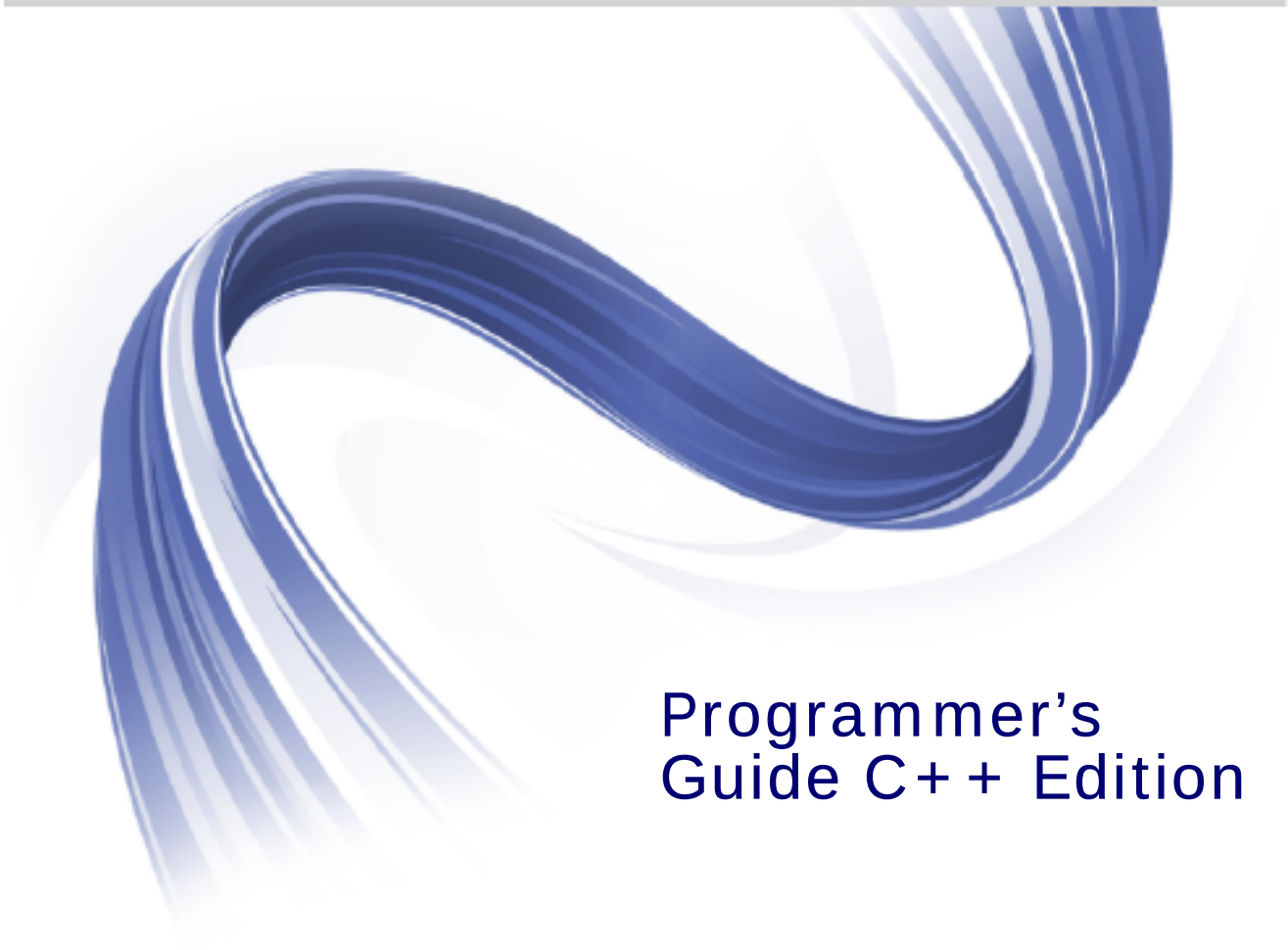
MICRO FOCUS

Orbix 3.3.13



Programmer's Guide C++ Edition

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2015-09-10

Contents

Part IV Advanced Orbix C++ Programming

Preface

Orbix is a standards-based programming environment for building and integrating distributed applications. Orbix is a full implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA).

Audience

This guide is intended for use by application programmers who wish to familiarize themselves with distributed programming with Orbix. This guide addresses all levels of Orbix programming, from getting started to advanced topics. Users should be familiar with the C++ programming language. Before reading this guide, you should read the ***Introduction to Orbix C++ Edition*** manual.

Organization of this Guide

The ***Orbix Programmer's Guide C++ Edition*** is divided into four parts as follows:

Part I, Getting Started

This part describes a simple example that enables you to get started with Orbix programming. Read this part first to get a sense of how the Orbix programming environment works.

Part II, Orbix C++ Programming

This part describes the core topics of Orbix programming that all programmers need to know. Read this part to learn the main programming techniques that most Orbix applications require.

Part III, Dynamic Orbix C++ Programming

This part describes a special subset of Orbix programming components that allow you to write *dynamic* applications. The concept of dynamic Orbix programming is described in this section. Each chapter is dedicated to a single dynamic Orbix component.

Part IV, Advanced Orbix C++ Programming

Orbix extends the CORBA specification by adding features that allow you to write more flexible distributed applications. Each chapter in this part describes an advanced Orbix feature. Browse this part to discover the advanced features available in Orbix and select the features that may be useful in your applications.

Part V, Appendix

This contains an appendix listing the command-line options to the Orbix IDL compiler.

Document Conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent emphasis and new terms. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`* Note: some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, no prompt is used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, or Windows 95 command prompt.
... . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/orbix/orbix-6.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx. (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Part I

Getting Started

In this part

This part contains the following:

Introduction to CORBA and Orbix

Orbix is a software environment that allows you to build and integrate distributed applications. Orbix is a full implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) specification. This chapter introduces CORBA and describes how Orbix implements this specification.

CORBA and Distributed Object Programming

The diversity of modern networks makes the task of network programming very difficult. Distributed applications often consist of several communicating programs written in different programming languages and running on different operating systems. Network programmers must consider all of these factors when developing applications.

The Common Object Request Broker Architecture (CORBA) defines a framework for developing object-oriented, distributed applications. This architecture makes network programming much easier by allowing you to create distributed applications that interact as though they were implemented in a single programming language on one computer.

CORBA also brings the advantages of object-oriented techniques to a distributed environment. It allows you to design a distributed application as a set of cooperating objects and to re-use existing objects in new applications.

The Role of an Object Request Broker

CORBA defines a standard architecture for Object Request Brokers (ORBs). An ORB is a software component that mediates the transfer of messages from a program to an object located on a remote network host. The role of the ORB is to hide the underlying complexity of network communications from the programmer.

An ORB allows you to create standard software objects whose member functions can be invoked by *client* programs located anywhere in your network. A program that contains instances of CORBA objects is often known as a *server*.

When a client invokes a member function on a CORBA object, the ORB intercepts the function call. As shown in Figure 1, the ORB redirects the function call across the network to the target object. The ORB then collects results from the function call and returns these to the client.

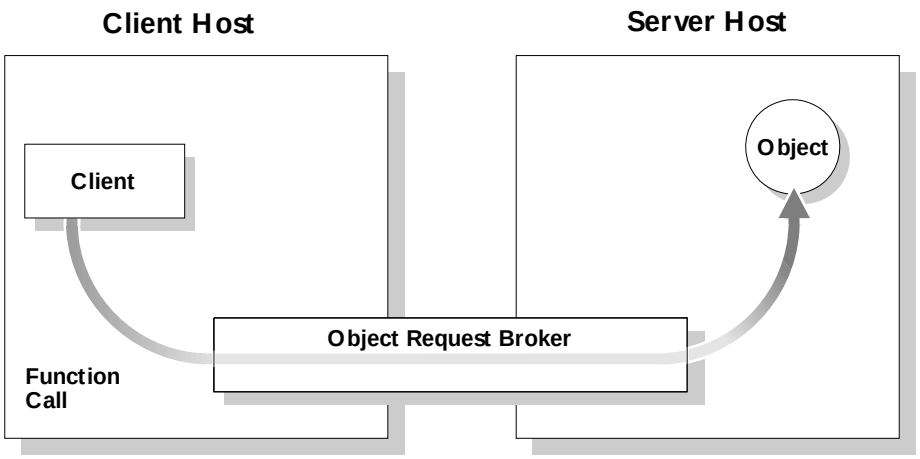


Figure 1: *The Object Request Broker*

The Nature of Objects in CORBA

CORBA objects are just standard software objects implemented in any supported programming language. CORBA supports several languages, including C++, Java, and Smalltalk.

