The entire IDL is part of the CosNaming module. Apart from type definitions, the module contains three interfaces:

- NamingContext

This interface provides most of the functionality of the Naming Service and allows you to create, remove, and lookup bindings.

- NamingContextExt

This interface was added in 1999 by a revision to the service known as the Interoperable Naming Service. The interface provides a few convenience functions that make it easier to deal with name bindings. The Interoperable Naming service was first published with CORBA 2.4.

- BindingIterator

This interface is provided so you can list the bindings of a naming context even if the context contains a very large number of bindings.

Name Representation

A name component is a *pair* of strings. A sequence of name components forms a pathname through a naming graph:

```
module CosNaming {
    typedef string Istring; // Historical hangover
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
    // ...
};
```

The **kind** field is meant to be used similarly to a file name extension (such as “filename.cc”).

For two name components to be considered equal, both **id** and **kind** must be equal.



19.5 Name Representation

Unlike a file name, which is a single string, the Naming Service uses a pair of strings as the name of a binding. (The diagram on page 19-4 only shows the `id` field and assumes that the `kind` field is empty.) The idea of using a pair of strings instead of a single string was inspired by file names, which often use an extension.¹

A sequence of name components (or pairs of strings) forms a pathname that can be used to navigate through a naming graph, such that each name component identifies one binding, and the sequence of name components identifies a path from a starting context via a number of intermediate contexts to a target binding.

Note that, unlike for file system names, a name component can be any string. In particular, you can freely use characters such as “/” or “.” as part of a name component, and even the empty pair of strings is a legal name component.

By definition, for two name components to be considered equal, both their `id` and `kind` fields must be equal. For two names to be equal, the names must have the same number of equal name components.

1. Unfortunately, this doesn’t achieve anything and only serves to make the Naming Service more complex to use than necessary. You can avoid most of the unnecessary complexity by simply leaving the `kind` field unused. (The C++ mapping initializes nested strings to the empty string, so you can effectively ignore the `kind` field.)

Stringified Names

Names are sequences of string pairs. We can show a name as a table:

Index	id	kind
0	Ireland	Country
1	Guinness	Brewery

This is a two-component name corresponding to the following graph:



The same name can be written as a string as:

Ireland.Country/Guinness.Brewery



6
The Naming Service
Copyright 2000–2001 IONA Technologies



19.6 Stringified Names

Names are sequences of structures, so we can show a name as a table that contains an `id` and `kind` value for each name component. The above slide shows the representation of a two-component name that uses both `id` and `kind` fields for each component.

The original Naming Services only provided the representation of names as sequences, which makes it difficult to deal with names as a single convenient unit. For example, if names are sequences of pairs, it is not clear how to display a name in a user interface, in particular if even the empty string is a legal value for `id` and `kind` fields. For this reason, the Interoperable Naming Service added a stringified representation for names, using “.” as the separator for `id` and `kind` fields, and “/” as the separator for name components. (We will discuss how stringified names deal with various special conditions, such as `id` and `kind` fields that are empty or contain a “.” in Section 19.18.)

Note that for two names to be equal, both `id` and `kind` fields must be equal. This means that the name components can all appear in the same parent context:

Guinness.Beer
Budweiser.Beer
Chair.Person
Chair.Furniture

Pathnames and Name Resolution

There is no such thing as an absolute pathname in a Naming Service.

All names must be interpreted relative to a starting context (because a Naming Service does not have a distinguished root context).

Name resolution works by successively resolving each name component, beginning with a starting context.

A name with components C_1 , C_2 , ..., C_n : is resolved as:

$$\text{cxt} \rightarrow \text{op}([c_1, c_2, \dots, c_n]) \equiv \text{cxt} \rightarrow \text{resolve}([c_1]) \rightarrow \text{op}([c_2, \dots, c_n])$$

This looks complex, but simply means that operation **op** is applied to the final component of a name after all the preceding components have been used to locate the final component.



19.7 Pathnames and Name Resolution

You must realize that the concept of an absolute pathname does not apply to the naming service. That is because the Naming Service (unlike a file system) does not have a distinguished root and interpretation of pathnames is always relative to some starting context on which you invoke an operation.

The process of name resolution is the same as for a file system: for a name with n components, the first $n-1$ components are used to navigate through the graph to the target context identified by the final component. Whatever operation you have specified then applies to the binding identified by the final component.

Obtaining an Initial Naming Context

You must obtain an initial naming context before you can do anything with the service.

The configured initial naming context is returned by

```
resolve_initial_references("NameService")
```

This returns an object reference to either a **NamingContext** or a **NamingContextExt** object. (For ORBacus, you always get a **NamingContextExt** interface.)

Exactly which context is returned depends on the ORB configuration.

You can override the default with the `-ORBInitRef` option:

```
./a.out -ORBInitRef NameService=<ior>
```



8
The Naming Service
Copyright 2000–2001 IONA Technologies



19.8 Obtaining an Initial Naming Context

Obviously, for clients and servers to be able to agree on advertised references, they must both agree to use the same Naming Service. Both clients and servers call `resolve_initial_references("NameService")` to obtain a reference to the initial naming context. For example:

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get initial naming context.
CORBA::Object_var obj
    = orb->resolve_initial_references("NameService");

// Narrow to NamingContext
CosNaming::NamingContext_var inc; // Initial naming context
inc = CosNaming::NamingContext::_narrow(obj);

// ...
```

If the Naming Service supports the CORBA 2.4 Interoperable Naming Service specification, the call will return a object of type `NamingContextExt` instead, so you can narrow that to `NamingContextExt`:

```

// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get initial naming context.
CORBA::Object_var obj
    = orb->resolve_initial_references("NameService");

// Narrow to NamingContextExt
CosNaming::NamingContextExt_var inc;    // Initial naming context
inc = CosNaming::NamingContextExt::_narrow(obj);

if (!CORBA::is_nil(inc)) {
    // It's an Interoperable Naming Service...
} else {
    // Doesn't support INS, must be the old service then...
}

// ...

```

Note that ORBacus fully supports the INS specification so, for ORBacus, the narrow to `NamingContextExt` will always succeed.

Exactly which instance of a naming context will be returned by `resolve_initial_references` depends on the ORB configuration (see Unit 18). However, even if the configuration is inappropriate for what you want to do, you can override the returned value from the command line by providing either a stringified IOR or a URL-style IOR (see Section 19.20) to the initial naming context:

```
$ ./myclient -ORBInitRef NameService=IOR:013a0d0...
```

This causes the call to `resolve_initial_references` to return the specified IOR when invoked with a service name of `NameService`. In general, you can use this mechanism to add arbitrary token-reference pairs to `resolve_initial_references`. For example:

```
$ ./myclient -ORBInitRef MyFavouriteService=IOR:013a0d0...
```

This returns the specified IOR when you call `resolve_initial_references("MyFavouriteService")`.

Naming Service Exceptions

The **NamingContext** interface defines a number of exceptions:

```
interface NamingContext {
    enum    NotFoundReason { missing_node, not_context, not_object };
    exception NotFound {
        NotFoundReason why;
        Name          rest_of_name;
    };
    exception CannotProceed {
        NamingContext cxt;
        Name          rest_of_name;
    };
    exception InvalidName {};
    exception AlreadyBound {};
    exception NotEmpty {};
    // ...
};
```



19.9 Naming Service Exceptions

NotFound Exception

This exception is raised by operations that require a name for lookup if the name does not resolve to an existing binding. The `NotFound` exception contains two data members.

- `why`

The `why` member provides more information as to why a lookup failed.

- `missing_node`

One of the components of a name specifies a binding that does not exist.

- `not_context`

One of the components of a name (other than the final component) specifies a binding to an application object instead of to a context.

- `not_object`

One of the components of a name specifies an object reference that dangles (points to a non-existent object).

- `rest_of_name`

The `rest_of_name` member contains the trailing part of the name that could not be resolved.

CannotProceed Exception

This exception indicates that the implementation has given up for some reason. Typically, this happens when a name binding denotes a context in a different Naming Service implemented in a

remote process, but that context could not be reached during name resolution (for example, because the network is down). The `CannotProceed` exception contains two data members.

- `cxt`
This is the object reference to the context containing the first unresolved binding.
- `rest_of_name`
This member contains the unresolved remainder of the name.

InvalidName Exception

This exception is raised if you attempt to resolve an empty name (a `Name` sequence with length zero, containing no components). If your Naming Service implementation restricts the permissible characters for name components (the ORBacus implementation does not), it raises this exception if you attempt to create a binding that contains an illegal character.

AlreadyBound Exception

This exception is raised if you attempt to create a binding that already exists. (Remember, name bindings must be unique within their parent context.)

NotEmpty Exception

This exception is raised if you attempt to destroy a context that still contains bindings. (As you will see on page 19-15, a context must be empty before you can destroy it.)

Creating and Destroying Contexts

NamingContext contains three operations to control the life cycle of contexts:

```
interface NamingContext {
    // ...
    NamingContext    new_context();
    NamingContext    bind_new_context(in Name n) raises(
        NotFound, CannotProceed,
        InvalidName, AlreadyBound
    );
    void             destroy() raises(NotEmpty);
    // ...
};
```



10
The Naming Service
Copyright 2000–2001 IONA Technologies



19.10 Creating and Destroying Contexts

Naming contexts act as a factory interface for other naming contexts and support a `destroy` operation.

NamingContext new_context()

This operation creates a new, empty naming context. Note that the operation does not accept an `in` parameter that could be used to give a name to the new context. This means that the new context is not bound into the naming graph by any name and therefore is orphaned. You can bind the new context into the graph later by calling the `bind` operation (see page 19-26).

The reason for providing a factory operation that creates orphaned contexts is that you may want to create a binding in one Naming Service that denotes a context in a different Naming Service (one that is implemented by a different process, possibly on a remote machine). To do this, you first create an orphaned context in one service and then add a binding to the second service in a separate step.

Because bindings are provided by object references, a single connected naming graph can span servers on different machines. Such distribution of a single logical service over multiple physical servers is known as *federation*.

NamingContext bind_new_context(in Name n) raises(NotFound, CannotProceed, InvalidName, AlreadyBound)

This factory operation creates a new context and binds the new context under the name `n` into the context on which `bind_new_context` was invoked. Typically, you will use this operation instead of `new_context` because it both creates and names a context in a single step.

`bind_new_context` is analogous to the UNIX `mkdir` command, which both creates and names a directory.

`bind_new_context` can raise some of the exceptions discussed in Section 19.9. For example, an `AlreadyBound` exception indicates that the binding passed to `bind_new_context` is already in use, and `NotFound` indicates that the name `n` could not be resolved to a target context on which to invoke the `bind_new_context` operation. For the remainder of this unit, we do not explicitly discuss the exceptions raised by operations. In all cases, they have the semantics explained in Section 19.9.

`void destroy() raises(NotEmpty)`

The `destroy` operation destroys a context. You can destroy a context only if it is empty (contains no bindings). The `destroy` operation, however, is *not* analogous to the UNIX `rmdir` command: `rmdir` both destroys a directory and removes its name from the parent directory. In contrast, `destroy` only destroys a context and does not remove any bindings to the destroyed context that may still exist in parent contexts. If you destroy a context that is bound into a parent context under some name, you must also invoke an `unbind` operation (see page 19-24) on the parent context; otherwise, you will leave a dangling binding behind. You will see source code examples of how to correctly destroy contexts in Section 19.15.

Creating Bindings

Two operations create bindings to application objects and to contexts:

```
interface NamingContext {
    // ...
    void bind(in Name n, in Object obj)
        raises(
            NotFound, CannotProceed,
            InvalidName, AlreadyBound
        );
    void bind_context(in Name n, in NamingContext nc)
        raises(
            NotFound, CannotProceed,
            InvalidName, AlreadyBound
        );
    // ...
};
```



11
The Naming Service
Copyright 2000–2001 IONA Technologies



19.11 Creating Bindings

```
void bind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound)
```

The `bind` operation adds the name `n` to the context on which `bind` is invoked. The new name denotes the passed reference `obj`. This is the operation you must use if you want to give a name to one of your objects. Note that you *can* bind a `nil` reference even though it is rather meaningless. We suggest that you not do this.

```
void bind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound)
```

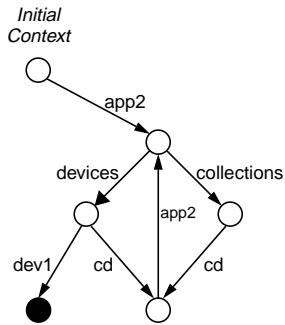
The `bind_context` operation works like `bind` but is used to bind contexts instead of normal application objects. The parameter `nc` has the type `NamingContext`, and that makes it impossible to pass something that is not a naming context. Attempts to bind a `nil` reference as a context raise a `BAD_PARAM` exception.

If you use `bind` (instead of `bind_context`) to bind a *context* object, the `bind` operation will work, but the binding will behave like an ordinary binding to an application object. If you incorrectly bind a context with `bind` instead of `bind_context`, the bound context will not participate in name resolution because as far as the Naming Service is concerned, the context will be treated like an application object.

Context Creation Example

To create a naming graph, you can use names that are all relative to the initial context or you can use names that are relative to each newly-created context.

The code examples that follow create the following graph:



12
The Naming Service
Copyright 2000–2001 IONA Technologies



19.12 Context Creation Example

In order to create a naming graph, you can either use operations that all use names that are relative to the initial naming context, or you can use single-component names that are relative to each newly-created context.

Here is an example that creates the above graph using names that relative to the initial naming context. The application object that is bound here under the name `app2/devices/dev1` is the CCS controller:

```
CosNaming::NamingContext_var inc = ...; // Get initial context

CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("app2"); // kind is empty

CosNaming::NamingContext_var app2;
app2 = inc->bind_new_context(name); // inc -> app2

name.length(2);
name[1].id = CORBA::string_dup("collections");
CosNaming::NamingContext_var collections;
collections = inc->bind_new_context(name); // app2 -> collections

name[1].id = CORBA::string_dup("devices");
CosNaming::NamingContext_var devices;
```

```
devices = inc->bind_new_context(name);    // app2 -> devices

name.length(3);
name[2].id = CORBA::string_dup("cd");
CosNaming::NamingContext_var cd;
cd = inc->bind_new_context(name);        // devices -> cd

name.length(4);
name[3].id = CORBA::string_dup("app2");
inc->bind_context(name, app2);          // cd -> app2

CCS::Controller_var ctrl = ...;
name.length(3);
name[2].id = CORBA::string_dup("dev1");
inc->bind(name, ctrl);                  // devices -> dev1

name[1].id = CORBA::string_dup("collections");
name[2].id = CORBA::string_dup("cd");
inc->bind_context(name, cd);           // collections -> cd
```

Note that (even though this is not advisable) it is legal to create loops in the graph.

Contrast this with the following example, which always uses a name with a single component to create the next part of the graph:

```
CosNaming::NamingContext_var inc = ...; // Get initial context

CosNaming::Name name; // Initialize name
name.length(1);
name[0].id = CORBA::string_dup("app2"); // kind is empty string

CosNaming::NamingContext_var app2;
app2 = inc->bind_new_context(name); // Create and bind

name[0].id = CORBA::string_dup("devices");
CosNaming::NamingContext_var devices;
devices = app2->bind_new_context(name); // Create and bind

name[0].id = CORBA::string_dup("collections");
CosNaming::NamingContext_var collections;
collections = app2->bind_new_context(name); // Create and bind

name[0].id = CORBA::string_dup("cd"); // Make cd context
CosNaming::NamingContext_var cd;
cd = devices->bind_new_context(name); // devices -> cd

collections->bind_context(name, cd); // collections -> cd

name[0].id = CORBA::string_dup("app2");
cd->bind_context(name, app2); // cd -> app2

CCS::Controller_var ctrl = ...; // Get controller ref

name[0].id = CORBA::string_dup("dev1");
devices->bind(name, ctrl); // Add controller
```

Which style of creation you use depends on your preferences and how much surrounding context is available at the point in your code where you want to create a binding. (As a rule of thumb, the less context information is needed by a bit of code, the easier that code becomes to modify and reuse.)

Rebinding

The **rebind** and **rebind_context** operations replace an existing binding:

```
interface NamingContext {
    // ...
    void    rebind(in Name n, in Object obj)
            raises(
                NotFound, CannotProceed, InvalidName
            );
    void    rebind_context(in Name n, in NamingContext nc)
            raises(
                NotFound, CannotProceed, InvalidName
            );
    // ...
};
```

Use **rebind_context** with caution because it may orphan contexts!



13
The Naming Service
Copyright 2000–2001 IONA Technologies



19.13 Rebinding

If you attempt to create a binding that already exists with `bind`, `bind_context`, or `bind_new_context`, you get an `AlreadyBound` exception. In contrast, `rebind` and `rebind_context` allow you to replace the object reference for an existing binding with a new one. (If no binding exists with the given name, it is created.) For example, the following code fragment does not raise an exception, no matter whether the specified binding already exists when the code first executes:

```
CORBA::Object_var obj = ...;           // Get an object
CosNaming::NamingContext_var cxt = ...; // Get a context...

CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("Some name");

cxt->rebind(name, obj);                 // Fine
cxt->rebind(name, obj);                 // Fine
```

Be careful when using `rebind_context`. If you replace an existing context with a new context under the same name, the previous context will be orphaned.

You cannot use `rebind` or `rebind_context` to change the type of a binding. For example, you cannot change a binding to a context to now denote an application object, or vice versa. Attempts to do so raise a `NotFound` exception with a `why` member of `not_context` and `not_object`, respectively.

Resolving Bindings

The **resolve** operation returns the reference stored in a binding:

```
interface NamingContext {
    // ...
    Object resolve(in Name n) raises(
        NotFound, CannotProceed, InvalidName
    );
    // ...
};
```

The returned reference is (necessarily) of type **Object**, so you must narrow it to the correct type before you can invoke operations on the reference.



14
The Naming Service
Copyright 2000–2001 IONA Technologies



19.14 Resolving Bindings

Naturally, the Naming Service is useless unless you can get information back out of it, specifically, unless you can resolve a binding to an object reference. The **resolve** operation returns the reference that is stored under a binding. For example, the following code retrieves the controller reference we advertised in Section 19.12:

```
CosNaming::NamingContext inc = ...; // Get initial context...

CosNaming::Name name;
name.length(3);
name[0].id = CORBA::string_dup("app2");
name[1].id = CORBA::string_dup("devices");
name[2].id = CORBA::string_dup("dev1");

CORBA::Object_var obj;
try {
    obj = inc->resolve(name);
} catch (const CosNaming::NamingContext::NotFound &) {
    // No such name, handle error...
    abort();
} catch (const CORBA::Exception & e) {
    // Something else went wrong...
    cerr << e << endl;
    abort();
}
```

```
    }

    if (CORBA::is_nil(obj)) {
        // Polite applications don't advertise nil references!
        cerr << "Nil reference for controller! << endl;
        abort();
    }

    CCS::Controller_var ctrl;
    try {
        ctrl = CCS::Controller::_narrow(obj);
    } catch (CORBA::SystemException & e) {
        // Can't figure it out right now...
        cerr << "Can't narrow reference: " << e << endl;
        abort();
    }

    if (CORBA::is_nil(ctrl)) {
        // Oops!
        cerr << "Someone advertised wrong type of object!" << endl;
        abort();
    }

    // Use ctrl reference...
```

This example includes more detailed error handling, by making sure that the advertised reference is not nil and that it is of the correct type.

Obviously, writing this code over and over again in an application is tedious, so you should encapsulate it in appropriate helper functions or classes. (See Henning & Vinoski, Section 18.14.1, for a template function that wraps `resolve`.)

Removing Bindings

You can remove a binding by calling **unbind**:

```
interface NamingContext {
    // ...
    void    unbind(in Name n) raises(
                NotFound, CannotProceed, InvalidName
            );
    // ...
};
```

unbind removes a binding whether it denotes a context or an application object.

Calling **unbind** on a context *will* create an orphaned context. To get rid of a context, you must both **destroy** and **unbind** it!



15
The Naming Service
Copyright 2000–2001 IONA Technologies



19.15 Removing Bindings

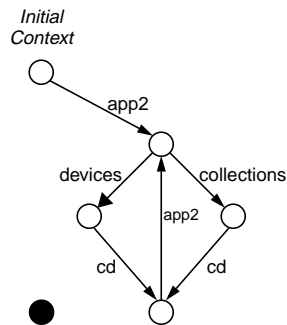
To get rid of a binding (whether to a context or to an application object), you call `unbind`. For example, the following code fragment removes the `dev1` binding from the graph shown on page 19-18, using a name relative to the initial naming context:

```
CosNaming::NamingContext_var inc = ...; // Get initial context

CosNaming::Name name;
name.length(3);
name[0].id = CORBA::string_dup("app2");
name[1].id = CORBA::string_dup("devices");
name[2].id = CORBA::string_dup("dev1");

inc->unbind(name);
```

Having unbound the application object from the graph, we end up with the following picture:



Note that the controller object still exists (but is no longer accessible via the Naming Service).

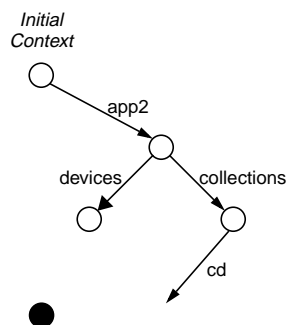
When you want to destroy a context, you must take care to both destroy the context and to remove the binding to that context in its parent context; otherwise, you will leave an orphaned context behind. In addition, you cannot destroy a context unless it is empty, so you have to make sure that all the context's bindings are removed first.

As an example of this, let's continue from the previous example and remove the `app2/devices/cd` context (the bottom-most context in the graph). To do this, we first remove the bindings labeled `app2` and `cd` and then destroy the context:

```
// ...
// Name is currently initialized "app2/devices/dev1".
// Change name to "app2/devices/cd/app2".
name.length(4);
name[2].id = CORBA::string_dup("cd");
name[3].id = CORBA::string_dup("app2");
inc->unbind(name); // Get rid of app2 link

name.length(3);
CosNaming::NamingContext_var tmp = inc->resolve(name);
tmp->destroy(); // Destroy cd context
name.length(2);
inc->unbind(name); // Remove binding in parent context
```

This creates the following graph:



Note that the code is incomplete because it leaves a dangling binding in the `collections` context!

Listing Name Bindings

```
// In module CosNaming:
enum BindingType { nobject, ncontext };

struct Binding {
    Name      binding_name;
    BindingType binding_type;
};
typedef sequence<Binding> BindingList;

interface BindingIterator; // Forward declaration
interface NamingContext {
    // ...
    void list(
        in unsigned long    how_many,
        out BindingList     bl,
        out BindingIterator it
    );
};
```



16
The Naming Service
Copyright 2000–2001 IONA Technologies



19.16 Listing Name Bindings

The `list` operation permits you read the list of bindings in a context. Because the number of bindings in a context may be very large (potentially larger than the amount of data that can be returned in a single call), `list` returns an iterator object if the number of bindings is larger than some (implementation-dependent) limit. We will go through the basics first and then examine how to use iterators.

Each binding is described by a `Binding` structure. The `binding_name` field provides the name of the binding and the `binding_type` field indicates whether the binding is to a context or an application object.

When you call the `list` operation, you must supply a value for the `how_many` parameter. This parameter limits the number of bindings that will be returned in the `bl` parameter; the `list` operation guarantees to return no more than `how_many` items (but may return fewer, either because the context does not have that many bindings, or because the service arbitrarily limits the number of bindings that will be returned in a single call). Once `list` returns, you can look at the length of the `bl` sequence to find out how many bindings were actually returned. If `bl` contains all bindings in the context, the `it` parameter is `nil`.

NOTE: `list` only returns the bindings for the context on which it is invoked, not the bindings that may be part of child contexts (that is, `list` behaves like the UNIX `ls` command, and not like `ls -R`). This means that the `binding_name` member of each of the returned `Binding` structures has a length of one. (Strictly speaking, the type of the `binding_name` member should be `NameComponent` instead of `Name`. However, we are stuck with this misfeature of the IDL.)

A call to `list` may not return all of the bindings in a context. This happens in the following cases:

- The context contains more bindings than the value of `how_many`. (It is legal to provide a value of zero for `how_many`.)
- The context contains fewer than `how_many` bindings, however, the number of bindings in the context is greater than some implementation-defined limit. (ORBacus does not enforce any limit.)

Either way, if not all bindings in the context are returned by the call to `list`, the `it` parameter is set to an iterator object of type `BindingIterator`:

```
interface BindingIterator {
    boolean next_n(
        in unsigned long    how_many,
        out BindingList     bl
    );
    boolean next_one(out Binding b);
    void    destroy();
};
```

This iterator object permits you to access the remaining bindings in a context (namely those that were not returned by the initial call to `list`).

boolean next_n(in unsigned long how_many, in out BindingList bl)

This operation returns the next `how_many` bindings in the parameter `bl`. As with `list`, there may be fewer sequence elements in `bl` than you requested with `how_many` because the operation may choose, for example, to never return more than some fixed number of bindings. (ORBacus does not enforce such a limit.) A value of zero for `how_many` raises a `BAD_PARAM` exception.

The return value from `next_n` tells you whether the `bl` parameter contains valid bindings. If this call to `next_n` returned bindings, the return value is `TRUE`. If this call to `next_n` returned no bindings, the return value is `FALSE`, and the value of `bl` is undefined.

boolean next_one(out Binding b)

This operation does the same thing as calling `next_n` with a `how_many` value of 1. You should avoid using it because it is rather inefficient: it incurs the cost of remote call for each binding.

void destroy()

The `destroy` operation permanently destroys the iterator. You can call `destroy` at any time even before you have retrieved all bindings from the context. However, you must call `destroy` even if you retrieve all bindings.

Interpreting a Binding List

The following example shows how you can correctly iterate over a naming context to list its contents. The code retrieves the bindings in lots of at most 100. The `show_chunk` helper function takes care of displaying each lot of bindings.

```
void
show_chunk(const CosNaming::BindingList & bl) // Helper function
{
    for (CORBA::ULong i = 0; i < bl.length(); ++i) {
        cout << bl[i].binding_name[0].id;
        if ( bl[i].binding_name[0].id[0] == '\\0'
            || bl[i].binding_name[0].kind[0] != '\\0') {
            cout << "." << bl[i].binding_name[0].kind;
        }
        if (bl[i].binding_type == CosNaming::ncontext)
            cout << ": context" << endl;
        else
            cout << ": reference" << endl;
    }
}

void
list_context(CosNaming::NamingContext_ptr nc)
{
    CosNaming::BindingIterator_var it; // Iterator reference
    CosNaming::BindingList_var bl; // Binding list
    const CORBA::ULong CHUNK = 100; // Chunk size

    nc->list(CHUNK, bl, it); // Get first chunk
    show_chunk(bl); // Print first chunk

    if (!CORBA::is_nil(it)) { // More bindings?
        while (it->next_n(CHUNK, bl)) // Get next chunk
            show_chunk(bl); // Print chunk
        it->destroy(); // Clean up
    }
}
```

Note that the code is careful to call `destroy` on the iterator (if one was returned). This is necessary to avoid an object leak in the Naming Service.

Pitfalls in the Naming Service

Here are a handful of rules you should adhere to when using the Naming Service:

- Do not advertise nil references.
- Do not advertise transient references.
- Stay clear of unusual characters for names, such as “.”, “/”, “*”, etc.
- Take care to destroy contexts correctly.
- Call destroy on iterators.
- Make the graph a single-rooted tree.



17
The Naming Service
Copyright 2000–2001 IONA Technologies



19.17 Pitfalls in the Naming Service

Sticking to a handful of rules when using the Naming Service will save you trouble in the long run.

Do not advertise nil references. Advertising a nil reference in the service is legal, but achieves nothing other than running the risk that a careless client might crash when it calls via nil reference that it has successfully resolved.

Do not advertise transient references. Advertising transient references makes no sense because they will become stale and raise either `TRANSIENT` or `OBJECT_NOT_EXISTS` if clients use them once the corresponding object has disappeared.

Avoid using unusual characters in names. If you use meta-characters that are non-printable or have special meaning to the shell, it becomes harder to use the `nsadmin` utility (see page 19-41) for maintenance.

Take care to destroy contexts correctly. Always call both `destroy` and `unbind` when you want to get rid of a context. (The order in which you make the calls does not matter.) That way, you avoid dangling bindings and orphaned contexts.

Call destroy on iterators. If you do not call `destroy`, the Naming Service does not know that an iterator is no longer wanted and keeps the iterator around until the service is restarted. This wastes memory.

Make the graph a single-rooted tree. Using a single-rooted tree structure avoids problems with ambiguous path names and make administration of the service easier.

Stringified Name Syntax

- A stringified name uses “/” and “.” to separate name components and **id** and **kind** fields:
a.b/c.d (id[0] = “a”, kind[0] = “b”, id[1] = “c”, kind[1] = “d”)
- A backslash (“\”) escapes the meaning of these characters:
a\.b\c\d.e (id = “a.b/c\d”, kind = “e”)
- A name without a trailing “.” denotes an empty **kind** field:
hello (id = “hello”, kind = “”)
- A name with a leading “.” indicates an empty **id** field:
.world (id = “”, kind = “world”)
- A single “.” denotes a name with empty id and kind fields:
. (id = “”, kind = “”)



18
 The Naming Service
 Copyright 2000–2001 IONA Technologies



19.18 Stringified Name Syntax

The Interoperable Naming Service defines a stringified representation for names, using “/” and “.” as the separator characters for name components and **id** and **kind** fields, respectively. A few special rules cover how to deal with empty name components, and how to escape these meta-characters:

- You can embed a literal “/”, “.”, or “\” in a name component by escaping it with a backslash. (The only legal escape sequences are therefore “\”, “\.”, and “\”; a backslash that appears preceding any other character is a syntax error.)
- If a name component has an empty **kind** field, the “.” separator is omitted. A name component with a trailing unescaped “.” is illegal.
- If a name component has an empty **id** field, the name component begins with a leading “.”.
- The only legal representation of a name component in which both **id** and **kind** are empty is a single “.”.
- A stringified name with a leading or trailing unescaped “/” is illegal.

These rules ensure that each name has exactly one legal representation as a string. The advantage of this is that you can safely compare two stringified names for equality because the string representation of names is unambiguous.

Using Stringified Names

The `NamingContextExt` interface provides convenience operations for using stringified names:

```
interface NamingContextExt : NamingContext {
    typedef string StringName;

    StringName to_string(in Name n) raises(InvalidName);
    Name       to_name(in StringName sn) raises(InvalidName);

    Object     resolve_str(in StringName sn) raises(
        NotFound, CannotProceed, InvalidName
    );
    // ...
};
```

`to_string` and `to_name` are like C++ static helper functions. They do the same thing no matter on what context you invoke them.



19
The Naming Service
Copyright 2000–2001 IONA Technologies



19.19 Using Stringified Names

The `NamingContextExt` interface provides helper functions that make life with names a bit easier. (Note that `NamingContextExt` is derived from `NamingContext`, so old (pre-INS) clients can continue to work with INS implementations of the service.)

StringName to_string(in Name n) raises(InvalidName)
Name to_name(in StringName sn) raises(InvalidName)

These operations convert between names and stringified names. Note that they are merely convenience operations that do the same thing no matter on what context you invoke them (like C++ static member functions).

Object resolve_str(in StringName sn)
raises(NotFound, CannotProceed, InvalidName)

This operation works exactly like `NamingContext::resolve`, but accepts a stringified name instead. Note that `NamingContextExt` provides this convenience only for name resolution but not for other operations (such as `bind`, `rebind`, and so on). The reason for this is that resolving a binding is by far the most frequently used operation; adding versions of the other operations that operate on stringified names to `NamingContextExt` would not have been worth the resulting interface bloat.

URL-Style IORs

The specification defines two alternative styles of object references:

- **corbaloc**

An IOR that denotes an object at a specific location with a specific object key, for example:

```
corbaloc::bobo.acme.com/obj17359
```

- **corbaname**

An IOR that denotes a reference that is advertised in a naming service, for example:

```
corbaname::bobo.acme.com/NameService#CCS/controller
```

URL-style IORs are useful for bootstrapping and configuration.

Do *not* use them as a general replacement for normal IORs!



20
The Naming Service
Copyright 2000–2001 IONA Technologies



19.20 URL-Style IORs

The specification defines two URL-style notations for IORs, the `corbaloc` format and the `corbaname` format. This makes it possible to easily create an IOR for configuration and bootstrapping purposes without a need to copy complex stringified IORs around that are usually several hundred bytes in length.

Before we explore these IORs further, we must make it clear that URL-style IORs are *not* a general-purpose replacement for normal IORs. In particular, URL-style IORs offer no way to encode information that is usually part of ordinary IORs, such as quality-of-service parameters, transaction policies, codeset information, or security attributes. This means that URL-style IORs cannot be used to invoke operations on objects that require such additional information. Use URL-style IORs exclusively for configuration and bootstrapping!

URL-Style IORs (cont.)

A `corbaloc` IOR encodes a host, a port, and an object key:

```
corbaloc::myhost.myorg.com:3728/some_object_key
```

The port number is optional and defaults to 2809:

```
corbaloc::myhost.myorg.com/some_object_key
```

Dotted-decimal addresses are legal:

```
corbaloc::123.123.123.123/some_object_key
```

You can specify a protocol and version. (The default is `iiop` and `1.0`):

```
corbaloc:iiop:1.1@myhost.myorg.com:3728/some_object_key
```

Multiple addresses are legal:

```
corbaloc::hostA:372, :hostB, :hostC:3728/some_object_key
```



21
The Naming Service
Copyright 2000–2001 IONA Technologies



19.20.1 `corbaloc` IORs

A `corbaloc` IOR specifies a transport end-point and an object key. A few examples of `corbaloc` IORs are shown above. You must specify a host name and, following a “/” character, an object key. The syntax allows you to use fully-qualified domain names, unqualified host names, or dotted-decimal IP addresses. You can also specify a port number; if omitted, the port number defaults to 2809.

A more interesting feature is that a `corbaloc` URL can contain more than one address, as shown in the final example above. This feature permits you to add some amount of redundancy to an IOR. For the above example, the URL specifies that an object with key “some_object_key” can be found on `hostA` at port 372, on `hostB` at port 2809, or on `hostC` at port 3728. When binding such a URL, the ORB tries to use each address until it finds one at which the corresponding request succeeds; a binding failure is reported to clients only if all of the addresses in the IOR fail.

NOTE: The complete syntax for `corbaloc` is extensible to protocols other than IIOP, and you can even embed addressing information for multiple different protocols in the same IOR. You can consult the CORBA specification for details on these features.

URL-Style IORs (cont.)

A **corbaname** IOR is like a **corbaloc** IOR with an appended stringified name:

```
corbaname::myhost:5000/NameService#controller
```

This URL denotes a naming context on **myhost** at port 5000 with object key **NameService**. That context, under the stringified name **controller**, contains the object denoted by the URL.

Complex names are possible:

```
corbaname::myhost:5000/ns#Ireland.Country/Guinness.Brewery
```

In this example, the naming context with key **ns** must contain a context named **Ireland.Country** containing a binding **Guinness.Brewery** that denotes the target object.



22
The Naming Service
Copyright 2000–2001 IONA Technologies



19.20.2 corbaname IORs

A **corbaname** IOR provides a convenient way to denote an object that is advertised in a Naming Service. The syntax is identical to that for **corbaloc** IORs, except that the stringified name to be resolved is appended following a “#” separator.

URL Escape Sequences

ASCII alphabetic and numeric characters can appear in URL-style IOR without escaping them. The following characters can also appear without escapes:

“;”, “/”, “:”, “?”, “@”, “&”, “=”, “+”, “\$”,
 “ ”, “-”, “_”, “.”, “!”, “~”, “*”, “#”, “(”, “)”

All other characters must be represented in escaped form. For example:

Stringified Name	Escaped Form
<a>.b/c.d	%3ca%3e.b/c.d
a.b/ c.d	a.b/%20%20c.d
a%b/c%d	a%25b/c%25d
a\\b/c.d	a%5c%5cb/c.d

A % is always followed by two hex digits that encode the byte value (in ISO Latin-1) of the corresponding character.



23
 The Naming Service
 Copyright 2000–2001 IONA Technologies



19.21 URL Escape Sequences

The IETF RFC 2396 specification requires certain characters to be escaped if they appear in a URL. Because the OMG is bound by this syntax specification, URL-style IORs are subject to the rules laid out in RFC 2396. Fortunately, simple alphanumeric names and names that include hyphens or underscores need no escape sequences. (We suggest that you limit yourself to such simple names because many of the other legal characters, even though they need not be escaped, have special meaning to the shell and can be difficult to deal with unless you add yet another layer of quoting.)

Resolving URL-Style IORs

You can pass a URL-style IOR directly to `string_to_object`:

```
CORBA::Object obj
    = orb->string_to_object("corbaname::localhost/nc#myname");
```

The ORB resolves the reference like any other stringified IOR, including the required **resolve** invocation on the target naming context.

This is useful particularly for configuration:

```
./myclient -ORBInitRef \
NameService=corbaloc::localhost/NameService
```



24
The Naming Service
Copyright 2000–2001 IONA Technologies



19.22 Resolving URL-Style IORs

One useful aspect of URL-style IORs is that you can convert them back to object references with `ORB::string_to_object`. If you pass a corbaname URL to `string_to_object`, the ORB transparently converts it into the target object reference (which includes the invoking the `resolve` operation on the appropriate naming context).

Creating URL-Style IORs

Apart from simply writing them down, you can create a URL-style IOR using a **NamingContextExt** object:

```
interface NamingContextExt : NamingContext {
    // ...
    typedef string StringName;
    typedef string Address;
    typedef string URLString;

    exception InvalidAddress;

    URLString to_url(in Address addrkey, in StringName sn)
        raises(InvalidAddress, InvalidName);
};
```

addrkey must be an address, optional port, and object key in **corbaloc** syntax.



25
The Naming Service
Copyright 2000–2001 IONA Technologies



19.23 Creating URL-Style IORs

URL-style IORs are simple enough to write out by hand but you can also create them programmatically.

The `to_url` operation on `NamingContextExt` is a helper function whose behavior is independent of the specific context you invoke it on. The `addrkey` parameter must be in valid `corbaloc` syntax and indicates the host, port, and object key. For example:

```
CosNaming::NamingContextExt_var nc = ...;
CORBA::String_var url =
    nc->to_url(":localhost:5789/abc", "CCS/controller");
cout << url << endl;
```

This prints

```
corbaname::localhost:5789/abc#CCS/controller
```

`to_url` automatically deals with escape sequences that may be required to legally represent a stringified name as part of a URL. For example:

```
url = nc->to_url(":bobo.ooc.com.au/nc", "a\\b%/c.d");
```

This results in the URL:

```
corbaname::bobo.ooc.com.au/nc#a%5c%5cb%25/c.d
```

What and When to Advertise

You should advertise key objects in the Naming Service, such as the CCS controller. (Such objects are public integration points.)

Bootstrap objects are normally added to the service at installation time.

Provide a way to recreate key bindings with a tool.

You can advertise all persistent objects you create. If you do, tie the updates to the Naming Service to the life cycle operations for your objects.

Decide on a strategy of what to do when the Naming Service is unavailable (deny service or live with the inconsistency).



26
The Naming Service
Copyright 2000–2001 IONA Technologies

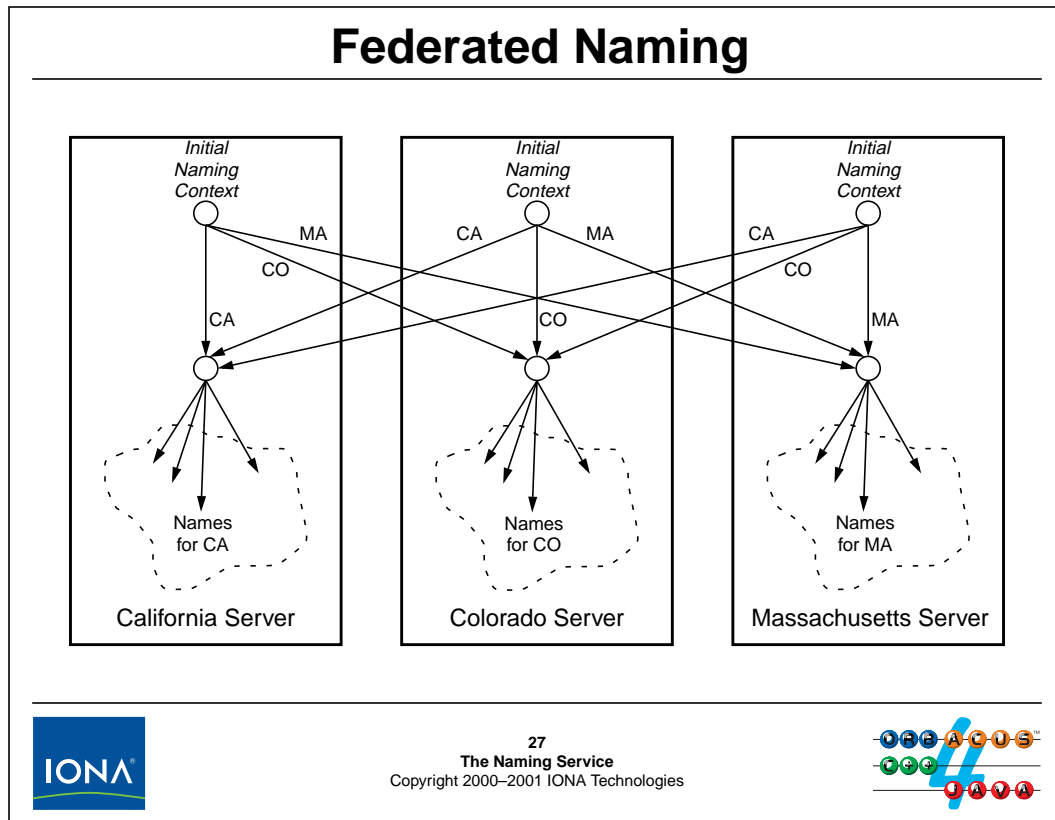


19.24 What to Advertise

What objects you should advertise in the Naming Service depends on your application. For the CCS system, the controller references is clearly the key public integration point, so you should advertise that. Typically, you can add bindings to the Naming Service for such singleton objects at installation time. It is a good idea to provide some means to recover lost bindings, either with a separate tool, or you can write your servers such that they re-advertise key objects every time they start up. That way, if the Naming Service database is lost or corrupted, customers can easily recreate the bindings required by your clients.)

If you decide to advertise other objects as well (such as thermometers and thermostats), you should tie updates to the Naming Service to the life cycle operations for those objects: add new bindings as part of the factory operation and remove bindings as part of the `destroy` operation.

If you add objects to the Naming Service that support life cycle operations, you must decide what to do if the Naming Service is down or unreachable when a binding must be created or removed. You can either fail the life cycle operation (that is, deny life cycle service), or you can create or remove the object anyway and live with the resulting inconsistency. If you permit the inconsistency to arise, you should also have a tool that can be run to restore consistency, to make sure that there are no dangling and no missing bindings for your application. However, it is usually simpler to deny service and to put the effort into ensuring that the Naming Service is healthy, rather than to try and create fancy recovery schemes.



19.25 Federated Naming

Because each naming context is identified by an object reference, it is trivial to link different Naming Services into a federation, by creating a binding in one service that points at a context in another service. You can exploit this to create naming hierarchies that use the same name no matter which initial naming context is used by clients. In the above example, we have three Naming Services, each in a different state. By cross-connecting the services as shown, a client can use the initial naming context for any one of the three servers because the pathnames are the same for all objects regardless of the starting context.

Arranging Naming Services along geographic or administrative boundaries and federating them has the added advantage that local lookups can be resolved locally, which works well if clients are more likely to be interested in local objects than in remote objects.

Note that fully connected structures such as the one shown above do not scale well beyond a handful of servers because the number of links at the top level grows as $O(n^2)$. For larger numbers of servers, you usually have to change to a hierarchical federation structure. (See Henning & Vinoski for details.)

Running the Naming Service

ORBacus Names is provided as the `nameserv` executable. Common options (use `nameserv -h` for a list):

- **-i**
Print initial naming context IOR on `stdout`
- **-d database_file**
Specifies database file for the service. (Without `-d`, the service is not persistent and uses an in-memory database.)
- **-s**
(Re)initializes the database. Must be used with `-d` option.

The object key of the initial naming context is **NameService**.



28
The Naming Service
Copyright 2000–2001 IONA Technologies



19.26 Running the Naming Service

The executable image for ORBacus Names is in the ORBacus `bin` directory, with the name `nameserv`. By default, the service runs in non-persistent mode, using an in-memory database. (Of course, this means that everything you have added to the service is forgotten as soon as it shuts down.) If you add the `-d` option, you can specify the name of a database file to make the naming graph persistent. (The first time you use a new database, you must also specify the `-s` option, which initializes (or re-initializes) the database file.)

It is convenient to register the Naming Service with the IMR for automatic start-up (see Unit 22). That way, not only does the service start up automatically when it is needed, but it also helps to ensure that consistent and correct command-line options are used. If you want to run a persistent Naming Service that does not use the IMR, you must use the `-OApport` option to ensure that the Naming Service uses the same port every time it starts.

The nsadmin Tool

The **nsadmin** tool provides a way to manipulate the Naming Service from the command line. Common options (use **nsadmin -h** for a list):

- **-b name IOR**
Bind **IOR** under the name **name**.
- **-c name**
Create and bind a new context under the name **name**.
- **-l [name]**
List the contents of the context **name**. (Initial context, by default.)
- **-r name**
Print the IOR for the binding identified by **name**.

IORs can be in normal or URL-style syntax.



29
The Naming Service
Copyright 2000–2001 IONA Technologies



19.27 The nsadmin Tool

The **nsadmin** tool allows you to manipulate a Naming Service from the command line. This is useful for configuration and installation, or if you need to debug an application. (Note that you can use URL-style IORs as well as normal ones when you create a binding.)

For example, you can create a context called **CCS** underneath the initial naming context and advertise a controller references (assumed to be in the file `ctrl.ref`) as follows:

```
nsadmin -c CCS
nsadmin -b CCS/controller `cat ctrl.ref`
```

Using the command

```
nsadmin -r CCS/controller
```

prints the controller reference on standard output.

Compiling and Linking

The IDL for ORBacus Names is installed in the ORBacus directory as `idl/OB/CosNaming.idl`.

The header files for the service are in `include/OB/CosNaming.h` and `include/OB/CosNaming_skel.h`.

The stubs and skeletons for ORBacus Names are pre-compiled and installed in `lib/libCosNaming.sl`.

To compile a client or server that uses ORBacus Names, compile with `-I /opt/OB4/include` and link with `-L /opt/OB4/lib -lCosNaming`.



30
The Naming Service
Copyright 2000–2001 IONA Technologies



19.28 Compiling and Linking

The IDL files for the Naming Service are installed in `idl/OB/CosNaming.idl`. You will need the IDL file if you want to compile stubs or skeletons for the service, or if you want to reuse some of the Naming Service IDL definitions for your own IDL (in which case you must `#include` the file).

The header files for the service are installed in `include/OB/CosNaming.h` and `include/OB/CosNaming_skel.h`, and a library containing the stubs and skeletons is provided in `lib/libCosNaming.sl`, so you do not have to compile the IDL or the stubs and skeletons in order to use the service.

20. Exercise: Using the Naming Service

Summary

In this unit, you will modify a server and client to rendezvous with each other using the Naming Service instead of using a stringified IOR.

Objectives

By the completion of this unit, you will know how to configure and start the Naming Service, how to use the administration tool for the service, and how to write clients and servers that use the Naming Service to exchange IORs.

20.1 Source Files and Build Environment

You will find this exercise in your `naming` directory. The files in this directory are the same as for Unit 16.

20.2 Server Operation

The server source code provided to you for this exercise implements the solution presented in Unit 17. The purpose of this exercise is to modify a client and server to use the Naming Service.

On start-up, the server advertises a reference to the controller in the Naming Service instead of writing it to a file. The client retrieves the reference from the Naming Service in order to locate the controller.

20.3 What You Need to Do

Step 1

Create a configuration file that sets a property to contain the reference returned by `resolve_initial_references("NameService")`. Set an environment variable or registry key to point at this configuration file.

Step 2

Start the Naming Service such that it runs at the host and port established in step 1.

Step 3

Use the `nsadmin` command to create a context under the initial naming context. Name the context using your login ID, such as `student_1`. Verify with `nsadmin` that the context was created correctly.

Step 4

The `run` function no longer writes a stringified reference to a file. Instead, it should advertise the controller reference in the Naming Service, in the context you created in step 3, under the name `controller`. Add code to `run` to do this.

Step 5

The `main` program in the client lacks the code to retrieve the controller reference from the Naming Service. Add code to `main` to do this.

21. Solution: Using the Naming Service

21.1 Solution

Step 1

```
$ cat ob.config
ooc.orb.service.NameService=corbaloc::janus.ooc.com.au:5000/NameService
$ ORBACUS_CONFIG=`pwd`/ob.config; export ORBACUS_CONFIG
```

Step 2

```
$ /opt/OB4/bin/nameserv -OApport 5000
```

Step 3

```
$ nsadmin -c student_1
$ nsadmin -l
Found 1 binding:
student_1 [context]
```

Step 4

```
// Get Naming Service reference
obj = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var inc
    = CosNaming::NamingContext::_narrow(obj);
if (CORBA::is_nil(inc))
    throw "Cannot find initial naming context!";

// Advertise the controller reference in the naming service.
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("student_1");
name[1].id = CORBA::string_dup("controller");
obj = ctrl_servant->_this();
inc->rebind(name, obj);
```

Step 5

```
// Get controller reference from Naming Service
CORBA::Object_var obj
    = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var inc
    = CosNaming::NamingContext::_narrow(obj);
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("student_1");
name[1].id = CORBA::string_dup("controller");
obj = inc->resolve(name);