With a few calls to an ORB's application programming interface (API), you can make CORBA objects available to client programs in your network. Clients can be written in any supported programming language and can call the member functions of a CORBA object using the normal programming language syntax.

Although CORBA objects are implemented using standard programming languages, each CORBA object has a clearly-defined interface, specified in the CORBA Interface Definition Language (IDL). The interface definition specifies which member functions are available to a client, without making any assumptions about the implementation of the object.

To call member functions on a CORBA object, a client needs only the object's IDL definition. The client does not need to know details such as the programming language used to implement the object, the location of the object in the network, or the operating system on which the object runs.

The separation between an object's interface and its implementation has several advantages. For example, it allows you to change the programming language in which an object is implemented without changing clients that access the object. It also allows you to make existing objects available across a network.

The Structure of a CORBA Application

The first step in developing a CORBA application is use CORBA IDL to define the interfaces to objects in your system. You then compile these interfaces using an IDL compiler.

An IDL compiler generates C++ from IDL definitions. This C++ includes *client stub code*, which allows you to develop client programs, and *object skeleton code*, which allows you to implement CORBA objects.

As shown in Figure 2, when a client calls a member function on a CORBA object, the call is transferred through the client stub code to the ORB. If the client has not accessed the object before, the ORB refers to a database, known as the *Implementation Repository*, to determine exactly which object should receive the function call. The ORB then passes the function call through the object skeleton code to the target object.

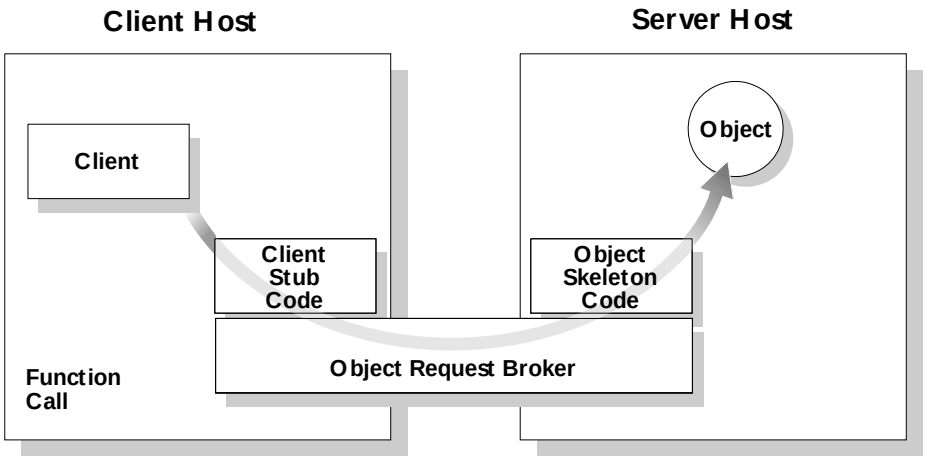


Figure 2: *Invoking on a CORBA Object*

The Structure of a Dynamic CORBA Application

One difficulty with normal CORBA programming is that you have to compile the IDL associated with your objects and use the generated C++ code in your applications. This means that your client programs can only call member functions on objects whose interfaces are known at compile-time. If a client wishes to obtain information about an object's IDL interface at runtime, it needs an alternative, *dynamic* approach to CORBA programming.

The CORBA *Interface Repository* is a database that stores information about the IDL interfaces implemented by objects in your network. A client program can query this database at runtime to get information about those interfaces. The client can then call member functions on objects using a component of the ORB called the *Dynamic Invocation Interface* (DII), as shown in Figure 3 on page 6.

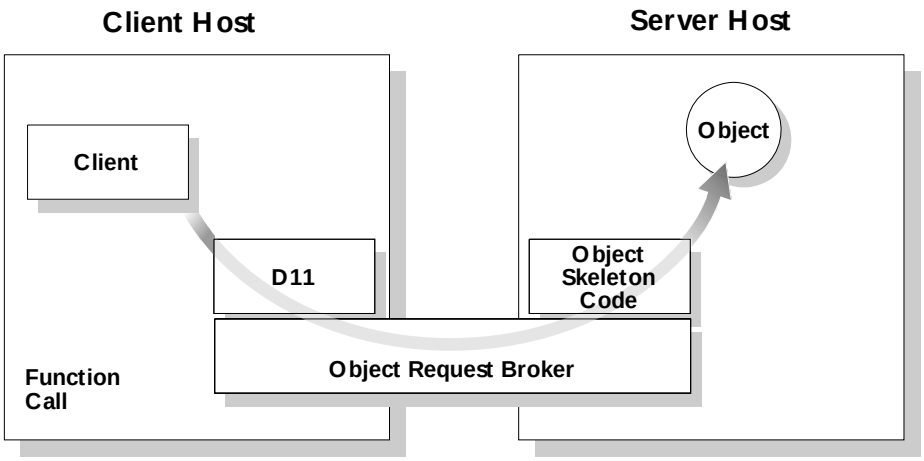


Figure 3: *Client Invoking a Function Using the DII*

CORBA also supports dynamic server programming. A CORBA program can receive function calls through IDL interfaces for which no CORBA object exists. Using an ORB component called the *Dynamic Skeleton Interface* (DSI), the server can then examine the structure of these function calls and implement them at runtime. Figure 4 on page 7 shows a dynamic client program communicating with a dynamic server implementation.

Interoperability between Object Request Brokers

The components of an ORB make the distribution of programs transparent to network programmers. To achieve this, the ORB components must communicate with each other across the network.

In many networks, several ORB implementations coexist and programs developed with one ORB implementation must communicate with those developed with another. To ensure that this happens, CORBA specifies that ORB components must communicate using a standard network protocol, called the *Internet Inter-ORB Protocol* (IIOP).

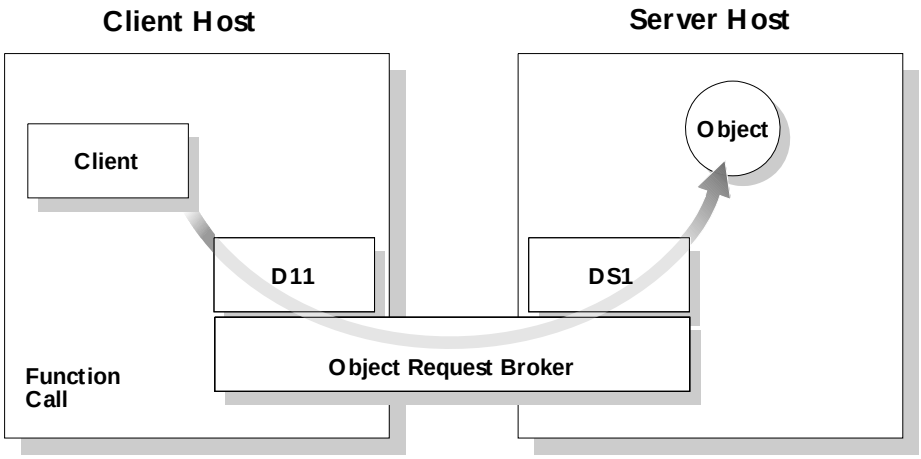


Figure 4: *Function Call Using the DII and DSI*

The Object Management Architecture

An ORB is one component of the OMG's Object Management Architecture (OMA). This architecture defines a framework for communications between distributed objects.

As shown in Figure 5 on page 8, the OMA includes four elements:

- Application objects.
- The ORB.
- The *CORBAservices*.
- The *CORBAfacilities*.

Application objects are objects that implement programmer-defined IDL interfaces. These objects communicate with each other, and with the CORBAservices and CORBAfacilities, through the ORB. The CORBAservices and CORBAfacilities are sets of objects that implement IDL interfaces defined by CORBA and provide useful services for some distributed applications.

When writing Orbix applications, you may require one or more CORBAservices or CORBAfacilities. This section provides a brief overview of these components of the OMA.

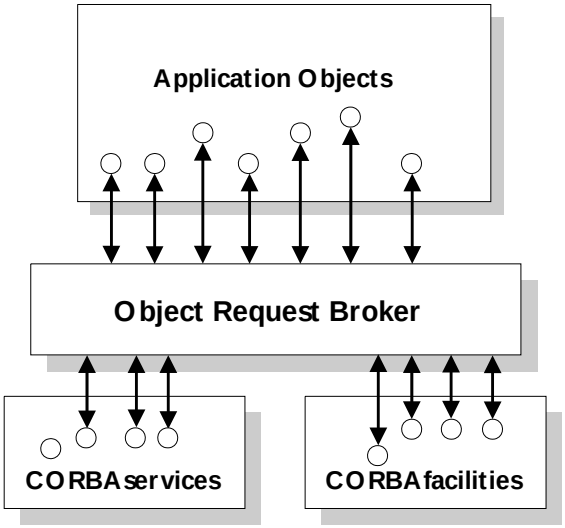


Figure 5: *The Object Management Architecture*

The CORBAservices

The CORBAservices define a set of low-level services that allow application objects to communicate in a standard way. These services include the following:

- The *Naming Service*. Before using a CORBA object, a client program must get an identifier for the object, known as an *object reference*. This service allows a client to locate object references based on abstract, programmer-defined object names.
- The *Trader Service*. This service allows a client to locate object references based on the desired properties of an object.
- The *Security Service*. This service allows CORBA programs to interact using secure communications.

Orbix 3 implements several CORBAservices including all the services listed above.

The CORBAfacilities

The CORBAfacilities define a set of high-level services that applications frequently require when manipulating distributed objects. The CORBAfacilities are divided into two categories:

- The *horizontal* CORBAfacilities.
- The *vertical* CORBAfacilities.

The horizontal CORBAfacilities consist of user interface, information management, systems management, and task management facilities. The vertical CORBAfacilities standardize IDL specifications for market sectors such as healthcare and telecommunications.

How Orbix Implements CORBA

Orbix is an ORB that fully implements the CORBA 2 specification. By default, all Orbix components and applications communicate using the CORBA standard IIOP protocol.

The components of Orbix are as follows:

- The *IDL compiler* parses IDL definitions and produces C++ code that allows you to develop client and server programs.
- The *Orbix library* is linked against every Orbix program and implements several components of the ORB, including the DII, the DSI, and the core ORB functionality.
- The *Orbix daemon* is a process that runs on each server host and implements several ORB components, including the Implementation Repository.
- The *Orbix Interface Repository server* is a process that implements the Interface Repository.

Orbix also includes several programming features that extend the capabilities of the ORB.

In addition, Orbix is an enterprise ORB that combines the functionality of the core CORBA standard with an integrated suite of services including OrbixNames and OrbixSSL. This chapter introduces the architecture of Orbix and briefly describes each of these services.

Note: Only an overview of these components is given here. For more detailed descriptions of functionality, refer to the individual programming guides and reference guides that accompany each component.

Orbix Components

Table 1 gives a brief synopsis of the Orbix suite.

Table 1: *The Orbix Suite*

Orbix	The multithreaded Orbix Object Request Broker (ORB) is at the heart of Orbix. This is a Micro Focus implementation of the OMG (Object Management Group) CORBA specification.
OrbixSSL	OrbixSSL integrates Orbix with Secure Socket Layer (SSL) security. Using OrbixSSL, distributed applications can securely transfer confidential data across a network. OrbixSSL offers CORBA level zero security.
OrbixNames	OrbixNames maintains a repository of mappings that associate objects with recognisable names. This is Micro Focus's implementation of the OMG CORBAservices Naming Service.

Orbix Architecture

The overall architecture of Orbix and its components is shown in Figure 6. On the lower part of Figure 6, a number of CORBA servers and clients are shown attached to an intranet and, on the top left, a sample client is shown attached to the system via the Internet. It is necessary to pencil in a number of server hosts in this basic illustration because Orbix is an intrinsically distributed system. In contrast to the star-shaped architecture of many traditional systems, with clients attached to a central monolithic server, the architecture of Orbix is based on a collection of components cooperating across a number of hosts.

Some standard services, such as the CORBA Naming Service (OrbixNames) are implemented as clearly identifiable processes with an associated executable. There can be many instances of these processes running on one or more machines.

Other services rely on cooperation between components. They are, either wholly or partly, based on libraries linked with each component. Services such as this are intrinsically distributed.

Since Orbix has an open, standards-based architecture it can readily be extended to integrate with other CORBA-based products. In particular, as Figure 6 shows, integration with a mainframe is possible when Orbix is combined with an ORB running on OS/390.

For more information on Orbix, see the ***Orbix Programmer's Guide C++ Edition***, ***Orbix Programmer's Reference C++ Edition*** and ***Orbix Administrator's Guide C++ Edition***.

In the remainder of this section on Orbix architecture, each of the components of Orbix will be presented with a brief description of the main features.

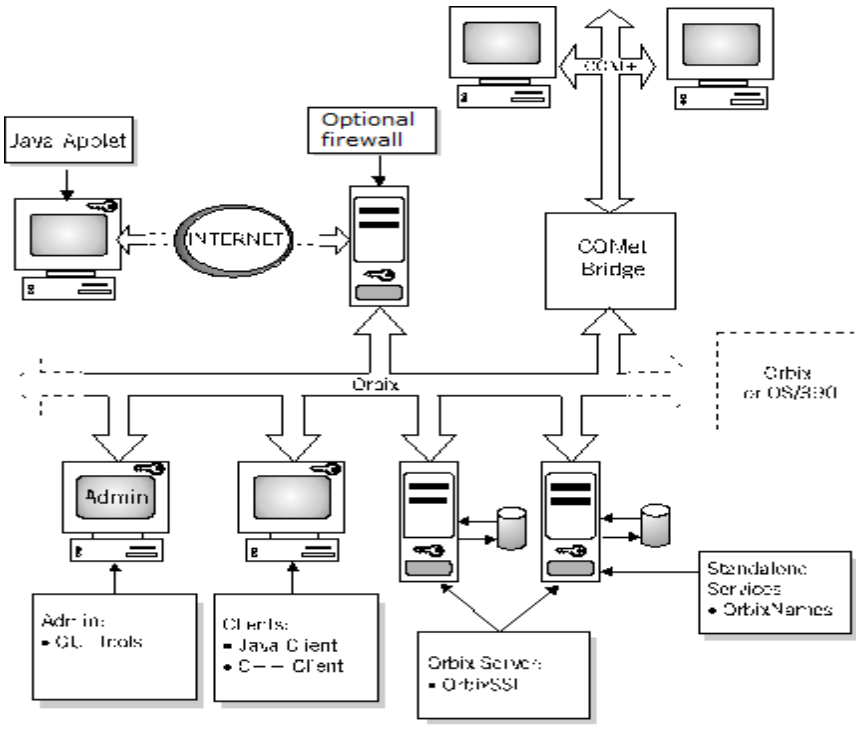


Figure 6: *The Orbix Architecture*

OrbixNames—The Naming Service

OrbixNames is Micro Focus's implementation of the CORBA Naming Service. The role of OrbixNames is to allow a name to be associated with an object and to allow that object to be found using that name. A server that holds an object can register it with OrbixNames, giving it a name that can be used by other components of the system to subsequently find the object. OrbixNames maintains a repository of mappings (*bindings*) between names and object references. OrbixNames provides operations to do the following:

- Resolve a name.
- Create new bindings.
- Delete existing bindings.
- List the bound names.

Using a Naming Service such as OrbixNames to locate objects allows developers to hide a server application's location details from the client. This facilitates the invisible relocation of a service to another host. The entire process is hidden from the client.

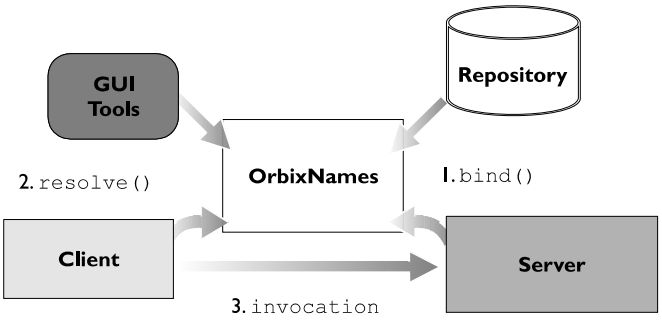


Figure 7: *The OrbixNames Architecture*

Figure 7 summarizes the functionality of OrbixNames, which is as follows:

1. A server registers object references in OrbixNames. OrbixNames then maps these object references to names.
2. Clients resolve names in OrbixNames.
3. Clients remotely invoke on object references in the server.