// Try to narrow to CCS::Controller.
```

```
CCS::Controller_var ctrl;
try {
    ctrl = CCS::Controller::_narrow(obj);
} catch (const CORBA::SystemException &se) {
    cerr << "Cannot narrow controller reference: " << se << endl;
    throw 0;
}
if (CORBA::is_nil(ctrl)) {
    cerr << "Wrong type for controller ref." << endl;
    throw 0;
}
```

22. The Implementation Repository (IMR)

Summary

This unit explains the difference between direct and indirect binding and shows how an implementation repository may be used to both enable server migration and to achieve automatic server activation on demand. The unit also covers performance and reliability trade-offs for implementation repositories and shows how to configure the IMR for various deployment scenarios.

Objectives

By the completion of this unit, you will know how to configure and use an IMR effectively. You will also understand the environment in which processes are started by the IMR and how to write your server to work within this environment.

Purpose of an Implementation Repository

An implementation repository (IMR) has three functions:

- It maintains a registry of known servers.
- It records which server is currently running on what machine, together with the port numbers it uses for each POA.
- It starts servers on demand if they are registered for automatic activation.

The main advantage of an IMR is that servers that create persistent references

- need not run on a fixed machine and a fixed port number
- need not be running permanently



1
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.1 Purpose of an Implementation Repository

The main motivation for providing an IMR is to permit servers to move from port to port or machine to machine without breaking existing references that are held by clients. Fundamentally, an IMR has three functions:

- The IMR maintains a registry of known servers.

A server that wants to take advantage of IMR functionality must be known to the IMR. This is achieved by explicitly registering the server with the IMR when the server is deployed.

- The IMR records which server is currently running on what machine, together with the port numbers it uses for each POA.

This function of the IMR allows servers to change location (machine or port) without clients being aware of this happening.

- The IMR starts servers on demand if they are registered for automatic activation.

This function of the IMR permits you to have servers that correctly respond to client requests without the need to run them permanently. This is useful particularly for large systems with many servers, some of which may not be needed all the time. (Idle servers still consume system resources and so incur a cost. In addition, the feature permits automatic recovery if a server crashes because the IMR will transparently restart it.)

Binding

There are two methods of binding object references:

- Direct Binding (for persistent and transient references)

References carry the host name and port number of the server. This works, but you cannot move the server around without breaking existing references.

- Indirect Binding (for persistent references)

References carry the host name and port number of an Implementation Repository (IMR). Clients connect to the IMR first and then get a reference to the actual object in the server. This allows servers to move around without breaking existing references.

IMRs are proprietary for servers (but interoperate with all clients).



2
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.2 Binding

When a client receives an object reference from somewhere, the client-side run time must establish a connection to the server that hosts the target object eventually. The process of associating an object reference with a transport end point is known as binding. Binding can either be direct or indirect:

- For direct binding, a server embeds its own host name (or IP address) and port number into each reference. When clients connect, they therefore connect directly to the server. This is simple and efficient, but has drawbacks:
 - You must assign a distinct port number to each server and set that port number consistently on every execution of the server, using the `-OApport` option (or using a server property—see Unit 18). For installations with a large number of servers, the administration of port numbers can get cumbersome.
 - Once a server has handed out persistent object references to its clients, you can no longer move the server to a different port or different machine, at least not if you want to avoid breaking the references that are held by clients. Because clients may store their references in files or a service such as the Naming Service (see Unit 19), you have no way of knowing whether it is safe to move a server.

These problems can be avoided by indirect binding.

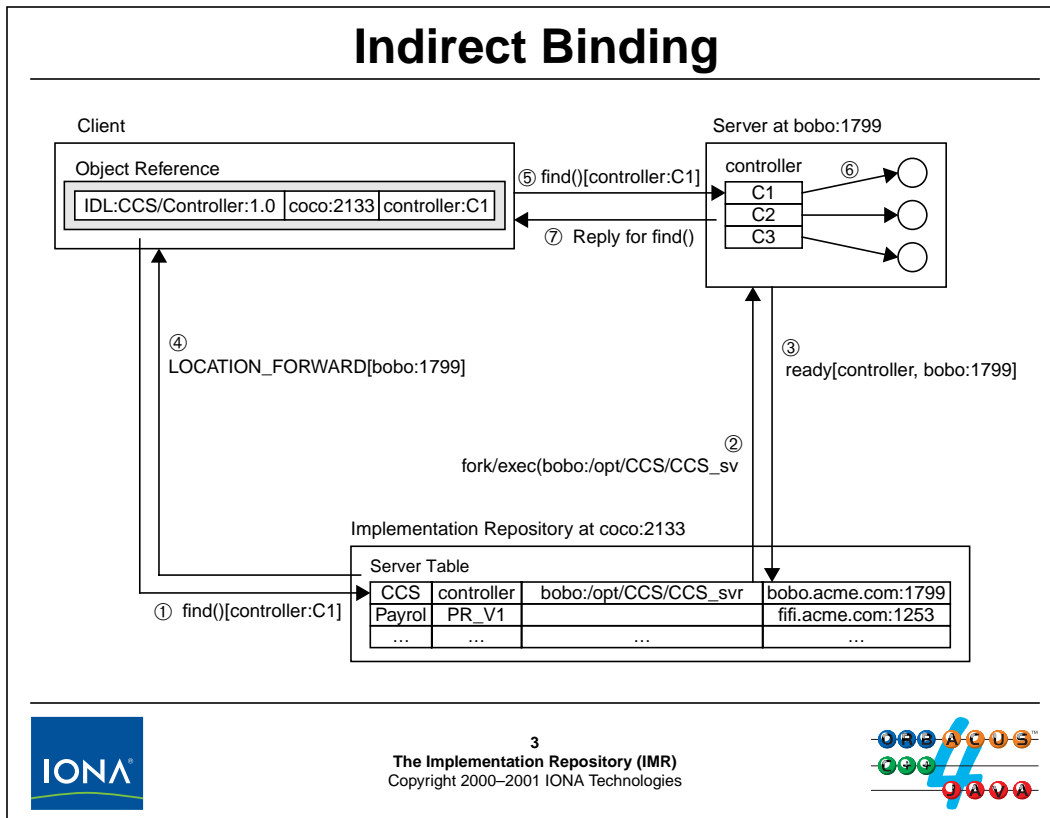
- Indirect binding requires a daemon process known as the Implementation Repository (IMR) to permanently run at a fixed location (host and port number). When a server creates a persistent reference, it embeds the host name and port number of the IMR in the reference, instead of its own addressing information. When clients bind a reference, they first connect to

the IMR and send their first request. On receipt of the request, the IMR works out where the server is currently running and returns a new reference to the client-side run time in a special location-forward reply. The client ORB then opens another connection using the new reference returned by the IMR, which contains the actual address details of the server. Provided the server is actually running at the location that was forwarded by the IMR, this second attempt succeeds and the client request ends up at the correct destination.

The extra level of indirection provided by the IMR allows you to move servers around over time without breaking existing references. Servers, when they start up, inform the IMR of where they currently are, so the IMR always knows about the current location of each server. In effect, the IMR provides a fixed point for object addressing in order to permit servers to change their addresses.

Implementation repositories are not standardized by the OMG, so their implementation is completely proprietary. In particular, servers can only use an implementation repository from the same ORB vendor. For example, an ORBacus server requires an ORBacus IMR and will not work with any other vendor's IMR; similarly, no other vendor's server can use an ORBacus IMR.

However, the interactions between clients and IMRs are completely standardized and only rely on IIOP. This means that a client using any vendor's ORB can interact with any other vendor's IMR.



22.3 Indirect Binding

The above diagram illustrates the sequence of interactions for indirect binding of a reference to the controller object. The diagram assumes that the implementation repository runs on machine `coco` at port 2133 and that the CCS server is not running when the client invokes the request. The sequence of steps during binding is as follows.

1. The client invokes the `find` operation on the controller. This results in the client-side run time opening a connection to the address found in the controller IOR, which is the address of the repository. With the request, the client sends the object key (which contains the POA name and the object ID—`controller` and `C1` in this example).
2. The IMR uses the POA name (`Controller`) to index into its server table and finds that the server is not running. Because the server is registered for automatic start-up, the IMR executes the command to start the server.
3. The server sends messages that inform the repository of its machine name (`bobo`), the names of the POAs it has created and their port numbers (`Controller` at 1799), and the fact that it is ready to accept requests.
4. The implementation repository constructs a new object reference that contains host `bobo`, port number 1799, and the original object key and returns it in a `LOCATION_FORWARD` reply to the client.
5. The client opens a connection to `bobo` at port 1799 and sends the request a second time.
6. The server uses the POA name (`Controller`) to locate the POA that contains the servant for the request, and uses the object ID (`C1`) to identify the target servant. The request is given to the servant for processing.

7. The servant completes the `find` operation and returns its results, which are marshaled back to the client in a `Reply` message.

As you can see, indirect binding uses the implementation repository as a location broker that returns a new IOR to the client that points at the current server location. The CORBA specification does not limit indirection to a single level. Instead, it requires a client to always respond to a `LOCATION_FORWARD` reply by attempting to send another request.

Automatic Server Start-Up

The IMR can optionally start server processes.

Two modes of operation are supported by the IMR:

- shared

All requests from all clients are directed to the same server. The server is started on demand.

- persistent

Same as the shared mode, but the server is started whenever the IMR starts and kept running permanently.

Servers are started by an Object Activation Daemon (OAD).

A single repository can have multiple OADs. An OAD must be running on each machine on which you want to start servers.



1
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies

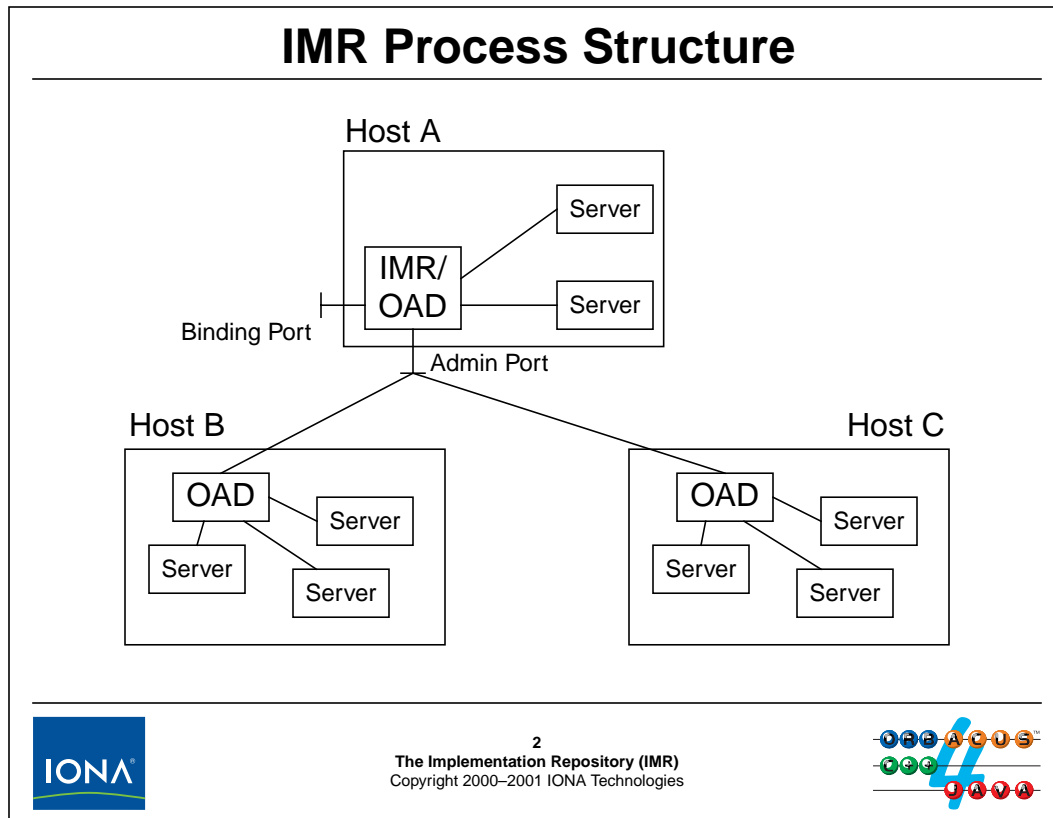


22.4 Automatic Server Start-Up

The IMR maintains knowledge of where each server is currently running in order to be able to forward that information to clients during indirect binding. This means that it is easy to extend the IMR to not only forward the location of a running server to clients, but to also start up servers on demand when a client request arrives for a server that is not currently running.

The ORBacus IMR offers two modes of automatic server activation:

- In shared mode, a single copy of the server is started when the first request for that server arrives. Thereafter, all requests from all clients are directed to that server instance.
- In persistent mode, a single server instance is started by the IMR as soon as the IMR itself is started. Thereafter, the IMR monitors the server process and restarts it when it goes down. All requests from all clients are sent to that same server instance.



22.5 IMR Process Structure

Each server that uses indirect binding is configured to use a specific OAD. The OAD monitors the state of the server process and is informed by the server of state changes, such as POA creation and destruction. A server cannot use more than one OAD.

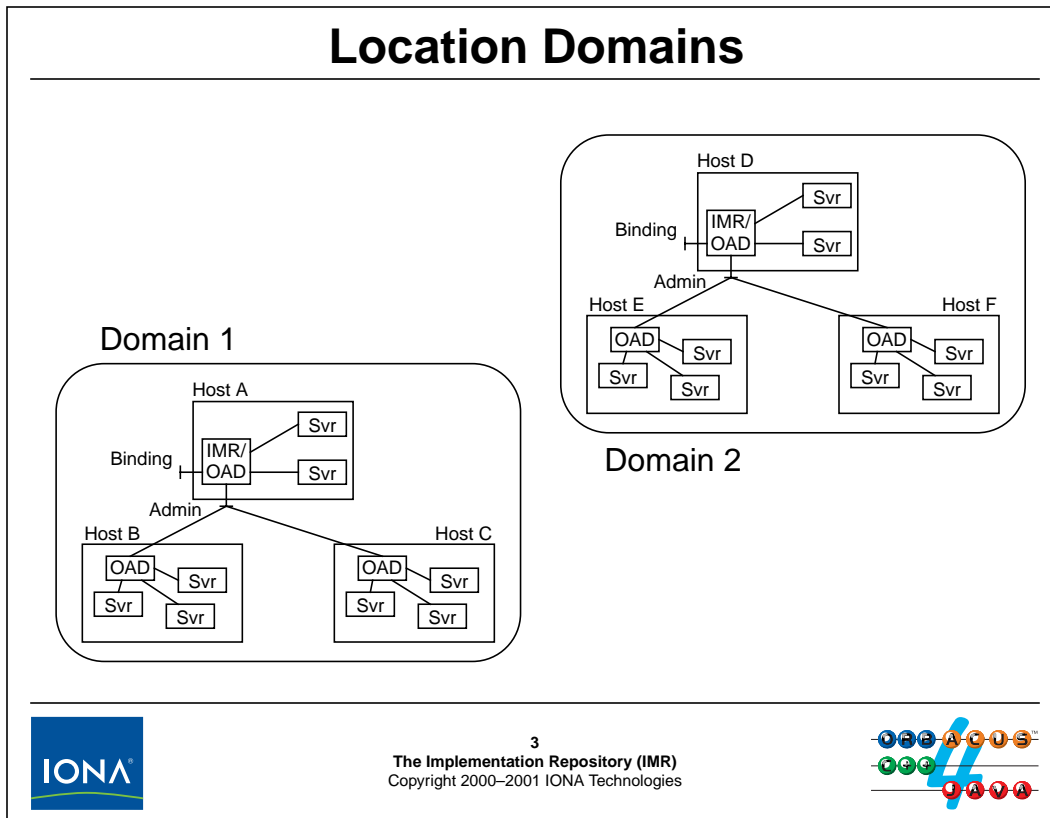
Each OAD, in turn, is configured to use a specific IMR. The IMR monitors the state its OADs; in turn, each OAD passes state changes to its IMR, such as start-up of a new server process.

The IMR uses the information it receives from its OADs (such as the host name and port number for each of the server's POAs) to resolve requests that arrive from clients. These requests arrive via the binding port of the IMR; the IMR replies with location forward replies which direct the clients to the correct server end point. The communication between the IMR and the OADs is carried out on a separate port, so administrative access to the IMR can be secured separately from the port on which clients bind requests.

Usually, the IMR on a host will also activate servers on that same host. In this case, the IMR can run in dual mode, in which a single process combines the functions of the IMR and OAD. You can also run the IMR in master mode (without the OAD functionality) if you want to activate servers only on remote hosts. If you run the IMR in slave mode, it acts as an OAD only.

Server activation on each host is carried out by the OAD on that host: the OAD creates a new server process when instructed by the IMR.

IMRs and OADs maintain state information in a small database. This information is used mainly for error recovery. For example, if the IMR machine goes down, the IMR uses the state in its database to update its knowledge of which servers are running where.



22.6 Location Domains

Each IMR defines a location domain. All the servers that are configured to use the same IMR are part of the IMR's location domain. How you choose your location domains has influence on the reliability and performance of a system, as well as on server migration:

- You can choose to run one IMR and OAD on each machine so each machine forms its own location domain.

The advantage of this approach is that the communication among the servers, the OAD, and the IMR is very fast because it happens via the back plane. In addition, you get high reliability with such a configuration because if one machine dies, only servers on that machine will be affected. All other servers continue to work normally.

The down side of this approach is that you cannot move a server to another machine without breaking existing references that are held by clients.

- You can place several machines into a location domain.

The advantage of this approach is that you can freely move servers among the machines in the location domain without invalidating references held by clients (because the IMR for the server remains the same when the server is moved).

The down side of this approach is that the communication overhead is a little larger. This is rarely a concern because binding of a new reference via the IMR happens only once, when the reference is first used by a client. However, if the machine running the IMR dies, none of the servers in the location domain are reachable by newly connecting clients until the IMR machine is available again.

The imradadmin Tool

imradadmin allows you to register servers with the IMR. General syntax:

```
imradadmin <command> [ <arg>... ]
```

You must register each server under a server name with the IMR. The server name must be unique for that IMR:

```
imradadmin --add-server CCS_server /bin/CCSserver
```

When the IMR starts the server, it automatically passes the server name in the **-ORBserver_name** option. (So the server knows that it should contact the IMR.)

If you want to manually start a server that is registered with the IMR, you must add the **-ORBserver_name** option when you start the server:

```
/bin/CCSserver -ORBserver_name CCS_server
```

Servers automatically register their persistent POAs with the IMR.



4
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.7 The imradadmin Tool

You must register a server with the IMR under a unique name with **imradadmin --add-server**. This creates an entry in the IMR's server table with the supplied command line. The IMR instructs the appropriate OAD to start the server using the registered command line when client requests arrive at the IMR's binding port. You can register a server to be started on a host other than the host on which the IMR runs by supplying a host name:

```
imradadmin --add-server CCS_server /bin/CCSserver HostB
```

Servers automatically register their persistent POAs with the IMR. You can disable this behavior by setting the **activate-poas** attribute to false (see page 22-15).

Once a server is registered, the first request from a client activates the server. Note that the OAD passes the **-ORBserver_name** option to the server. That option is processed by **ORB_init** and establishes communication between the server and its OAD.

Server Execution Environment

An NT server started by the IMR becomes a detached process.

A UNIX server started by the IMR has the execution environment of a daemon:

- File descriptors `0`, `1`, and `2` are connected to `/dev/null`
- One additional file descriptor is open to the OAD.
- The `umask` is set to `027`.
- The working directory is `/`.
- The server has no control terminal.
- The server is a session and group leader.
- The user and group ID are those of the OAD.
- Signals have the default behavior.



5
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.8 Server Execution Environment

Under NT, the server is created as a detached server. (The standard file descriptors are disconnected.)

Under UNIX, the server is turned into a proper daemon process. This mainly means that the working directory is the root directory and that the standard file descriptors are connected to `/dev/null`.

22.8.1 Security Issues

There are a few consequences of this environment that you need to keep in mind:

- *Never* run the OAD from the `root` user. Doing so means that all servers started by the OAD are run as `root`! The safest approach is to either run the OAD as the user `nobody` or to create a separate user for the OAD and use that user ID exclusively. You can ensure that the OAD starts with the correct privileges by making it `set-uid` and `set-gid` to that user.
- If you have followed the previous advice, you may find that your server has insufficient privileges to do its work. If so, the easy and reliable solution is to make the server executable `set-uid` and `set-gid` to a user and group ID with appropriate privileges. (There are no security issues with artificially raising the privilege of a process by setting s-bits—all the security holes stem from binaries that are `set-uid` to `root` and then do not correctly lower their privilege level when they should.)

22.8.2 Setting the Server Environment

The environment in which a server is started by the OAD means that some things are inconvenient. For example, because the working directory is set to /,¹ you must use absolute pathnames for files. Similarly, you cannot simply write to the standard output for tracing because the server's file descriptors are connected to the null device.

You should consider the following points when you decide to run a server with automatic activation:

- Catch SIGTERM, SIGHUP, and SIGINT and ensure that you shut down cleanly and quickly on receipt of those signals.
- If your server creates child processes, pass received signals to the children; otherwise, you will leave the children abandoned. You can easily do this by using a kill system call with a process ID of 0 to send the signal to all processes in your process group.
- You must log to syslog or redirect your standard file descriptors to a terminal or file for tracing and debugging.
- You cannot rely on environment variables to be set up to anything meaningful because the environment is that of the OAD.
- Set your umask to something meaningful for your server. The default of 027 may not be correct for your needs.

You can do all of these things directly in your server executable. However, doing so is inconvenient and may require quite a complex configuration mechanism. A much better way to start your server with the correct file descriptors, environment variables, umask, or working directory is to not register the server with the IMR, but to register a shell script instead. The minimum skeleton for such a script is:

```
#!/bin/sh
exec "$@"
```

For example, if your script is called /usr/local/bin/launch, you can register the CCS server command as

```
/usr/local/bin/launch /bin/CCSserver
```

As shown, the script does nothing and effectively becomes a no-op. However, you can add whatever setup you require before the script execs the actual server. For example, you can easily change the umask, redirect file descriptors, or set environment variables this way:

```
#!/bin/sh
umask 077
PATH=/bin:/usr/bin:/usr/local/bin; export PATH
HOME=/tmp; export HOME
cd $HOME
exec 1>>/$HOME/CCSserver.stdout
exec 2>>/$HOME/CCSserver.stderr
exec "$@"
```

1. If you change the working directory of your server, you should change to somewhere in the root file system. If you do not, a system administrator cannot unmount the file system without killing your server.

Server Attributes

`imradmin` permits you to set attributes for a registered server with the `--set-server` command:

```
imradmin --set-server <server-name> <mode>
```

Valid modes are:

- **exec**
Changes the executable path for the server.
- **args**
Changes the arguments passed to the server.
- **mode**
Changes the mode. The mode must be **shared** or **persistent**.



6
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.9 Server Attributes

The `--set-server` command allows you to change the registration details of a server.

- The **exec** attribute stores the pathname to the server executable, for example:

```
imradmin --set-server CCS_server exec /usr/local/bin/CCSserver
```

- The **args** attribute changes the additional arguments that are passed to a server, for example:

```
imradmin --set-server CCS_server args -dbfile /tmp/CCS_DB
```

The server will now be invoked as:

```
/usr/local/bin/CCSserver -dbfile /tmp/CCS_DB
```

Note that there are additional options beginning with **-ORB** that are used by the OAD to pass additional information to the server.

- The **mode** attribute is either persistent or shared and changes the activation mode of the server, for example:

```
imradmin --set-server CCS_server mode persistent
```

This changes the server's mode to persistent activation.

Server Attributes (cont.)

- **activate_poas**

If **true**, persistent POAs are registered automatically. If **false**, each persistent POA must be registered explicitly.

- **update_timeout** (msec)

The amount of time the IMR waits for updates to propagate.

- **failure_timeout** (sec)

How long to wait for the server to start up and report as ready.

- **max_spawns**

The number of times to try and restart the server before giving up.

imradadmin --reset-server <server-name>
resets the failure count.



- The **activate_poas** attribute controls whether POAs are registered automatically with the IMR as they are created or must each be explicitly registered with the IMR. For example:

```
imradadmin --set-server CCS_server activate_poas false
imradadmin --add-poa CCS_server Controller
imradadmin --add-poa CCS_server Controller/Thermometers
```

With this registration, only requests for the controller and for thermometers cause server activation, but requests for thermostats do not.

- The **update_timeout** attribute controls how long the IMR waits for status changes to propagate to all the OADs. You should not have to change this value.
Note that there are additional options beginning with **-ORB** that are used
- The **failure_timeout** controls the amount of time the OAD waits for a server to contact it and report as ready before the OAD concludes that the server is hanging or otherwise broken.
- The **max_spawns** attribute controls the number of times the OAD will attempt to restart a server that does not respond within **failure_time** before concluding that the server is permanently broken. Once in that state, the OAD will no longer activate the server until you explicitly reset the server state with the **--reset-server** command:

```
imradadmin --reset-server CCS_server
```

Getting IMR Status

A number of `imradmin` commands show you the status of the IMR and its OADs:

- `imradmin --get-server-info <server-name>`
- `imradmin --get-oad-status [<host>]`
- `imradmin --get-poa-status <server-name> <poa-name>`
- `imradmin --list-oads`
- `imradmin --list-servers`
- `imradmin --list-poas <server-name>`
- `imradmin --tree`
- `imradmin --tree-oad [<host>]`
- `imradmin --tree-server <server-name>`



8
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.10 Getting IMR Status

The IMR closely keeps track of the status of its OADs and servers. You can use various commands to check on the current status and the configuration of the system.

- `imradmin --get-server-info <server-name>`

This command displays the status of the specified server. You can see all the configuration attributes of the server, as well as its current status. The status of a server has one of the following values:

- **forked**
The OAD has created the server process, but the server has not contacted the OAD yet.
- **starting**
The server has initiated contact with the OAD.
- **running**
The server is running and ready to accept requests (that is, the server has activated the POA manager for the Root POA).
- **stopping**
The server has called `ORB::shutdown` but has not exited yet.
- **stopped**
The server is not running.

- **imradmin --get-oad-status [*<host>*]**
This command shows the status (**up** or **down**) of all OADs or the status for the OAD on the specified host.
- **imradmin --get-poa-status *<server-name>* *<poa-name>***
This command shows the status of the specified POA. The status is that of the POA's POA manager, that is **active**, **inactive**, **holding**, or **discarding**. In addition, the **nonexistent** state indicates that the POA is not currently instantiated.
- **imradmin --list-oads**
This command shows all OADs that are registered with the IMR.
- **imradmin --list-servers**
This command lists all servers that are registered with the IMR.
- **imradmin --list-poas *<server-name>***
This command lists all the POA names that are known to the IMR for the specified server.
- **imradmin --tree**
This command lists the complete status of the IMR as a tree structure.
- **imradmin --tree-oad [*<host>*]**
This command limits the OAD status display to the specified host.
- **imradmin --tree-server *<server-name>***
This command limits the server status display to the specified server.

IMR Configuration

1. Set up a configuration file for each host in the location domain.
2. Run an IMR in master or dual mode on exactly one machine in your location domain.
3. Run an OAD on each of the other hosts in the location domain by running the IMR in slave mode.

Once you have configured the IMR, run the `imr` commands from a start-up script in `/etc/rc`.

You can explicitly add an OAD (instead of having OADs add themselves implicitly) with:

```
imradmin --add-oad [<host>]
```

To remove an OAD from the configuration:

```
imradmin --remove-oad [<host>]
```



9
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.11 IMR Configuration

Configuration of the IMR is quite easy:

1. Create a configuration file for each machine on which you want to run an IMR or OAD.
2. Run an IMR in master mode or dual mode on exactly one of those machines.
3. Run an OAD on each of the other hosts by running the IMR in slave mode on those machines.

This completes the configuration. You can now register servers with `imradmin` as discussed in the previous sections.

`imradmin` also offers commands to remove items from the configuration:

- `imradmin --remove-oad [<host>]`
- `imradmin --remove-server <server-name>`
- `imradmin --remove-poa <server-name> <poa-name>`

IMR Properties

The IMR and OAD use configuration properties:

- `ooc.imr.dbdir=<dir>`
- `ooc.imr.forward_port=<port>`
- `ooc.imr.admin_port=<port>`
- `ooc.imr.administrative=<true/false>`
- `ooc.imr.slave_port=<port>`
- `ooc.imr.mode=<dual/master/slave>`
- `ooc.orb.service.IMR=<corbaloc URL>`
- `ooc.imr.trace.peer_status=<level>`
- `ooc.imr.trace.process_control=<level>`
- `ooc.imr.trace.server_status=<level>`



10
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.12 IMR Properties

The IMR reads configuration properties on start-up. These properties determine how the various processes connect to each other and where they store their persistent state. You can control these properties by setting the `ORBACUS_CONFIG` environment variable to the pathname of a configuration file, or you can point a server at a configuration file with the `-ORBconfig <pathname>` option. By controlling these properties, you can run more than one IMR and OAD on a single host and therefore have servers on the same machine use different location domains. This is particularly useful during development and for debugging purposes.

- `ooc.imr.dbdir=<dir>`

This property controls where the IMR stores its persistent state. The current directory is the default.

- `ooc.imr.forward_port=<port>`

This property determines the port number that is written into each persistent IOR. In other words, this is the port number on which client requests are forwarded. The default port is 9998.

- `ooc.imr.admin_port=<port>`

This property determines the port number on which the IMR communicates with its OADs when it is in dual or master mode. The default port is 9999.

- `ooc.imr.administrative=<true/false>`

This property controls whether the IMR runs in administrative mode. When in administrative mode, the IMR accepts modifications to its registration database. When this mode is disabled,

the registration database cannot be modified, that is, it is impossible to change registration of OADs, servers, server attributes, and POAs. If you run an IMR in administrative mode, make sure that the administrative port is not accessible through your firewall!

- **ooc.imr.slave_port=<port>**

This property determines the port number on which the OAD communicates with its IMR. You must set this property on all hosts on which the IMR or an OAD runs, otherwise the IMR cannot find its OADs. (All OADs must use the same port.) The default port is 9997.

- **ooc.imr.mode=<dual/master/slave>**

This property controls the mode in which the IMR runs. In dual mode, it acts as both the IMR and as an OAD for the local host. In master mode, the IMR acts as a pure IMR (that is, it cannot activate server processes on the local host). In slave mode, the IMR acts as a pure OAD (that is, expects an IMR to be running elsewhere).

- **ooc.orb.service.IMR=<corbaloc URL>**

This property contains the IOR to the implementation repository. The IOR denotes the IMR's admin port (**ooc.imr.admin_port**) and is required by tools such as **imradmin**. You can use a **corbaloc** IOR (see Section 19.20) with an object key of **IMR** to set this property:

```
ooc.orb.service.IMR=corbaloc::janus.ooc.com.au:9999/IMR
```

Note that the port number in the IOR must match the setting of **ooc.imr.admin_port**. You must set this property also on each machine that runs an OAD; otherwise, the OAD will not know where to find its IMR.

Setting this property controls the reference that is returned by a call to **resolve_initial_references** with a token of **IMR**.

- **ooc.imr.trace.peer_status=<level>**

This property controls the tracing output for process control messages sent by the IMR to its OADs. Valid levels are 0, 1, and 2. The default level is 0, which produces no output.

- **ooc.imr.trace.process_control=<level>**

This property controls the tracing output for server process (server start-up and shut-down). Valid levels are 0, 1, and 2. The default level is 0, which produces no output.

- **ooc.imr.trace.server_status=<level>**

This property controls the tracing output for diagnostic messages. Valid levels are 0, 1, and 2. The default level is 0, which produces no output.

The Boot Manager

The IMR can act as a boot manager for `corbaloc` references.

A URL of the form

```
corbaloc::<IMR-host>/<token>
```

returns the object reference for the service identified by `<token>` as configured for the IMR.

For example:

```
corbaloc::janus.iona.com/NameService
```

denotes the naming service as returned by `resolve_initial_references` when called by the IMR.

```
-ORBDefaultInitRef corbaloc::janus.iona.com
```

configures the initial reference environment of a client as for the IMR.



11
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.13 The Boot Manager

The IMR also serves as service for bootstrapping. This allows you to easily configure clients to use a defined initial reference environment. All you need to configure are the initial references for the IMR; thereafter, a `corbaloc` reference of the form `corbaloc::<IMR-host>/<token>` refers to the corresponding initial reference configured for the IMR.

By default, the bootstrap service runs at port 2809 (which is also the default port for `corbaloc` references that do not contain an explicit port number). You can change the port for the bootstrap service by setting the `oc.imr.locator_endpoint` property (or by passing the `--locator-endpoint` option to the IMR on start-up).

Once the initial reference environment for the IMR is configured, other processes get the same initial reference environment by setting the `oc.orb.default_init_ref` property (or by passing `-ORBDefaultInitRef` option on the command line).

For example, the following command provides the same initial reference environment to client as the initial reference environment that is configured for the IMR:

```
./client -ORBDefaultInitRef corbaloc::janus.iona.com
```

The advantage of this approach is that you can configure all the initial references (such as for the Naming Service, the Trading Service, etc.) only once for the IMR, and then easily configure any number of clients and servers to use the same set of initial references as the IMR (instead of having to configure the references for all clients and servers separately).

The `mkref` Tool

You can create an object reference on the command line:

```
mkref <server-name> <object-ID> <poa-name>
```

For example:

```
mkref CCS_server the_controller Controller
```

This writes a **corbaloc** reference to standard output that you can use to configure clients:

```
corbaloc::janus.ooc.com.au:9998/%AB%AC%AB0_RootPOA%00forward%00%00%AB%AC%AB0CCS_server%00Controller%00%00the_controller
```

mkref is useful during installation, for example, if you want to produce an IOR for bootstrapping.



12
The Implementation Repository (IMR)
Copyright 2000–2001 IONA Technologies



22.14 The `mkref` Tool

Especially during installation of your application, it is useful to be able to create references to bootstrapping objects without having to run the actual server that hosts the object. The **mkref** command allows you to do this. Note that, as for any URL-style IOR, you should not use references created with **mkref** as a general IOR replacement. They are purely a convenience.

23. Exercise: Using the Implementation Repository

Summary

In this unit, you will register your server for automatic start-up with the IMR and use the `mkref` utility to create a reference from the command line.

Objectives

By the completion of this unit, you will know how to use the Implementation Repository with your applications.

23.1 Source Files and Build Environment

You will find this exercise in your `imr` directory. The files in this directory are the same as for Unit 20, with one addition:

- `launch`

This is a simple shell script that launches the server with the passed arguments and redirects standard output and standard error into the file `/tmp/trace`.

23.2 Server Operation

The server source code provided to you for this exercise implements the solution presented in Unit 21. The purpose of this exercise is to register a server to use the Implementation Repository.

23.3 What You Need to Do

Step 1

Create a configuration file that sets the initial reference for the Naming Service. (See Unit 21). Add to this file all the properties that are required by the IMR.

Step 2

Start the Naming Service and the IMR. Verify that the IMR works by using the `imradmin` command.

Step 3

Instead of launching the server directly from the IMR, we will use the `launch` script as an intermediate step. This allows you to follow any error messages by looking at the contents of `/tmp/trace`.

Register the `launch` script with the IMR as a server. Use your login ID as the server name. Verify that the server was registered correctly with `imradmin`. Set the command-line arguments for `launch` to the absolute path of the server and verify that the registration looks correct.

Step 4

Create an object reference with `mkref` that denotes the controller. Advertise that reference in the Naming Service under the name `<login ID>/controller`.

Step 5

Edit the `client.cpp` and `server.cpp` source files to reflect the correct path in the Naming Service and recompile. Test that the IMR starts your server correctly by running the client and verifying that the server is started on demand.

24. Solution: Using the Implementation Repository

24.1 Solution

Step 1