OrbixNames, which runs as an Orbix server, has a number of interfaces defined in IDL that allow the components of the system to use its facilities. Other features of OrbixNames include an enhanced GUI browser interface. OrbixNames can support clients that use either IIOP or the Orbix protocol.

For more information on OrbixNames, see the ***OrbixNames Programmer's and Administrator's Guide***.

Security with OrbixSSL

OrbixSSL introduces Level 0 CORBA security, as specified by the OMG, to the Orbix product suite. Level 0 corresponds to the provision of authentication and session encryption, which maps onto the functionality provided by the Secure Socket Layer (SSL) library.

SSL is a protocol for providing data security for applications that communicate across networks via TCP/IP. By default, Orbix applications communicate using the standard CORBA Internet Inter-ORB Protocol (IIOP). These application-level protocols are layered above the transport-level protocol TCP/IP.

OrbixSSL provides authentication, privacy, and integrity for communications across TCP/IP connections as follows:

Authentication	Allows an application to verify the identity of another application with which it communicates.
Privacy	Ensures that data transmitted between applications can not be understood by a third party.
Integrity	Allows applications to detect whether data was modified during transmission.

To initiate a TCP/IP connection, OrbixSSL provides a security ‘handshake’. This handshake results in the client and server agreeing on an ‘on the wire’ encryption algorithm, and also fulfils any authentication requirements for the connection. Thereafter, OrbixSSL’s only role is to encrypt and decrypt the byte stream between client and server.

The steps involved in establishing an OrbixSSL connection are as follows:

1. The client initiates a connection by contacting the server.
2. The server sends an X.509 certificate to the client. This certificate includes the server’s public encryption key.
3. The client authenticates the server’s certificate (for example, an X.509 certificate, endorsed by an accredited certifying authority).
4. The client sends the certificate to the server for authentication.
5. The server generates a session encryption key and sends it to the client encrypted using the client’s public key: the session is now established.

Once the connection has been established, certain data is cached so that in the event of a dropping and resumption of the dialogue, the handshake is curtailed and connection re-establishment is accelerated.

For more information on OrbixSSL, see ***OrbixSSL Programmer’s and Administrator’s Guide C++ Edition***.

Developing Applications with Orbix

The chapter describes how to develop a distributed application using Orbix. An example application illustrates the steps involved in the development process. These include defining an IDL interface, implementing this interface in C++, and developing a C++ client application.

This chapter describes the basic programming steps required to create Orbix objects, write server programs that expose those objects, and write client programs that access those objects.

This chapter illustrates the programming steps using an example named `BankSimple`. In this example, an Orbix server program implements two types of objects: a single object implementing the `Bank` interface, and multiple objects implementing the `Account` interface. A client program uses these clearly-defined object interfaces to create and find accounts, and to deposit and withdraw money.

On both Windows and UNIX, the source code for the example described in this chapter is available in the `demos\banksimple` directory of your Orbix installation.

Developing a Distributed Application

To develop an Orbix application, you must perform the following steps:

1. Identify the objects required in your system and define public interfaces to those objects using CORBA Interface Definition Language (IDL).
2. Compile the IDL interfaces.
3. Implement the IDL interfaces using C++ classes.
4. Write a server program that creates instances of the implementation classes.
5. Write a client program that accesses the server object.
6. Compile the client and server.
7. Run the application

Defining IDL Interfaces

Defining IDL interfaces to your objects is the most important step in developing an Orbix application. These interfaces define how clients access objects regardless of the location of those objects on the network.

An interface definition contains *attributes* and *operations*. Attributes allow clients to get and set values on the object. Operations are functions that clients can call on an object.

For example, the following IDL from the `BankSimple` example defines two interfaces for objects representing a bank application. The interfaces are defined inside an IDL module to prevent clashes with similarly-named interfaces defined in subsequent examples.

The interfaces to the `BankSimple` example are defined in IDL as follows:

```
    // IDL
    // In file banksimple.idl

1   module BankSimple {

        typedef float CashAmount;

2           interface Account;
3           interface Bank {
            Account create_account (in string name);
            Account find_account (in string name);
            };

4           interface Account {
            readonly attribute string name;
            readonly attribute CashAmount balance;

5               void deposit (in CashAmount amount);
            void withdraw (in CashAmount amount);
            };
    };
```

This code is explained as follows:

1. An IDL module is equivalent to a C++ namespace, and groups the definitions into a common namespace. Using a module is not mandatory, but is good practice.
2. This is a forward declaration to the `Account` interface. It allows you to refer to `Account` in the `Bank` interface, before actually defining `Account`.
3. The `Bank` interface contains two operations: `create_account()` and `find_account()`, allowing a client to create and search for an account.
4. The `Account` interface contains two attributes: `name` and `balance`; both are *readonly*. This means that clients can get the balance or name, but cannot directly set them. If the `readonly` keyword is omitted, clients can also set these values.
5. The `Account` interface also contains two operations: `deposit()` and `withdraw()`. The `deposit()` operation allows a client to deposit money in the account. The `withdraw()` operation allows a client to withdraw money from the account.

The parameters to these operations are labeled with the IDL keyword `in`. This means that their values are passed from the client to the object. Operation parameters can be labeled as `in`, `out` (passed from the object to the client) or `inout` (passed in both directions).

Compiling IDL Interfaces

You must compile IDL definitions using the Orbix IDL compiler. Before running the IDL compiler, ensure that your configuration is correct.

Setting Up Configuration for the IDL Compiler

You should ensure that the environment variable `IT_CONFIG_PATH` is set to the location of `iona.cfg`, the root Orbix configuration file.

UNIX

On UNIX, if `iona.cfg` is in directory `/local/microfocus/orbix33`, perform the following steps:

1. Under `sh enter`:

   ```
   % IT_CONFIG_PATH=/local/microfocus/orbix33
   % export IT_CONFIG_PATH
   ```

   or under `csh` enter:

   ```
   % setenv IT_CONFIG_PATH /local/microfocus/orbix33
   ```

2. Set the environment variable `LD_LIBRARY_PATH` to include the location of the Orbix `lib` directory in a similar manner.

Windows

On Windows, if `iona.config` is in directory `C:\Micro Focus\Orbix 3.3\config`, enter the following at a command prompt:

```
set IT CONFIG PATH = C:\Micro Focus\Orbix 3.3\config
```

Running the IDL Compiler

The IDL compiler checks the validity of the specification and generates C++ code that allows you to write client and server programs.

Windows and UNIX

To compile the `Bank` and `Account` interfaces defined in file `banksimple.idl`, run the IDL compiler as follows:

```
idl [options] banksimple.idl
```

The `-B` compiler option produces BOAImpl classes for the server. Refer to "Orbix IDL Compiler Options" for a complete list of IDL compiler options.

Output from the IDL Compiler

The IDL compiler produces three C++ files that communicate with Orbix:

1. A common header file containing declarations used by both client and server mode. This header file should be included in all client and server programs.
2. A source file to be compiled and linked with servers (*object skeleton code*).
3. A source file to be compiled and linked with clients (*client stub code*).

These source files contain C++ definitions that correspond to your IDL definitions. These C++ definitions allow you to write C++ client and server programs.

By default, these files are named as follows:

File	Windows	UNIX
Header file	banksimple.hh	banksimple.hh
Client stub code	banksimpleC.cpp	banksimpleC.C
Server skeleton code	banksimpleS.cpp	banksimpleS.C

The Client Stub Code

The files banksimple.hh and banksimple.client.cxx define the C++ code that a client uses to access a Bank object. This code is termed the client stub code. For example, the banksimple.hh file for the BankSimple IDL includes a class to represent Bank and Account objects from a client's point of view.

The IDL declarations for the Account interface include the C++ definitions in the following code extract:

```
// C++
// In file banksimple.hh

// Automatically generated by the IDL compiler.
class Account: public virtual CORBA::Object  {
public:
    // CORBA support functions and error handling are
    // omitted here for clarity
    virtual char*  name ()
        throw (CORBA::SystemException);
    virtual CashAmount balance ()
        throw (CORBA::SystemException);
    virtual void deposit (CashAmount amount)
        throw (CORBA::SystemException);
    virtual void withdraw (CashAmount amount)
        throw (CORBA::SystemException);
};
```

The environment argument (the last argument passed to each method) is omitted here.

This class represents the IDL Account interface in C++ allowing C++ clients to treat Account objects like any other C++ object. The readonly name and balance attributes map to member functions of the same name. The deposit() and withdraw() operations map to C++ member functions with equivalent parameters.

The Object Skeleton Code

The files `banksimple.hh` and `banksimple.server.cxx` define the C++ code that allows a server program to implement IDL interfaces and accept operation calls from clients to objects. This code is known as the object skeleton code. These server-side skeletons receive CORBA calls and pass them onto application code. When implementing a server using the *BOAImpl* approach, you inherit from a BOAImpl class generated by the IDL compiler.

For the `Account` interface the BOAImpl class includes the following C++ definitions:

```
// C++
// In file banksimple.hh

// Automatically generated by IDL compiler.
class AccountBOAImpl: public virtual Account  {
public:
    virtual char*  name ()
        throw (CORBA::SystemException) = 0;
    virtual CashAmount balance ()
        throw (CORBA::SystemException) = 0;
    virtual void deposit (CashAmount amount)
        throw (CORBA::SystemException) = 0;
    virtual void withdraw(CashAmount amount)
        throw (CORBA::SystemException) = 0;
    };
```

To implement the `Account` interface, you must inherit from this class and override the pure virtual functions that represent IDL operations with application code.

Implementing the IDL Interfaces

This example uses the CORBA BOAImpl approach to implementing an IDL interface. It uses two classes to implement the `Bank` and `Account` IDL interfaces in C++: `BankSimple_BankImpl` and `BankSimple_AccountImpl`. These classes inherit the IDL compiler-generated `BankSimple::BankBOAImpl` and `BankSimple::AccountBOAImpl` classes. These base classes provide all the Orbix functionality. All that remains is to override the abstract member functions that represent the IDL operations.

For example, the code for `BankSimple_BankImpl` is as follows:

```
// C++
// In file BankSimple\banksimple_bankimpl.h
// Implementation class for the Bank IDL interface.
...
1   class BankSimple_BankImpl : public virtual
        BankSimple::BankBOAImpl
    {
    public:
        // Mapped IDL operations.
2       virtual BankSimple::Account_ptr
            create_account(const char*  name,
  CORBA::Environment&);
        virtual BankSimple::Account_ptr
            find_account( const char*  name,
  CORBA::Environment&);
```

```
            // C++ constructor and destructor.
3           BankSimple_BankImpl();
            virtual ~BankSimple_BankImpl();

        protected:
            static const int MAX_ACCOUNTS;
4           BankSimple::Account_var*  m_accounts;
};
```

This code is explained as follows:

1. Inheriting from the BOAImpl class generated by the IDL compiler provides Orbix functionality for the server objects.
2. Operations defined in IDL are implemented by corresponding operations in C++. The IDL `Account` type is represented by an `Account_ptr`.
3. The constructor and destructor are normal C++ functions that can be called by server code. Only IDL functions can be called remotely by clients.
4. The accounts created by the bank are stored in an array of `Account_var`. These are like pointers; for more information on `Account_var`, refer to "CORBA Object References".

You can implement the member functions of `BankSimple_BankImpl` as follows:

```
// C++
// In file banksimple_bankimpl.cxx

#include "banksimple_bankimpl.h"
#include "banksimple_accountimpl.h"

1   const int BankSimple_BankImpl::MAX_ACCOUNTS = 1000;
BankSimple_BankImpl::BankSimple_BankImpl() :
m_accounts(new BankSimple::Account_var[MAX_ACCOUNTS]) {
// Make sure all accounts are nil.
    for (int i = 0; i < MAX_ACCOUNTS; ++i){
        m_accounts[i] = BankSimple::Account::_nil();
    }
}

BankSimple_BankImpl::~BankSimple_BankImpl() {
    delete [] m_accounts;
}

// Add a new account.
BankSimple::Account_ptr BankSimple_BankImpl::create_account
(const char*  name, CORBA::Environment& ) {

    int i = 0;
    for ( ; i < MAX_ACCOUNTS && !CORBA::is_nil(m_accounts[i]);
                ++i)
    {}
    if (i < MAX_ACCOUNTS){
2           m_accounts[i] = new BankSimple_AccountImpl(name, 0.0);
            cout << "create_account: Created account with name: "
                 << name << endl;
3           return BankSimple::Account::_duplicate(m_accounts[i]);
    }
    else{
        cout << "create_account: failed, no space left!" << endl;
```

```
4           return BankSimple::Account::_nil();
        }
    }

    // Find a named account.
    BankSimple::Account_ptr BankSimple_BankImpl::find_account
    (const char*  name, CORBA::Environment& ) {

        int i = 0;
        for (  ; i < MAX_ACCOUNTS &&( CORBA::is_nil(m_accounts[i]) ||
                strcmp(name, m_accounts[i]->name()) != 0); ++i)
        { }

        if (i < MAX_ACCOUNTS){
            cout << "find_account: found account named" << name <<
       endl;
            return BankSimple::Account::_duplicate(m_accounts[i]);
        }
        else{
            cout << "find_account: no account named" << name << endl;
            return BankSimple::Account::_nil();
        }
    }
```

The code is explained as follows:

1. The maximum number of accounts that the bank can handle in this simple implementation is set as a constant of `1000`.
2. New accounts are created with a balance of zero.
3. When an `Account` reference is returned from `create_account()` and `find_account()` operations, it must be duplicated. According to CORBA memory management rules, this reference is released by the caller.
4. If an account cannot be created, nil is returned.

Refer to the `banksimple\demos` directory of your Orbix installation for the corresponding code for `BankSimple_AccountImpl`.

Writing an Orbix Server Application

To write a C++ program that acts as an Orbix server, perform the following steps:

1. Initialize the server connection to the Orbix ORB, and to the Basic Object Adapter (BOA).
2. Create an implementation object. This is done by creating instances of the implementation classes.
3. Allow Orbix to receive and process incoming requests from clients.

This section describes each of these programming steps in turn.

Initializing the ORB

Because Orbix uses the standard OMG IDL to C++ mapping, all servers and clients must call `CORBA::ORB_init()` to initialize the ORB. This returns a reference to the ORB object. The ORB methods defined by the standard can then be invoked on this instance.

```
// C++
// In file server.cxx
...
try {
    ...
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc,argv,"Orbix");
    ...
}
catch (const CORBA::SystemException& e) {
    cout << "Unexpected exception" << e << endl;
}
```

In this code sample, the `argc` parameter refers to the number of arguments in `argv`. The `argv` parameter is a sequence of configuration strings used if `"Orbix"` is a null string; the string `"Orbix"` identifies the ORB. Refer to the *Orbix Reference Guide* for more information on `CORBA::ORB_init()`.

Orbix raises a C++ exception to indicate that a function call has failed. All CORBA exceptions derive from `CORBA::Exception`. Many Orbix functions (for example, `ORB_init()`) and all IDL operations may raise a CORBA system exception, of type `CORBA::SystemException`.

You must use C++ `try/catch` statements to handle exceptions, as illustrated in the preceding code sample. In the remainder of this chapter, `try/catch` statements are omitted for clarity.

Creating an Implementation Object

To create an implementation object, you must create an instance of your implementation class in your server program. Typically a server program creates a small number of objects in its `main()` function, and these objects may in turn create further objects. In the `BankSimple` example, the server creates a single bank object in its `main()` function. This bank object then creates accounts when `create_account()` is called by the client.

For example, to create an instance of `BankSimple::Bank` in your server `main()` function, do the following:

```
// C++
// In file server.cxx

#include "banksimple_bankimpl.h"
int main ( ... ) {
...
// Create a bank implementation object.
BankSimple::Bank_var my_bank = new BankSimple_BankImpl;
    ...
}
```

A server program can create any number of implementation objects for any number of IDL interfaces.