```
$ cat ob.config
ooc.orb.service.NameService=corbaloc::janus.ooc.com.au:5000/NameService
ooc.orb.service.IMR=corbaloc::janus.ooc.com.au:9999/IMR
ooc.imr.mode=dual
ooc.imr.administrative=true
ooc.imr.dbdir=/home/michi/imr
ooc.imr.admin_port=9999
ooc.imr.forward_port=9998
ooc.imr.slave_port=9997
ooc.imr.trace.peer_status=2
ooc.imr.trace.process_control=2
ooc.imr.trace.server_status=2
$ ORBACUS_CONFIG=`pwd`/ob.config
$ export ORBACUS_CONFIG
```

Step 2

```
$ /opt/OB4/bin/nameserv -OApport 5000 &
[1] 7292
$ imr &
[2] 7295
[ IMR: register_oad: janus ]
[ IMR: OAD for janus processes: EMPTY ]
[ OAD: ready for janus ]
$ imradmin --tree
domain
`-- janus (up)
```

Step 3

```
$ imradmin --add-server CCS_server `pwd`/launch
$ imradmin --tree
domain
`-- janus (up)
   |-- CCS_server (stopped)
$ imradmin --set-server CCS_server args `pwd`/server
$ imradmin --get-server-info CCS_server
Server CCS_server:
  ID: 1
  Status: stopped
  Name: CCS_server
  Host: janus
  Path: /home/michi/labs/imr/launch
  RunDir:
  Arguments: /home/michi/labs/imr/server
```

```
Activation Mode:          shared
POA Activation:          true
Update timeout (ms):     20000
Failure timeout (secs):  60
Maximum spawn count:    2
Started manually:       no
Number of times spawned: 0
```

Step 4

```
$ mkref CCS_server the_controller Controller >ctrl.ref
$ cat ctrl.ref
corbaloc::janus:9998/%AB%AC%AB0_RootPOA%00forward%00%00%AB%AC%AB0
CCS_server%00Controller%00%00the_controller
$ nsadmin -c michi
$ nsadmin --bind michi/controller `cat ctrl.ref`
$ nsadmin -r michi/controller
IOR:01f2894001000000000000000100000000000005a0000000101005011000
0006a616e75732e6f6f632e636f6d2e617500000e273a000000abacab305f526f
6f74504f4100666f72776172640000abacab306d6963686900436f6e74726f6c6
c657200007468655f636f6e74726f6c6c6572
```

25. Threaded Clients and Servers

Summary

This unit covers the threading models supported by ORBacus and the JThreads/C++ abstraction library. It also discusses the options for creating thread-safe servers and what guarantees are provided by the ORB to your application with respect to thread safety.

Objectives

By the completion of this unit, you will know how to create threaded clients and servers using ORBacus and JThreads/C++.

Overview

ORBacus supports a number of concurrency models for clients and servers. These models control how requests are mapped onto threads:

- Blocking (default for clients, applies only to clients)
- Reactive (default for servers)
- Threaded
- Thread-per-Client
- Thread per request
- Thread pool

You can select a concurrency model by setting a property, by passing an option on the command line, or programmatically.



1
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.1 Overview

ORBacus supports various concurrency models for clients and servers. By default, both clients and servers are single-threaded (but calls to servers may be made reentrantly, even though there is only a single thread).

To specify a concurrency model, you can pass a command-line argument, set the appropriate property, or explicitly set the concurrency model from within your program. (See page 25-13.)

NOTE: For multi-threaded servers, request dispatch starts as soon as you activate a POA manager.

The Blocking Concurrency Model

The blocking concurrency model applies only to clients and is the default model.

- After sending a request, the client-side run time enters a blocking read to wait for the reply from the server.
- For **oneway** requests, the ORB avoids blocking the client by holding the request in a buffer if it would block the client. Buffered requests are sent during the next request that goes to the same server.

No other activity can take place in the client while the client waits for a request to complete.

Each call is synchronous for the application code and the ORB.

The blocking model is simple and fast.



25.2 The Blocking Concurrency Model

The blocking concurrency model applies only to clients and is the default model for clients. With this concurrency model, all (tway) requests are synchronous and blocking. As soon as the client invokes an operation, the client-side run time sends the corresponding request to the server and then immediately calls a blocking read on the connection to that server to wait for the reply. The net effect is that the entire process (because there is only a single thread) is blocked until the operation completes on the server side and the reply arrives via the network.

For oneway requests, the client-side run time ensures that the call will not block synchronously by examining the state of the connection first. If sending the request immediately would block the client (because of flow control on the connection when transport buffers are full), the client-side run time allocates a buffer to hold the request and returns from the oneway call immediately. If the client sends another oneway request, that request may end up being buffered as well. Whenever the client sends a twoway request to a server for which there are buffered oneway requests, the client-side run time first sends the buffered oneway requests and then blocks after sending the twoway request until the reply arrives from the server.

The blocking model is attractive because it has low overhead. It uses the minimum number of system calls to send requests and receive their replies and, because the client is single-threaded, there is obviously no overhead for locking or thread context switching.

The Reactive Concurrency Model

The reactive concurrency model applies to clients and servers and is the default for servers.

The reactive model is single-threaded.

- For servers, a `select` loop is used to monitor existing connections. This permits the server to accept requests from several clients.
- For clients, after sending a request, the client-side run time calls `select` instead of using a blocking read.

If the client is also a server, it can accept incoming calls to its objects while it is acting as the client for a synchronous call to some other server.

The reactive model permits nested callbacks for single-threaded servers. (Many ORBs cannot support this without multiple threads.)



25.3 The Reactive Concurrency Model

25.3.1 Reactive Model for Servers

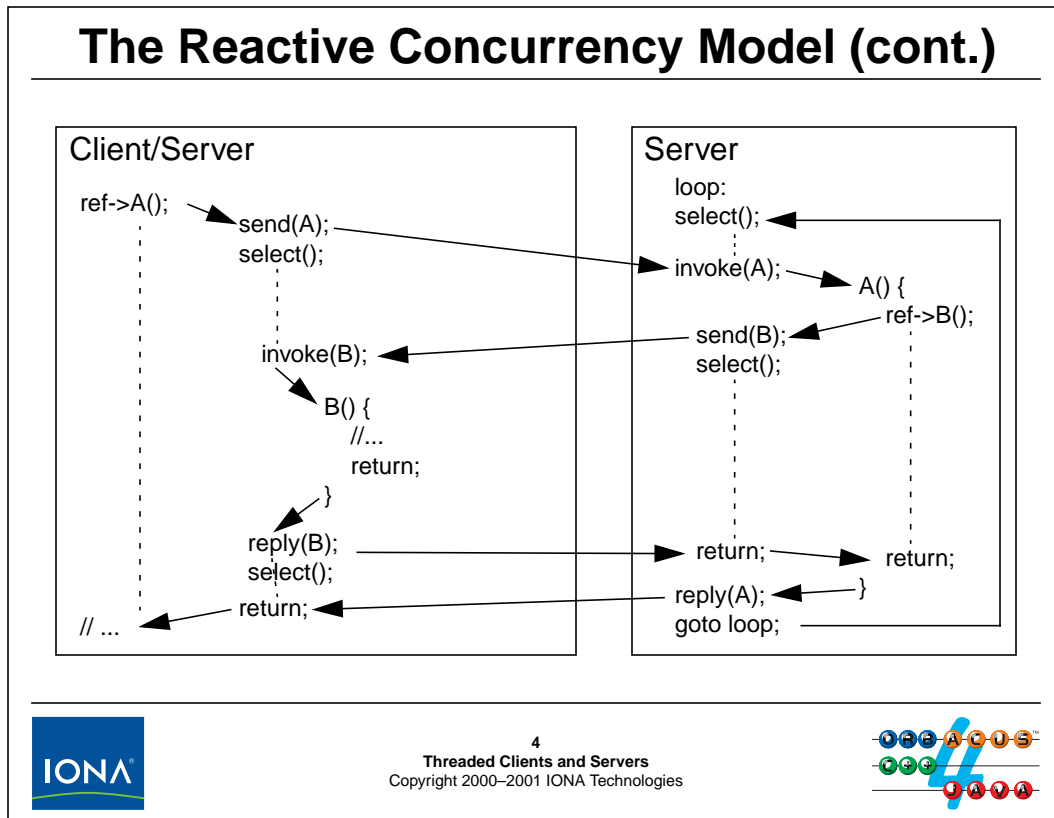
For servers, the reactive concurrency model means that a `select` loop is used in the server to monitor the connections to clients. The server never enters a blocking read, which means that it can accept incoming requests from several clients (and that it can accept new incoming connections from clients whenever it calls `select`).

Note that this model is still single-threaded and that the server cannot respond to network events while it is busy servicing a request. For example, if you implement a CORBA operation such that it takes minutes to complete, all incoming requests from other clients are simply queued until the operation completes. In effect, the reactive model serializes all incoming requests and works by completely finishing one request before paying attention to the next one.

25.3.2 Reactive Model for Clients

On the client side, the reactive model means that, when the client application makes a synchronous call, the client-side run time uses `select` to avoid blocking. It first determines whether the request can be sent immediately without blocking. If so, it sends the request and then calls `select` to be notified when the reply arrives; if not, it monitors the outgoing connection to the server and sends the request once it can avoid blocking and then calls `select` to await the reply.

The advantage of the reactive model for the client side is that, although each call still appears synchronous and blocking to the application, behind the scenes, the run time does not lose its only thread of control because it never makes a blocking system call. This means that, for example, multiple buffered oneway requests can be sent to several servers in parallel.



25.3.3 Nested Callbacks

Consider a simple nested callback scenario. A client invokes an operation A on a server. The implementation of A, in turn, acts as a client and invokes an operation B on another object. If that object is in the same address space as the client that called A, the client really is a combined client and server; we end up with the client having to service an invocation on one of its objects while its only thread is conceptually blocked in a synchronous invocation.

The reactive concurrency model makes it possible for this scenario to work to any level of nesting depth without deadlock. The sequence of events and each transfer of control is outlined above. Initially, the client makes a call on A. That call passes control to the proxy which sends the request for A to the server and then calls **select** to wait for some network activity. Meanwhile, the call has made its way to the server, which comes out of its **select** loop and dispatches the call (synchronously) to the servant for A. The servant, in turn, decides to invoke operation B which happens to be on an object in the original client. The server now behaves like the original client: it sends the request for B and waits in a **select** for some network activity. Meanwhile, the original client has detected the network activity and dispatch the request to the servant for B, which carries out its task and returns, at which point the reply for B is sent to the server. This is detected by the **select** call in the server, which now returns control to A. In turn, A is finished and returns itself which gives the thread of control back to the skeleton for A. The skeleton sends the reply for A back to the client and reenters its **select** loop. Meanwhile, the reply for A makes its way back to the client, where it causes the proxy to return, completing the original operation invocation.

The reactive concurrency model is the only way to permit nested callbacks for single-threaded processes without deadlock. It is somewhat more expensive than the blocking model because it incurs more system calls per request.

The Reactive Concurrency Model (cont.)

Advantages of the reactive concurrency model:

- permits creation of single-threaded processes that are both client and server
- avoids deadlock if callbacks are nested
- asynchronous dispatch of multiple buffered oneway requests to different servers
- transparent to the application code (but beware that operations may be called reentrantly)
- permits integration of foreign event loops



5
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.3.4 Advantages of the Reactive Concurrency Model

The reactive concurrency model offers a few advantages. For one, it permits you to create clients that also act as servers without having to write a multi-threaded program. Second, it avoids deadlocks for nested callbacks. Third, the model is transparent to your code because all the work of monitoring network connections and invoking operations is handled by the ORB run time.

Keep in mind that the reactive model may cause operations to be called reentrantly. For example, if A calls B, and B calls A, you will have a suspended thread of control in A as well as an active thread of control. This means that all writable state that is accessed by operation A must be local to A; if you modify non-local state, the second invocation of A will operate on potentially inconsistent state.

The reactive model for ORBacus is extensible. ORBacus ships with three reactors: the standard select reactor, an X11 reactor, and a Windows reactor. The X11 and Windows reactors permit you to handle GUI events simultaneously with incoming network traffic in a single-threaded server without blocking either the GUI or the CORBA invocations.

You can create reactors for your own purposes by extending the `ORB::Reactor` interface, for example, to react to events generated on network connections for an instrument control protocol.

The Threaded Concurrency Model

The threaded concurrency model applies to clients and servers.

The ORB run time runs with threads, so sending and receiving of network packets can proceed in parallel for many requests.

- For clients, multiple deferred requests sent with the DII are truly dispatched in parallel, and **oneway** invocations do not block.
- For servers, the threaded model demultiplexes requests and unmarshals in parallel.

To the application code, the threaded model appears single-threaded.

Operation bodies in the server are strictly serialized!

This model is useful on multi-processor machines for servers under high load.



6
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.4 The Threaded Concurrency Model

The threaded concurrency model means that the ORB run time runs multi-threaded, but that the application code runs single-threaded. The main advantage of this model is that the ORB run time can do things in parallel, such as request demultiplexing, while the application code is strictly serialized. This means that you do not have to write threaded code to take care of mutual exclusion and data consistency as you would have to if application code were threaded.

The threaded concurrency model is useful particularly on multi-processor machines running servers under high load because the ORB can take advantage of true parallelism for its work. On single-processor machines, the reactive model is usually faster because it incurs less overhead for locking and context switching.

The Thread-per-Client Concurrency Model

The thread-per-client concurrency model applies to the server side.

The ORB creates one thread for each incoming connection.

- Requests coming in on different connections are dispatched in parallel.
- Requests coming in on the same connection are serialized.
- Requests on POAs with **SINGLE_THREAD_MODEL** are serialized.
- Requests on POAs with **ORB_CTRL_MODEL** are dispatched in parallel if those requests are dispatched by different POA managers or are coming from different clients.

The model would better be named “thread-per-connection” because a single server can use multiple POA managers.

You must take care of critical regions in your application with this model!



7
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.5 The Thread-per-Client Concurrency Model

The thread-per-client concurrency model applies only to servers and dispatches requests that come in on different connections in separate threads. In effect, the ORB run time creates a separate dispatch thread for each incoming connection. The model is somewhat misnamed because a single client may have more than one connection to objects in the same server (namely, if the server uses more than one POA manager for its objects). Therefore, “thread-per-connection” would be a more appropriate name.

Each client that sends requests to a server has a separate connection to the server for each of the server’s POA managers. This means that requests that arrive on different connections from the same client are dispatched in parallel, as are requests that arrive from the same client but for different POA managers. (Obviously, for any real parallelism to occur requests from the same client for different POA managers must actually be dispatched in parallel by that client.)

The POA threading policy further determines the degree of parallelism:

- If a POA has the **SINGLE_THREAD_MODEL** policy, requests to that POA will be strictly serialized, no matter where they are coming from.
- If a POA has the **ORB_CTRL_MODEL** policy, requests for that POA will be dispatched in parallel if they arrive via different POA managers (from a threaded single client or from multiple clients) or if they arrive via the same POA manager from different clients.

The thread-per-client concurrency model implies true application parallelism. This means that your application must take care of any critical regions it may have.

The model is efficient because it creates a thread only once per connection. It is useful on both multi- and single-processor machines (because of I/O interleaving).

Thread-per-Request Concurrency Model

The thread-per-request concurrency model only applies to servers.

- Each incoming request creates a new thread and is dispatched in that thread.
- No request for a POA with **ORB_CTRL_MODEL** is ever blocked from dispatch.
- On return from a request, its thread is destroyed.

The thread-per-request model supports nested callbacks with unlimited nesting depth (subject to memory constraints and limits on the maximum number of threads).

The model is inefficient for small operations (thread creation and destruction overhead dominates throughput).

Use this model only for long-running operations that do a substantial amount of work and can proceed in parallel.



25.6 The Thread-per-Request Concurrency Model

The thread-per-request concurrency model creates a new thread for each incoming request, dispatches the request in that thread, and destroys the thread again on completion of the request. All requests for POAs with **ORB_CTRL_MODEL** are dispatched in parallel. (Requests for POAs with **SINGLE_THREAD_MODEL** are still serialized.)

While thread-per-request provides for true parallelism, it also expensive and therefore slow. The cost of thread creation and destruction plus the cost of context switching tend to dominate performance. For this reason, the model is suitable only for long-running operations that do a substantial amount of work and can proceed in parallel.

The Thread Pool Concurrency Model

The thread pool concurrency model dispatches requests onto a fixed-size pool of threads.

- If a thread is idle in the pool, each incoming request is dispatched in a thread taken from the pool.
- The number of concurrent operations in the server is limited by the number of threads in the pool. (The run time uses two additional threads for each connection).
- Requests that arrive while all threads are busy are transparently delayed until a thread becomes idle.

This model is efficient because threads are not continuously created and destroyed and provides a high degree of parallelism.

For general-purpose threaded servers, it is the best model to use.



25.7 The Thread-Pool Concurrency Model

The thread pool concurrency model creates a fixed-size pool of threads and dispatches requests by passing them to an idle thread taken from the pool. The degree of parallelism in the server is limited by the number of threads in the pool. If a request arrives while all threads are busy, that request is blocked until a thread becomes idle and rejoins the pool.

All requests for POAs with `ORB_CTRL_MODEL` are dispatched in parallel for this model. (Requests for POAs with `SINGLE_THREAD_MODEL` are still serialized.)

The thread pool model is far more efficient than the thread-per-request model because it avoids the cost of thread creation and destruction for each request. This model is the best model to use for general-purpose threaded servers that offer a mixture of short- and long-running operations.

Selecting a Concurrency Model

Concurrency models are selected by:

- setting a property in a configuration file
- passing a command-line option
- setting a property programmatically

Properties that apply to concurrency models:

- `ooc.orb.conc_model` (client side)
`blocking` (default), `reactive`, `threaded`
- `ooc.orb.oa.conc_model` (server side)
`reactive` (default), `threaded`, `thread_per_client`,
`thread_per_request`, `thread_pool`
- `ooc.orb.oa.thread_pool=<n>`



10
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.8 Selecting a Concurrency Model

The concurrency model is controlled by three properties:

- `ooc.orb.conc_model`
This property controls the behavior for making calls (that is, when the process acts in the client role). Legal values are **blocking** (which is the default), **reactive**, and **threaded**.
- `ooc.orb.oa.conc_model`
This property controls the behavior for accepting calls (that is, when the process acts in the server role). Legal values are **reactive** (which is the default), **threaded**, **thread-per-client**, **thread_per_request**, and **thread_pool**.
- `ooc.orb.oa.thread_pool=<n>`
This property takes effect only if `ooc.orb.oa.conc_model` is set to **thread_pool** and determines the number of threads in the pool. The default value is 10.

You have a number of ways to select a concurrency model for clients and servers.

25.8.1 Selecting the Concurrency Model via a Configuration File

To select a concurrency model via a configuration file, simply add the corresponding definition to the configuration file for the process (see Unit 18). For example, the following settings set the model to **threaded** for the client role and to **thread_pool** for the server role:

```
# Select threaded for the client role
ooc.orb.conc_model=threaded

# Select thread pool with 20 threads for the server role
ooc.orb.oa.conc_model=thread_pool
ooc.orb.oa.thread_pool=20
```

25.8.2 Selecting the Concurrency Model via Command-Line Options

You can also set the value of a property via command-line options. For example, the following command line sets the same property values as the preceding example:

```
./a.out -ORBthreaded -OAtthread_pool 20
```

If you use a configuration file as well as a command-line option, the command-line option takes precedence.

25.8.3 Selecting the Concurrency Model Programmatically

```
// Get default properties (established by config file)
OB::Properties_var dflt = OB::Properties::getDefaultProperties();

// Initialize a property set with the defaults
OB::Properties_var props = new OB::Properties(dflt);

// Set the properties we want
props->setProperty("ooc.orb.conc_model", "threaded");
props->setProperty("ooc.orb.oa.conc_model", "thread_pool");
props->setProperty("ooc.orb.oa.thread_pool", "20");

// Initialize the ORB with the given properties
CORBA::ORB_var orb = OBCORBA::ORB_init(argc, argv, props);
```

Properties set via `OBCORBA::ORB_init` are overridden by any configuration file specified on the command line (with `-ORBconfig <file>`) and by any explicit command-line options, such as `-ORBthreaded`. However, they do override any properties that are specified by a configuration file set by the `ORBACUS_CONFIG` environment variable.

Overview of JThreads/C++

- JThreads/C++ (JTC) is required for ORBacus to support threaded models.
- JTC is a threads abstraction library.
- JTC is implemented as a thin layer on top of the underlying native threads package.
- JTC adds virtually no overhead.
- JTC provides a Java-like thread model (simpler than POSIX threads).
- JTC shields you from idiosyncrasies of the underlying native threads package.
- JTC provides common synchronization, mutual exclusion, and thread control primitives.



11
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.9 Overview of JThreads/C++

When using one of the threaded concurrency models, ORBacus is fully thread safe. That is, you can call any ORB-related function and be sure that the ORB will correctly synchronize access to its internal data. This means that you need to take care of critical regions only for your application code, but not for the ORB.

For multi-threaded operation, ORBacus is implemented on top of a threads abstraction library called JThreads/C++ (or JTC, for “Java-like Threads for C++”). This library shields the ORB source code from idiosyncrasies of the underlying threads package. Because threads packages differ considerably among operating systems (even versions of UNIX), JTC permits the ORB threading logic to be written independently from the underlying operating system. The same source code works across all versions of UNIX and Windows. If you have purchased JTC, you can take advantage of it in the same way as the ORB.

By using JTC, you gain the advantage that your threaded code is portable across different operating systems (even UNIX and Windows), which can reduce your development time substantially. In addition, JThreads/C++ offers a simpler model than either Windows or POSIX threads, so you get the best of both worlds. JTC provides commonly used and powerful synchronization, mutual exclusion, and thread control primitives. Additionally, migration of source code between C++ and Java becomes easier because the same concepts apply.

JTC is implemented as a thin layer on top of the underlying native threads API and adds virtually no overhead, so there is no performance penalty for using it.

JTC Initialization

Your code must contain a `#include <JTC/JTC.h>` directive.

You must initialize JTC before making any other JTC-related calls by constructing a `JTCInitialize` instance:

```
void JTCInitialize();
void JTCInitialize(int & argc, char * * argv);
```

The second constructor works like `ORB_init` in that it looks for JTC-specific command line options and strips these options from `argv`.

Valid options are:

- `-JTCversion`
- `-JTCss <stack_size>` (in kB)

If you call `ORB_init`, you need not use `JTCInitialize`.



25.10 JTC Initialization

In order to use JTC, your code must include the `JTC/JTC.h` header file to import the relevant declarations. Before you do anything else, initialize an instance of the `JTCInitialize` class. For example:

```
#include <JTC/JTC.h>

// ...

int
main(int argc, char * argv[])
{
    // Initialize JTC
    JTCInitialize jtc(argc, argv);

    // ...
}
```

This code instructs the library to scan the command line for JTC-specific arguments by passing `argc` and `argv` to the constructor. Any options that are recognized by JTC are stripped from `argv` by the constructor.

The `-JTCss` option controls the stack size for threads created by JTC. If not specified, the library uses the default of the underlying threads package.

If you call `ORB_init`, you need not use `JTCInitialize` and you need not include `JTC.h`.

Simple Mutexes

The `JTCMutex` class provides a simple non-recursive mutex:

```
class JTCMutex {
public:
    void lock();
    void unlock();
};
```

You must:

- call `unlock` only on a locked mutex
- call `unlock` only from the thread that called `lock`

Calling `lock` on a mutex that the calling thread has already locked causes deadlock.

Never destroy a mutex that is locked!



13
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.11 Simple Mutexes

The `JTCMutex` class provides a simple mutual exclusion mechanism. Only one thread can hold a mutex at a time, so you can use a mutex to establish simple critical regions in which only one thread can execute at a time. For example:

```
class MyClass {
public:
    void do_something() {
        // ...

        // Start critical region
        m_mutex.lock();

        // Update shared data structure here...

        // End
        m_mutex.unlock()

        // ...
    }
private:
    JTCMutex m_mutex;
};
```

Recursive Mutexes

JTCRecursiveMutex provides a mutex that can be locked multiple times by its owner:

```
class JTCRecursiveMutex {
public:
    void lock();
    void unlock();
};
```

- The first thread to call `lock` locks the mutex and the calling thread becomes its owner.
- Multiple calls to `lock` increment a lock count.
- The owner must call `unlock` as many times as `lock` to unlock the mutex.

Otherwise, the same restrictions apply as for non-recursive mutexes.



25.12 Recursive Mutexes

A recursive mutex is a mutex that keeps track of its owner (that is, the thread that locked it). Multiple calls to `lock` by the same thread are legal and increment a lock count. The mutex must be unlocked as many times as it was locked.

Recursive mutexes are more expensive to implement than non-recursive mutexes because of the need to synchronize access to the lock count internally. However, they are convenient if you have a number of functions that call each other and each use the same mutex to protect a critical region because you do not need to know when it is safe to call each function.

Automatic Unlocking

You must ensure that a mutex is unlocked before it is destroyed.

JTCSynchronized makes this easy:

```
JTCSynchronized {
    JTCSynchronized(JTCMutex & m);           // Lock mutex
    JTCSynchronized(JTCRecursiveMutex & m); // Lock rec. mutex
    JTCSynchronized(JTCMonitor & m);       // Lock monitor
    ~JTCSynchronized();                     // Unlock
};
```

The constructor calls `lock` and the destructor calls `unlock`. This makes it impossible to leave a block containing a **JTCSynchronized** object without calling `unlock`.

JTCSynchronized makes errors much less likely, especially if you have multiple return paths or call something that may throw an exception. The class works for mutexes, recursive mutexes, and monitors.



15
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.13 Automatic Unlocking

Destroying a mutex without unlocking it first causes undefined behavior. (Typically, it results in a deadlock eventually.) The **JTCSynchronized** class makes it impossible to forget to unlock a mutex, a recursive mutex, or a monitor. In order to protect a critical region, you simply instantiate a **JTCSynchronized** object. The critical region ends when the thread of control leaves the block containing the declaration, so you cannot forget to call `unlock`, even if an exception is thrown or if you have multiple return paths out of a block. Using **JTCSynchronized**, we can rewrite the example on page 25-16 as follows:

```
class MyClass {
public:
    void do_something() {
        // ...

        // Start critical region
        JTCSynchronized lock(m_mutex);

        // Update shared data structure here...
        // ...

    } // Critical region ends here
private:
    JTCMutex m_mutex;
};
```

Monitors

The `JTCMonitor` class implements a Java-like monitor:

```
class JTCMonitor {
public:
    JTCMonitor();
    virtual ~JTCMonitor();
    void wait(); // Wait for condition
    void wait(long n); // Wait at most n msec for condition
    void notify(); // Wake up one thread
    void notifyAll(); // Wake up all threads
};
```

Only one thread can enter the critical region protected by a monitor.

A thread inside the region can call `wait` to suspend itself and give access to another thread.

When a thread changes the condition, it calls `notify` to wake up a thread that was waiting for the condition to change.



25.14 Monitors

A monitor provides a way to establish critical regions that provide access to only one thread at a time. However, in contrast to a mutex, a monitor permits you to suspend a thread until a particular condition holds and to suspend the thread if the condition is false. This allows another thread to enter the critical region. (Hopefully, that thread will eventually do something that makes the condition true.)

When the condition is true, the second thread calls `notify` and leaves the critical region. This wakes up the first thread which now finds the condition true and continues inside the critical region (with exclusive access) until it leaves the critical region again.

Simple Producer/Consumer Example

Assume we have a simple queue class:

```
template<class T> class Queue {
public:
    void enqueue(const T & item);
    T    dequeue();
};
```

- Producer threads read items from somewhere and place them on the queue by calling **enqueue**.
- Consumer threads fetch items from the queue by calling **dequeue**.
- The queue is a critical region and the consumer threads must be suspended when the queue is empty.



17
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.15 Simple Producer/Consumer Example

To illustrate the use of monitors, let us examine a very simple unbounded queue. The queue itself is a critical region, which means that only one thread can be in the `enqueue` or `dequeue` functions at a time. In addition, we want to permit multiple consumer and producer threads to remove and deposit items. However, if the queue is empty, we need to suspend the consuming thread.

A very simple non-threaded implementation of the queue looks like this:

```
#include <list>

template<class T> class Queue {
public:
    void enqueue(const T & item) {
        m_q.push_back(item);
    }
    T    dequeue() {
        T item = m_q.front();
        m_q.pop_front();
        return item;
    }
private:
    list<T> m_q;
};
```

To make this queue thread-safe, we simply inherit from `JTCMonitor`.

The implementation of `enqueue` (called by producer threads) simply locks the monitor, deposits an item, and calls `notify`. If one or more consumer threads are currently waiting for an item to arrive, the call to `notify` wakes up one consumer thread; otherwise, the call to `notify` is forgotten and has no effect.

The implementation of `dequeue` (called by consumer threads) first checks whether items are available. If not, it calls `wait` to suspend the calling thread. Calling `wait` also releases the lock on the critical region. Otherwise, if items are in the queue, `dequeue` removes one item from the head of the queue and leaves the critical region unlocked:

```
#include <list>
#include <JTC/JTC.h>

template<class T> class Queue : JTCMonitor {
public:
    void    enqueue(const T & item) {
            JTCSynchronized lock(*this);
            m_q.push_back(item);
            notify();
        }
    T      dequeue() {
            JTCSynchronized lock(*this);
            while (m_q.size() == 0) {
                try {
                    wait();
                } catch (const JTCInterruptedException &) {
                }
            }
            T item = m_q.front();
            m_q.pop_front();
            return item;
        }
private:
    list<T> m_q;
};
```

It is important to note that, if a call to `notify` wakes up a consumer thread that is currently suspended inside `wait`, the consumer thread comes out of `wait` with the critical region locked. In other words, a return from `wait` also guarantees exclusive access to the critical region.

Also note that, after a return from `wait`, the consumer retests the condition in a `while` loop before dequeuing an item and also checks whether the `wait` was interrupted. We will examine the reason for this on page 25-24.

Rules for Using Monitors

You must always catch and ignore a `JTCInterruptedException` exception around a `wait`:

```
T dequeue() {
    JTCSynchronized lock(*this);
    while (m_q.size() == 0)
        wait();                // WRONG!!!
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

Failure to catch and ignore the exception may result in undefined behavior.

In addition, you must call `wait` and `notify` with the mutex locked; otherwise, you get a `JTCIllegalMonitorState` exception.



25.16 Rules for Using Monitors

You must follow a few simple rules when using monitors. One of these is that you must catch and ignore a `JTCInterruptedException` when calling `wait`:

```
T dequeue() {
    JTCSynchronized lock(*this);
    while (m_q.size() == 0) {
        try {
            wait();
        } catch (const JTCInterruptedException &) { // Correct
        }
    }
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

This is necessary because you may get a spurious interrupted system call exception from some threads packages.

Make sure that you hold the lock on the monitor when calling `wait` or `notify`. If you don't, the calls throw a `JTCIllegalMonitorState` exception.

Rules for Using Monitors (cont.)

Always test the condition under protection of the monitor:

```
T dequeue() {
    while (m_q.size() == 0) {           // WRONG!!!
        JTCSynchronized lock(*this);
        try {
            wait();
        } catch (const JTCInterruptedException &) {
        }
        T item = m_q.front();
        m_q.pop_front();
        return item;
    }
}
```

If you do not acquire access to the critical region first, the condition may be changed by another thread in between the test and the update.



You must acquire access to the critical region *before* you test the condition. If you test the condition first and then lock the monitor, the condition may have been changed in the mean time by another thread. This can result in live lock or undefined behavior. You must *always* acquire the lock before looking at anything inside the critical region (even if you only read a single integer—read access of even a simple value is *not* guaranteed to be atomic):

```
T dequeue() {
    JTCSynchronized lock(*this); // Correct
    while (m_q.size() == 0) {
        try {
            wait();
        } catch (const JTCInterruptedException &) {
        }
    }
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

Rules for Using Monitors (cont.)

Always retest the condition when coming out of a `wait`:

```
T dequeue() {
    JTCsynchronized lock(*this);
    if (m_q.size() == 0) {           // WRONG!!!
        try {
            wait();
        } catch (const JTCInterruptedException &) {
        }
    }
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

If you do not retest the condition, it may not be what you expect!



20
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



You must always use a `while` loop to retest the condition before entering a critical region:

```
T dequeue() {
    JTCsynchronized lock(*this);
    while (m_q.size() == 0) {       // Correct
        try {
            wait();
        } catch (const JTCInterruptedException &) {
        }
    }
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

This is necessary because some threads packages can send what is known as “spurious wakeups”. In other words, it is possible for a thread suspended in `wait` to wake up even though no other thread called `notify` (or, more commonly, a call to `notify` can incorrectly wake up more than one thread). The solution is to retest the condition and, if it is still false, wait again, which is achieved by the `while` loop.

Static Monitors

Occasionally, you need to protect static data from concurrent access. You can safely use a `JTCMonitor` to do this:

```
class StaticCounter {
public:
    static void          inc() { JTCSynchronized lock(m_m);
                          ++m_counter;          }
    static void          dec() { JTCSynchronized lock(m_m);
                          --m_counter;          }
    static unsigned long val() { JTCSynchronized lock(m_m);
                          return m_counter;      }
private:
    static unsigned long m_counter;
    static JTCMonitor    m_m;
};

unsigned long  StaticCounter::m_counter = 0;
JTCMonitor     StaticCounter::m_m;
```



25.17 Static Monitors

You can protect static data with a static instance of `JTCMonitor`, with some restrictions. In particular, you must not use any of the monitor's member functions until after you have instantiated one or more `JTCInitialize` objects and you must not use the monitor after the final instance of `JTCInitialize` goes out of scope. Note that construction and destruction of a static `JTCMonitor` is not considered "use", so the above example is guaranteed to work.

The JTCThread Class

To create a new thread, you instantiate a class instance that is derived from `JTCThread`:

```
class JTCThread : public virtual JTCRefCount {
public:
    JTCThread();
    ~JTCThread();
    virtual void run();
    void start();
    // ...
};
```

Override the `run` method to provide a starting stack frame for the thread.

Call `start` to set the thread running in its starting stack frame.



25.18 The JTCThread Class

ORBacus creates threads for you depending on the concurrency model you have selected. However, on occasion, you may want to create threads for your own purposes. To create a thread, you must derive a class from `JTCThread` and provide an implementation for `run`. The code in `run` becomes the thread's starting code. For example:

```
class ProducerThread : public virtual JTCThread {
public:
    ProducerThread(Queue<int> & q, int c) :
        m_q(q), m_c(c) {}
    virtual void run() {
        for (int i = 0; i < m_c; ++i)
            m_q.enqueue(i);
    }
private:
    Queue<int> & m_q; // Thread-safe queue
    int m_c;
};
```

Here, we have a simple producer thread that, given a queue of integers and a count, adds the requested number of integers to the queue. Similarly, we can create a consumer thread that removes the requested number of items from a queue:

```

class ConsumerThread : public virtual JTCThread {
public:
    ConsumerThread(Queue<int> & q, int c) :
        m_q(q), m_c(c) {}

    virtual void run() {
        for (int i = 0; i < m_c; ++i)
            m_q.dequeue(i);
    }

private:
    Queue<int> & m_q; // Thread-safe queue
    int m_c;
};

```

Neither of these thread classes does anything particularly useful, but it serves to illustrate the concepts. (Note that the `Queue<int>` class used by these two classes is assumed to be thread safe.)

To start a consumer and a producer thread, we can write:

```

// ...

JTCThreadInitialize jtcinit; // Important!!!

Queue<int> the_queue; // Queue to use

JTCThreadHandle consumer; // Note: JTCThreadHandle
JTCThreadHandle producer; // Note: JTCThreadHandle

// Start both threads
consumer = new ConsumerThread(the_queue, 10000);
producer = new ProducerThread(the_queue, 10000);
consumer->start();
producer->start();

```

The `start` method on each thread starts it running in its `run` method. Note that we assign the return value from `new` for each thread to a `JTCThreadHandle` instance. This class implements reference counting for threads (just as a `_var` class implements reference counting in the C++ mapping). Assigning the returned thread pointer to a `JTCThreadHandle` class relieves us from having to remember when the thread can be safely destroyed. As for `_var` classes, you can access the member functions of the underlying `JTCThread` instance via the overloaded `->` operator.

Joining with Threads

Given a thread, any other thread can join with it:

```
class JTCThread : public virtual JTCRefCount {
public:
    // ...
    void          join();
    void          join(long msec);
    void          join(long msec, int nsec);
    bool          isAlive() const;
    //...
};
```

The purpose of `join` is to suspend the caller until the thread being joined with terminates.

Always join with threads in a loop, catching `JTCInterruptedException` and reentering `join` if that exception was thrown!



25.19 Joining with Threads

Given a `JTCThread` or `JTCThreadHandle` object, any thread can join with any other thread. Joining with a thread means that the caller of `join` is suspended until the thread being joined with completes its `run` method. The `join` method is overloaded so you can limit the amount of time to wait for a thread to terminate.

While in a `join`, the calling thread may get a `JTCInterruptedException`. This means that you *must* write your calls to `join` as follows:

```
// ...

// Wait for consumer thread to finish
do {
    try {
        consumer->join();
    } catch (const JTCInterruptedException &) {
    }
} while (consumer->isAlive());
```

The call to `isAlive` in the terminating condition of the loop ensures that, in the presence of a stray exception, the caller does not erroneously conclude that the thread being joined with has terminated.

Other JThreads/C++ Functionality

JThreads/C++ offers many more features:

- Named threads
- thread groups
- thread priorities
- `sleep` and `yield`
- thread-specific storage

Please consult the manual for details.



24
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.20 Other JThreads/C++ Functionality

The JThreads/C++ library offers many more features than described here. Please have a look at the manual to familiarize yourself with the complete API.

Synchronization Strategies for Servers

You can use several strategies for synchronization:

- Permit only one concurrent request per servant.

This approach is very easy to implement with a monitor.

- Allows multiple concurrent read operations but require exclusive access to the entire object for a write operation.

This approach provides more parallelism at the cost of greater implementation complexity. (You need to create reader/writer locks and synchronize explicitly by calling `wait` and `notify`.)

Use this approach only if you have high contention on a servant, for example, with default servants.

For both approaches, take care of interactions among life cycle and collection manager operations!



25
Threaded Clients and Servers
Copyright 2000–2001 IONA Technologies



25.21 Synchronization Strategies for Servers

By far the easiest way to synchronize access to your servants is to use a monitor for each servant. This ensures that only one thread can be in any operation on the servant at a time. The advantage of this approach is that it requires almost zero effort. The downside is that invocations are serialized if they are serviced by the same servant. For many implementations, this is not an issue. However, if you use the same servant to incarnate multiple objects (with the `MULTIPLE_ID` policy) or use a default servant, you may find that performance is hardly better than with a single-threaded server because everything is serialized on that servant.

Another approach is to permit finer-grained parallelism by using reader/writer locks. The idea is to permit concurrent access to the same servant for read operations, and to require exclusive access to the servant only for update operations. This provides much better performance if you have a lot of contention on the servant, but requires considerably more implementation effort.¹

Whichever approach you choose, a simple one thread per object approach will not work if you also support life cycle and collection manager operations. For example, if one thread is currently in the middle of the `list` operation on the controller, it is unlikely that the `destroy` operation on a device will be successful if carried out concurrently (because both operations modify shared internal data structures). This means that you must protect access to such shared data separately.

1. How to implement this approach is beyond the scope of this course. For an in-depth discussion, see Kleiman, S., et. al. 1995. *Programming With Threads*. Englewood, NJ: Prentice Hall.

Basic Per-Servant Synchronization

For basic per-servant synchronization, use inheritance from **JTCMonitor** for the servant:

```
class Thermometer_impl :
    public virtual POA_CCS::Thermometer,
    public virtual PortableServer::RefCountServantBase,
    public virtual JTCMonitor {
// ...
};
```

In each operation body, instantiate a **JTCSynchronized** object on entry to the operation.

With almost zero effort, all operations on the servant are serialized.

JTCMonitor uses recursive mutexes, so an operation implementation can invoke operations on its own servant without deadlock.



25.22 Basic Per-Servant Synchronization

Achieving per-servant synchronization is almost too easy with JThreads/C++. All you need to do is inherit from **JTCMonitor** and instantiate a **JTCSynchronized** object on entry to every operation. For example:

```
// IDL model attribute.
CCS::ModelType
Thermometer_impl::
model() throw(CORBA::SystemException)
{
    JTCSynchronized lock(*this);
    // ...
}

// IDL asset_num attribute.
CCS::AssetType
Thermometer_impl::
asset_num() throw(CORBA::SystemException)
{
    JTCSynchronized lock(*this);
    // ...
}

// etc...
```

Life Cycle Considerations

You must pay attention to potential race conditions for life cycle operations and collection manager operations:

- Factory operations, such as **create_thermometer** and **create_thermostat**, must interlock with themselves and with **destroy**.
- **destroy** must interlock with itself and with the factory operations.
- Collection manager operations, such as **list** and **find**, must interlock among each other and with the life cycle operations.

The easiest solution is to have a global life cycle lock.

This serializes all life cycle and collection manager operations, but permits other operations to proceed in parallel (if they are for different target objects).



25.23 Life Cycle Considerations

If you support life cycle or collection manager operations, you must interlock these operations against each other because they all access shared state (such as the list of devices that is part of our `Controller_impl` class). While it is possible to provide fine-grained access to at least some of these operations (for example, by locking on a per-device basis), it is usually not worth the effort: life cycle operations and collection manager operations are typically carried out only rarely, so the increased concurrency is usually not realized.

The simple solution is to serialize all operations that either touch life cycle or operate on the collection of devices. We can achieve this by making the controller implementation inherit from `JTCMonitor` and adding a boolean member variable to the class; that variable is true whenever it is OK to proceed with a life cycle or collection manager operation:

```
class Controller_impl :
    public virtual POA_CCS::Controller,
    public virtual PortableServer::RefCountServantBase,
    public virtual JTCMonitor {
private:
    bool m_lifecycle_ok;    // True if OK to do a life cycle op

public:
    // ...
};
```

Next, we need to make sure that a thread that wants to enter a life cycle or collection manager operation atomically tests whether `m_lifecycle_ok` is true and suspends itself if it is false. Similarly, we want to make sure that a thread that is currently suspended on the `m_lifecycle_ok` condition gets woken up when the condition becomes true. We can do this by adding two methods to the controller, one to acquire the lock and one to release it:

```
class Controller_impl :
    public virtual POA_CCS::Controller,
    public virtual PortableServer::RefCountServantBase,
    public virtual JTCMonitor {
private:
    bool m_lifecycle_ok;    // True if OK to do a life cycle op
public:
    // Life cycle guard methods
    void lifecycle_lock() {
        JTCSynchronized lock(*this);
        while (!m_lifecycle_ok) {
            try {
                wait();
            } catch (const JTCInterruptedException &) {
            }
        }
        m_lifecycle_ok = false;
    }
    void lifecycle_unlock() {
        JTCSynchronized lock(*this);
        m_lifecycle_ok = true;
        notify();
    }
    // ...
};
```

The `lifecycle_lock` and `lifecycle_unlock` methods implement a simple sleep/wake-up mechanism. If a thread that wants to enter a critical region finds the `m_lifecycle_ok` variable false, it goes to sleep in `lifecycle_lock`. Whenever a thread leaves its critical region, it sets the variable to true and wakes up a thread that may currently be waiting to enter the critical region. Of course, we need to make sure that things are initialized properly and set the `m_lifecycle_ok` variable to true in the constructor:

```
Controller_impl::
Controller_impl(const char * asset_file) throw(int)
    : m_asset_file(asset_file), m_lifecycle_ok(true)
{
    // ...
}
```

We can now call the `lifecycle_lock` and `lifecycle_unlock` methods on entry and exit of every life cycle operation. For example:

```

CCS::Thermometer_ptr
Controller_impl::
create_thermometer(CCS::AssetType anum, const char * loc)
throw(CORBA::SystemException, CCS::Controller::DuplicateAsset)
{
    m_ctrl->lifecycle_lock();

    if (exists(anum))
        throw CCS::Controller::DuplicateAsset();    // OOPS!!!
    if (ICP_online(anum) != 0)
        abort();
    if (ICP_set(anum, "location", loc) != 0)
        abort();
    Thermometer_impl * t = new Thermometer_impl(anum);
    PortableServer::ObjectId_var oid = make_oid(anum);
    Thermometer_impl::poa()->activate_object_with_id(oid, t);
    t->_remove_ref();

    m_ctrl->lifecycle_unlock();

    return t->_this();
}

```

This is right, provided we make sure that `lifecycle_unlock` is always called before the operation returns. Of course, this is not the case in the above code example because it leaves the critical region locked when the code throws an exception.

The solution is to use Stroustrup's "resource acquisition is initialization" pattern. We can use a helper object that calls `lifecycle_lock` in the constructor and `lifecycle_unlock` in the destructor, so it becomes impossible to leave the scope in which the object is instantiated without unlocking the critical region. The obvious place to do this is as a nested class in the controller implementation:

```

class Controller_impl :
    public virtual POA_CCS::Controller,
    public virtual PortableServer::RefCountServantBase,
    public virtual JTCMonitor {
private:
    bool m_lifecycle_ok;    // True if OK to do a life cycle op
public:
    // Life cycle methods
    void lifecycle_lock() { /* ... */ };
    void lifecycle_unlock() { /* ... */ };
    class LifeCycleSynchronized {
    public:
        static Controller_impl * m_ctrl;
        LifeCycleSynchronized() { m_ctrl->lifecycle_lock(); }
        ~LifeCycleSynchronized() { m_ctrl->lifecycle_unlock(); }
    };
    // ...
};

```

The `LifeCycleSynchronized::m_ctrl` member can be static because there is only one instance of the controller. (The constructor of `Controller_impl` initializes the member.) The advantage is that, when we lock the critical region, we need not specify the controller instance.

With this machinery in place, making the life cycle and collection manager operations thread safe simply requires instantiating `Controller_impl::LifeCycleSynchronized` object on entry to each operation:

```

CCS::Thermometer_ptr
Controller_impl::
create_thermometer(CCS::AssetType anum, const char * loc)
throw(CORBA::SystemException, CCS::Controller::DuplicateAsset)
{
    LifeCycleSynchronized lock;
    // ...
}

CCS::Thermostat_ptr
Controller_impl::
create_thermostat(
    CCS::AssetType  anum,
    const char*    loc,
    CCS::TempType  temp)
throw(
    CORBA::SystemException,
    CCS::Controller::DuplicateAsset,
    CCS::Thermostat::BadTemp)
{
    LifeCycleSynchronized lock;
    // ...
}

CCS::Controller::ThermometerSeq *
Controller_impl::
list() throw(CORBA::SystemException)
{
    LifeCycleSynchronized lock;
    // ...
}

void
Controller_impl::
find(CCS::Controller::SearchSeq & slist)
throw(CORBA::SystemException)
{
    LifeCycleSynchronized lock;
    // ...
}

```

Note that there is no need to establish a critical region for the `change` operation. That is because `change` simply loops over a list of references and invokes an operation on each one. However,

especially for multi-threaded operation, `change` may fail for another reason: another thread may have destroyed a device, so one or more of the references in the list passed to `change` may denote a non-existent object. To fix this, we need to change the IDL for the CCS to add another error indicator of some kind (which is left as an exercise for the lab session for this unit).

The changes we have made so far take care of interlocking all life cycle operations, except for `destroy`. The `destroy` operation requires special care: we cannot simply remove the persistent state for the device inside `destroy` because other member functions, such as `set_nominal`, may still be executing concurrently. This means that `destroy` simply marks the device as removed and calls `deactivate_object`. The key here is to note that `destroy` does *not* release the life cycle lock before it returns:

```
void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    m_ctrl->lifecycle_lock();

    // Remove entry in the AOM for the servant.
    // Controller map and persistent state are cleaned up in
    // the servant destructor.
    PortableServer::ObjectId_var oid = make_oid(m_anum);
    PortableServer::POA_var poa = _default_POA();
    poa->deactivate_object(oid);

    m_removed = true;    // Mark device as destroyed

    // Note: lifecycle lock is still held.
}
```

The call to `deactivate_object` eventually results in the servant destructor to run (namely, once the last operation on the servant has completed) because `deactivate_object` drops the reference count on the servant. Once the destructor runs, we know that the servant is no longer needed and clean up the persistent state for the object. The final step is to unlock the life cycle lock so other threads can proceed with life cycle operations:

```
Thermometer_impl::
~Thermometer_impl()
{
    // Remove device from map and take it off-line
    // if it was destroyed.
    if (m_removed) {
        m_ctrl->remove_impl(m_anum);
        if (ICP_offline(m_anum) != 0)
            abort();
    }

    // Permit life cycle operations again.
    m_ctrl->lifecycle_unlock();
}
```


Threading Guarantees for the POA

- `_add_ref` and `_remove_ref` are thread safe.
- For requests arriving on the same POA
 - calls to **incarnate** and **etherealize** on a servant activator are serialized,
 - calls to **incarnate** and **etherealize** are mutually exclusive,
 - **incarnate** is never called for a specific object ID while that object ID is in the AOM.
- For requests arriving on different POAs with the same servant activator, no serialization guarantees are provided.
- **preinvoke** and **postinvoke** are not interlocked. **preinvoke** may be called concurrently for the same object ID.
- **preinvoke**, the operation, and **postinvoke** run in one thread.



25.24 Threading Guarantees for the POA

The POA provides a number of threading guarantees:

- `_add_ref` and `_remove_ref` are thread safe.
This guarantee ensures that the reference count will not be destroyed by concurrent access to the same servant.
- If you use a servant activator for a POA
 - calls to `incarnate` and `etherealize` on a servant activator are serialized,
 - calls to `incarnate` and `etherealize` are mutually exclusive,
 - `incarnate` is never called for a specific object ID while that object ID is in the AOM.

These guarantees make sure that you will not attempt to activate a servant for the same object twice, or try to activate a servant at the same time as a different thread attempts to destroy it.

- For requests arriving on different POAs with the same servant activator, no serialization guarantees are provided.

If you share a servant activator among several POAs, there are no serialization guarantees for requests that arrive via different POAs. This rarely presents a problem because you cannot share servants across different POAs anyway. (The specification permits this, but we strongly suggest you don't do this because it is almost impossible to get right.) However, you must make sure that the code for `activate` and `incarnate` is reentrant because multiple threads may be calling these methods concurrently if you share a servant activator among POAs.

- `preinvoke` and `postinvoke` are not interlocked. `preinvoke` may be called concurrently for the same object ID.

There are no serialization guarantees for `preinvoke` and `postinvoke`. The operations may execute concurrently in any number of threads. This means that they must be reentrant and, in addition, must make sure that anything they do is thread safe. The POA guarantees that `preinvoke` and `postinvoke` will always be called in pairs though, so it cannot happen that you get a call to `preinvoke` without the corresponding call to `postinvoke`.

- `preinvoke`, the operation, and `postinvoke` run in the same thread.

The POA guarantees that it will call `preinvoke`, the operation body, and `postinvoke` in the same thread. This means that you can, for example, communicate via `preinvoke` and `postinvoke` by using thread-specific storage, or acquire a lock in `preinvoke` and release that lock in `postinvoke`.