Note that implementation object has a name that uniquely identifies it to the server. This name is called the "marker" (discussed more in "Making Objects Available in Orbix"). The above code does not explicitly set the marker for the Bank implementation object, hence the ORB picks an unused random name. In general, you always need to explicitly set the marker from your implementation objects (see "Making Objects Available in Orbix").

Receiving Client Requests

When a server instantiates an Orbix object (for example, one inheriting from the BOAImpl class), it is automatically registered with Orbix as a distributed object. To make objects available to clients, the server must call the Orbix function `CORBA::BOA::impl_is_ready()` to complete its initialization and to process operation calls from clients.

You can code a complete server `main()` function as follows:

```
// C++
// In file server.cxx

#include "banksimple_bankImpl.h"
#include "banksimple_accountImpl.h"
#include <it_demo_nsw.h>

// Server mainline.
int main (int argc, char*  argv[]) {
    try {
        // Use standard demo server options.
1       IT_Demo_ServerOptions
            serveropt("IT_Demo/BankSimple/Bank");
        ...
2           CORBA::ORB_var orb = CORBA::ORB_init(argc, argv,
          "Orbix");
           CORBA::BOA_var boa = orb->BOA_init(argc, argv,
       "Orbix_BOA");

            // Set diagnostics.
            orb->setDiagnostics(serveropt.diagnostics());

            // Set server name.
3           orb->setServerName(serveropt.server_name());

4           // Indicate server should not quit while clients
            // are connected.
            boa->setNoHangup(1);
            // Set up Naming Service Wrappers (NSW).
5           IT_Demo_NSW ns_wrapper;
6           ns_wrapper.setNamePrefix(serveropt.context());
7           const char* bank_name = "BankSimple.Bank";
            ...
            // Create a bank implementation object.
8           BankSimple::Bank_var my_bank = new BankSimple_BankImpl;

9           // Register server object with the Naming Service.
```

```
            if (serveropt.bindns()) {
                cout << "Binding objects in the Naming Service"
                 << endl;
                ns_wrapper.registerObject(bank_name, my_bank);
            }

            // Server has completed initialization, wait for
            // incoming requests.
            boa->impl_is_ready( (char*)serveropt.server_name(),
                serveropt.timeout());

            // impl_is_ready() returns only when Orbix times-out
            // an idle server.
            cout << "server exiting" << endl;
        }
        catch (const CORBA::Exception& e) {
            cerr << "Unexpected exception" << e << endl;
            return 1;
        }
        return 0;
    };
```

This code is explained as follows:

1. Create the standard server options for use throughout the demonstration and set the server name to `IT_Demo/BankSimple/Bank`. The Orbix `demos\demolib` directory contains the standard server and client options used by the Bank series examples in this book.
2. Initialize the ORB and BOA. The ORB object provides functionality common to both clients and servers. The BOA (Basic Object Adapter) object is derived from the ORB and provides additional server-side functionality.

   The ORB and the BOA are different views of the same ORB API—this object is also available via the global variable `CORBA::Orbix`. However, use of this variable is not CORBA-defined and is discouraged.
3. Set the server name using `setServerName(serveropt.server_name())`. This is required by Orbix before exporting object references.
4. Create a *Naming Service Wrapper* (NSW) object. To simplify the use of the Naming Service, a Naming Service Wrapper is provided. This hides the low-level detail of the CORBA Naming Service. Refer to "Using the Naming Service in Orbix Example Applications" for details of the Naming Service wrapper functions.
5. Define a name prefix that is used for subsequent operations.
6. `BankSimple.Bank` is the name that the bank object is known by in the Naming Service.
7. The created `BankSimple` instance is `my_bank`. This object implements an instance of the IDL interface `Bank`. This is called directly from client applications using the CORBA standard Internet Inter-ORB Protocol (IIOP).
8. The server now registers its objects in the Naming Service using the Naming Service wrapper function `registerObject()`.
9. The `CORBA::BOA::impl_is_ready()` operation is called to complete server initialization. This takes a server name and a timeout value as parameters. You can specify any name for

your server; however, the name should match the name used to register the server in the Implementation Repository, and the argument used to call `setServerName()`.

The timeout value indicates the period of time, in milliseconds, that the `impl_is_ready()` call should block for while waiting for an operation call to arrive from a client. If no call arrives in this period, `impl_is_ready()` returns. If a call arrives, Orbix calls the appropriate member function on the implementation object and the timeout counter starts again from zero.

Writing an Orbix Client Application

To write a C++ client program to an Orbix object, you must perform the following steps:

1. Initialize the client connection to the ORB.
2. Get a *reference* to an object.
3. Invoke attributes and operations defined in the object's IDL interface.

This section describes each of these steps in turn.

Initializing the ORB

All clients and servers must call `CORBA::ORB_init()` to initialize the ORB. This returns a reference to the ORB object. The ORB methods defined by the standard can then be invoked on this instance.

CORBA Object References

A CORBA object reference identifies an object in your system. When an object reference enters a client address space, Orbix creates a *proxy* object that acts as a local representative for the remote implementation object. Orbix forwards operation invocations on the proxy object to corresponding functions in the implementation object.

Consider an object reference as a pointer that can point to an object in a remote server process. Object references to an object of interface `X` are represented by a type `X_ptr`, which behaves like a normal C++ pointer.

An object reference requires some memory in the client (the memory needed by the proxy object), so you must release each reference when finished by calling `CORBA::release()`. The `CORBA::release()` method releases the client memory used by the object reference—it does not affect the remote server object.

For interface `X`, the IDL compiler also generates a smart pointer class called `X_var` that automates memory management. `X_var` behaves just like `X_ptr`, except it releases the reference when it goes out of scope, or if a new reference is assigned.

Getting a Reference to an Object

The flexible CORBA-defined way to obtain object references is to use the standard CORBA Naming Service. The CORBA Naming Service allows a name to be *bound* to an object and allows that object to be found subsequently by *resolving* that name within the Naming Service.

A server that holds an object reference can register it with the Naming Service, giving it a name that can be used by other components of the system to find the object. The Naming Service maintains a database of *bindings* between names and object references. A binding is an association between a name and an object reference. Clients can call the Naming Service to resolve a name, and this returns the object reference bound to that name. The Naming Service provides operations to resolve a name, to create new bindings, to delete existing bindings, and to list the bound names.

A name is always resolved within a given naming context. The naming context objects in the system are organized into a graph, which may form a naming hierarchy, much like that of a file system. The following sample code shows how the client uses the Naming Service wrapper functions to obtain an object reference:

```
    // C++
    // In file client.cxx
    ...
    // Naming Service Setup.
    // Create a Naming Service Wrapper object.
        IT_Demo_NSW ns_wrapper;
1     ns_wrapper.setNamePrefix(clientopt.context());

    // Get CORBA object.
    // Specify the object name in the Naming Service.
2       const char* object_name = "BankSimple.Bank";

    // Get a reference to the required object from the NSW.
3     CORBA::Object_var obj = ns_wrapper.resolveName(object_name);

    // Narrow the object reference.
4     BankSimple::Bank_var bank = BankSimple::Bank::_narrow(obj);
        if (CORBA::is_nil(bnk)) {
        cerr << "Object \"" << object_name
                << "\"in the Naming Service" << endl
                << "\tis not of the expected type."<< endl;
        return 1;
    }

    // Start client menu loop
5       BankMenu main_menu(bank);
            main_menu.start);
    }
    ...
}
```

This code is described as follows:

1. Define a name prefix used by the Naming Service wrapper object for subsequent operations.

2. `BankSimple.Bank` is the name by which the bank object is known in the Naming Service.
3. The method `nswrapper::resolveName()` retrieves the object reference from the Naming Service placed there by servers. The `object_name` parameter is the name of the object to resolve. This must match the name used by the server when it calls `registerObject()`.
4. The return type from `resolveName()` is of type `CORBA::Object`. You must call `_narrow()` to safely cast down from the base class to the `Bank` IDL class, before you can make invocations on remote `Bank` objects. The client stub code generated for every IDL class contains the `_narrow()` function definition for that class.
5. This creates and runs a main menu for `Bank` clients. This menu enables you to find or create accounts by calling the appropriate C++ member function on the object reference.

Invoking IDL Attributes and Operations

To access an attribute or an operation associated with an object, call the appropriate C++ member function on the object reference. The client-side proxy redirects this C++ call across the network to the appropriate member function of the implementation object.

The main `BankSimple` client program calls a simple interactive menu. This enables you to call IDL operations on a `Bank`. The following code extracts show the code called when you choose to create or find an account:

```
// C++
// In file bankmenu.cxx

void BankMenu::do_create() throw(CORBA::SystemException) {

    cout << "Enter account name: " << flush;
    CORBA::String_var name = IT_Demo_Menu::get_string();

1     BankSimple::Account_var account =
       m_bank->create_account(name);

    // Start a sub-menu with the returned account ref.
    AccountMenu sub_menu(account);
    sub_menu.start();
}

// do_find -- calls find account and runs account menu.

void BankMenu::do_find throw (CORBA::SystemException) {

    cout << "Enter account name: " << flush;
2       CORBA::String_var name = IT_Demo_Menu::get_string();

    BankSimple::Account_var account =
   m_bank->find_account(name);
        AccountMenu sub_menu(account)
    sub_menu.start();
}
```

This code is explained as follows:

1. `m_bank` is a `Bank_var`—a C++ helper class automatically generated by the IDL compiler from the `Bank` interface. This is used like a normal C++ pointer to call IDL operations just like C++ operations.
2. The `String_var name` variable is used for the account name entered. The caller is not responsible for releasing the memory—`String_var` automatically does this when it goes out of scope.

Use the C++ arrow operator (`->`) to access the operations defined in IDL through a `BankSimple::Bank_var` object. Call those member functions using normal C++ calls and test for errors using C++ exception handling.

Compiling the Client and Server

To build the client and server, you must compile and link the relevant C++ files with the Orbix library. On UNIX, this is `liborbix`; on Windows, this is `ITMi.lib`. These files are available in the Orbix `lib` directory.

Note: For demonstration-specific functionality, you must also include `libdemo.a` on UNIX and `demolib.lib` on Windows.

Compiling the Client

To build the client application, compile and link the following C++ files, and the Orbix library:

- `banksimple.client.cxx`
- `client.cxx`
- `bankmenu.cxx`
- `accountmenu.cxx`

`client.cxx` is the source file for the client `main()` function.

Compiling the Server

To build the server application, compile and link the following C++ files, and the Orbix library.

- `banksimple.server.cxx`
- `banksimple_bankimpl.cxx`
- `banksimple_accountimpl.cxx`
- `server.cxx`

`server.cxx` is the source file for the server `main()` function.

The Orbix `demos/banksimple` directory includes a *makefile* that compiles and links the bank client and server demonstration code.

To build the executables, type one of the following in the `demos\banksimple` directory of your Orbix installation:

Windows	`>nmake`
UNIX	`%make`

Running the Application

To run the application, do the following:

1. Run the Orbix daemon process (`orbixd`) on the server host.
2. Register the server in the Orbix Implementation Repository.
3. Run the client program.

Running the Orbix Daemon

Before a client can access a server, the server must be registered with the Orbix daemon. Before running the Orbix daemon, ensure that the environment variable `IT_CONFIG_PATH` is set as described in "Setting Up Configuration for the IDL Compiler" on page 17.

Windows and UNIX

You can run the Orbix daemon on the server host by typing `orbixd` at the command line or using the **Start** menu on Windows.

Registering the Server

The Implementation Repository is the component of Orbix that stores information about servers available in the system. Before running your application, you must register your server in the Implementation Repository.

Windows and UNIX and OpenVMS

To register the server(s), use either the Server Manager GUI tool or run the Orbix `putit` command on the server host as follows:

```
putit server_name server_executable
```

On all platforms, `server_name` is the name of your server passed to `impl_is_ready()`.

If a server binds names in the Naming Service, you may need to run it once to allow it to set up the name bindings. Details of how to do this depend on the server used. The demonstrations provide a makefile that do the necessary server registration and set up names in the Naming Service.

To register the server, type one of the following:

Windows	`> nmake register`
UNIX	`% make register`

Running the Client

When a client binds to an object in a server registered in the Implementation Repository, the Orbix daemon automatically launches the server executable file. Consequently, you can run the client without running the server in advance.

Before running the client, ensure that the environment variable `IT_CONFIG_PATH` is set as described in "Setting Up Configuration for the IDL Compiler" on page 17.

Windows and UNIX

Run the example client by entering `client` at the command-line prompt. The client displays a text menu allowing you to choose the actions you want to take, and then prompts you for the necessary information. The server outputs messages when it processes incoming calls. You can see these messages by looking at the application shell window launched by the Orbix daemon.

Summary of Programming Steps

To develop a distributed application with Orbix, do the following:

1. Identify the objects required in your system and define the public interfaces to those objects using the CORBA Interface Definition Language (IDL).
2. Compile the IDL interfaces.
3. Implement the IDL interfaces with C++ classes.
4. Write a server program that creates instances of the implementation classes. This involves:
   i. Initializing the ORB.
   ii. Creating initial implementation objects.
   iii. Allowing Orbix to receive and process incoming requests from clients.
5. Write a client program that accesses the server objects. This involves:
   i. Initializing the ORB.
   ii. Getting a reference to an object.
   iii. Invoking object attributes and operations.
6. Compile the client and server.
7. Run the application. This involves:
   i. Running the Orbix daemon process.
   ii. Registering the server in the Implementation Repository.
   iii. Running the client.

Part II

Orbix C++ Programming

In this part

This part contains the following:

Introduction to CORBA IDL

The CORBA Interface Definition Language (IDL) is used to define interfaces to objects in your network. This chapter introduces the features of CORBA IDL and illustrates the syntax used to describe interfaces.

The first step in developing a CORBA application is to define the interfaces to the objects required in your distributed system. To define these interfaces, you use CORBA IDL.

IDL allows you to define interfaces to objects without specifying the implementation of those interfaces. To implement an IDL interface, you define a C++ class that can be accessed through that interface and then you create objects of that class within an Orbix server application.

In fact, you can implement IDL interfaces using any programming language for which an IDL mapping is available. An IDL mapping specifies how an interface defined in IDL corresponds to an implementation defined in a programming language. CORBA applications written in different programming languages are fully interoperable.

CORBA defines standard mappings from IDL to several programming languages, including C++, Java, and Smalltalk. The Orbix IDL compiler converts IDL definitions to corresponding C++ definitions, in accordance with the standard IDL to C++ mapping.

IDL Modules and Scoping

An IDL module defines a naming scope for a set of IDL definitions. Modules allow you to group interface and other IDL type definitions in logical name spaces. When writing IDL definitions, always use modules to avoid possible name clashes.

The following example illustrates the use of modules in IDL:

```
// IDL
module BankSimple {

    interface Bank {
    ...
    };

    interface Account {
    ...
    };
};
```

The interfaces `Bank` and `Account` are *scoped* within the module `BankSimple`. IDL definitions are available directly within the scope in which you define them. In other naming scopes, you must use the scoping operator (`::`) to access these definitions. For example, the fully scoped name of interfaces `Bank` and `Account` are `BankSimple::Bank` and `BankSimple::Account` respectively.

IDL modules can be *reopened*. For example, a module declaration can appear several times in a single IDL specification if each declaration contains different data types. In most IDL specifications, this feature of modules is not required.

Defining IDL Interfaces

An IDL interface describes the functions that an object supports in a distributed application. Interface definitions provide all of the information that clients need to access the object across a network.

Consider the example of an interface that describes objects which implement bank accounts in a distributed application. The IDL interface definition is as follows:

```
//IDL
module BankSimple {

        // Define a named type to represent money.
        typedef float CashAmount;
              // Forward declaration of interface Account.
        interface Account;

        interface Bank {
              ...
        };

        interface Account {
              // The account owner and balance.
              readonly attribute string name;
              readonly attribute CashAmount balance;

              // Operations available on the account.
              void deposit (in CashAmount amount);
              void withdraw (in CashAmount amount);
        };
};
```

The definition of interface `Account` includes both *attributes* and *operations*. These are the main elements of any IDL interface definition.

Attributes in IDL Interface Definitions

Conceptually, attributes correspond to variables that an object implements. Attributes indicate that these variables are available in an object and that clients can read or write their values.

In general, attributes map to a pair of functions in the programming language used to implement the object. These functions allow client applications to read or write the attribute values. However, if an attribute is preceded by the keyword `readonly`, then clients can only read the attribute value.

For example, the `Account` interface defines the attributes `name` and `balance`. These attributes represent information about the account which the object implementation can set, but which client applications can only read.

Operations in IDL Interface Definitions

IDL operations define the format of functions, methods, or operations that clients use to access the functionality of an object. An IDL operation can take parameters and return a value, using any of the available IDL data types.

For example, the `Account` interface defines the operations `deposit()` and `withdraw()` as follows:

```
//IDL
module BankSimple {
    typedef float CashAmount;
    ...

    interface Account {
        // Operations available on the account
        void deposit(in CashAmount amount);
        void withdraw(in CashAmount amount);
        ...
    };
};
```

Each operation takes a parameter and has a `void` return type.

Each parameter definition must specify the direction in which the parameter value is passed. The possible parameter passing modes are as follows:

`in`	The parameter is passed from the caller of the operation to the object.
`out`	The parameter is passed from the object to the caller.
`inout`	The parameter is passed in both directions.

Parameter passing modes clarify operation definitions and allow an IDL compiler to map operations accurately to a target programming language.

Raising Exceptions in IDL Operations

IDL operations can raise exceptions to indicate the occurrence of an error. CORBA defines two types of exceptions:

- *System exceptions* are a set of standard exceptions defined by CORBA.
- *User-defined exceptions* are exceptions that you define in your IDL specification.

Implicitly, all IDL operations can raise any of the CORBA system exceptions. No reference to system exceptions appears in an IDL specification.

To specify that an operation can raise a user-defined exception, first define the exception structure and then add an IDL `raises` clause to the operation definition.

For example, the operation `withdraw()` in interface `Account` could raise an exception to indicate that the withdrawal has failed, as follows:

```
// IDL
module BankExceptions {
    typedef float CashAmount;
    ...

    interface Account {
        exception InsufficientFunds {
            string reason;
        };

        void withdraw(in CashAmount amount)
            raises(InsufficientFunds);
        ...
    };
};
```

An IDL exception is a data structure that contains member fields. In the preceding example, the exception `InsufficientFunds` includes a single member of type string.

The `raises` clause follows the definition of operation `withdraw()` to indicate that this operation can raise exception `InsufficientFunds`. If an operation can raise more then one type of user-defined exception, include each exception identifier in the `raises` clause and separate the identifiers using commas.

Invocation Semantics for IDL Operations

By default, IDL operations calls are *synchronous*, that is a client calls an operation and blocks until the object has processed the operation call and returned a value. The IDL keyword `oneway` allows you to modify these invocation semantics.

If you precede an operation definition with the keyword `oneway`, a client that calls the operation will not block while the object processes the call. For example, you could add a oneway operation to interface `Account` that sends a notice to an `Account` object, as follows:

```
module BankSimple {
    ...

    interface Account {
        oneway void notice(in string text);
        ...
    };
};
```

Orbix does not guarantee that a oneway operation call will succeed; so if a oneway operation fails, a client may never know. There is only one circumstance in which Orbix indicates failure of a oneway operation. If a oneway operation call fails *before* Orbix transmits the call from the client address space, then Orbix raises a system exception.

A oneway operation can not have any `out` or `inout` parameters and can not return a value. In addition, a oneway operation can not have an associated `raises` clause.

Passing Context Information to IDL Operations

CORBA context objects allow a client to map a set of identifiers to a set of string values. When defining an IDL operation, you can specify that the operation should receive the client mapping for particular identifiers as an implicit part of the operation call. To do this, add a `context` clause to the operation definition.

Consider the example of an `Account` object, where each client maintains a set of identifiers, such as `sys_time` and `sys_location` that map to information that the operation `deposit()` logs for each deposit received. To ensure that this information is passed with every operation call, extend the definition of `deposit()` as follows:

```
// IDL
module BankSimple {
    typedef float CashAmount;
    ...

    interface Account {
        void deposit(in CashAmount amount)
            context("sys_time", "sys_location");
        ...
    };
};
```

A `context` clause includes the identifiers for which the operation expects to receive mappings.

Note that IDL contexts are rarely used in practice.

Inheritance of IDL Interfaces

IDL supports inheritance of interfaces. An IDL interface can inherit all the elements of one or more other interfaces.

For example, the following IDL definition illustrates two interfaces, called `CheckingAccount` and `SavingsAccount`, that inherit from interface `Account`:

```
// IDL
module BankSimple{
    interface Account {
        ...
    };

    interface CheckingAccount : Account {
        readonly attribute overdraftLimit;
        boolean orderChequeBook ();
    };

    interface SavingsAccount : Account {
        float calculateInterest ();
    };
};
```

Interfaces `CheckingAccount` and `SavingsAccount` implicitly include all elements of interface `Account`.

An object that implements `CheckingAccount` can accept invocations on any of the attributes and operations of this interface, and on any of the elements of interface `Account`. However, a

CheckingAccount object may provide different implementations of the elements of interface Account to an object that implements Account only.

The following IDL definition shows how to define an interface that inherits both CheckingAccount and SavingsAccount:

```
// IDL
module BankSimple {
    interface Account {
        ...
    };

    interface CheckingAccount : Account {
        ...
    };

    interface SavingsAccount : Account {
        ...
    };

    interface PremiumAccount :
        CheckingAccount, SavingsAccount {
        ...
    };
};
```

Interface PremiumAccount is an example of multiple inheritance in IDL. Figure 8 on page 38 illustrates the inheritance hierarchy for this interface.

If you define an interface that inherits from two interfaces which contain a constant, type, or exception definition of the same name, you must fully scope that name when using that constant, type, or exception. An interface can not inherit from two interfaces that include operations or attributes that have the same name.

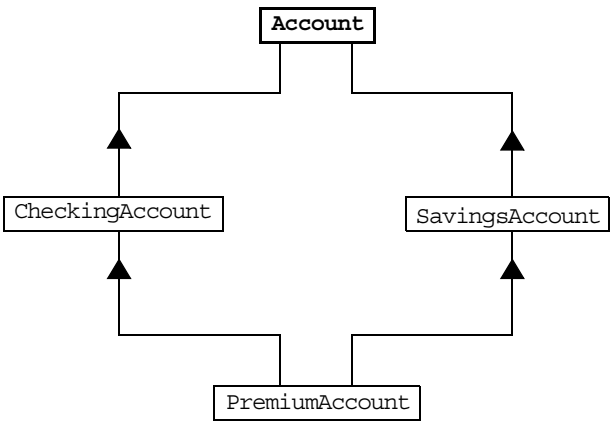


Figure 8: *Multiple Inheritance of IDL Interfaces*

The Object Interface Type

IDL includes the pre-defined interface Object, which all user-defined interfaces inherit implicitly. The operations defined in this interface are described in the ***Orbix Programmer's Reference C++ Edition***.

While interface `Object` is never defined explicitly in your IDL specification, the operations of this interface are available through all your interface types. In addition, you can use `Object` as an attribute or operation parameter type to indicate that the attribute or operation accepts any interface type, for example:

```
// IDL
interface ObjectLocator
{
    void getAnyObject (out Object obj);
};
```

Note that it is not legal IDL syntax to inherit interface `Object` explicitly.

Forward Declaration of IDL Interfaces

In an IDL definition, you must declare an IDL interface before you reference it. A forward declaration declares the name of an interface without defining it. This feature of IDL allows you to define interfaces that mutually reference each other.

For example, IDL interface `Bank` includes an operation of IDL interface type `Account`, to indicate that `Bank` stores a reference to an `Account` object. If the definition of interface `Account` follows the definition of interface `Bank`, you must forward declare `Account` as follows:

```
// IDL
module BankSimple {
    // Forward declaration of Account.
    interface Account;

    interface Bank {
        Account create_account (in string name);
        Account find_account (in string name);
    };
    // Full definition of Account.
    interface Account {
        ...
    };
};
```

The syntax for a forward declaration is the keyword `interface` followed by the interface identifier.

Overview of the IDL Data Types

In addition to IDL module, interface, and exception types, there are three general categories of data type in IDL:

- *Basic types*.
- *Complex types*.
- *Pseudo object types*.

This section examines each category of IDL types in turn and also describes how you can define new data type names in IDL.

IDL Basic Types

The following table lists the basic types supported in IDL.

IDL Type	Range of Values
short	$-2^{15}...2^{15}-1$ (16-bit)
unsigned short	$0...2^{16}-1$ (16-bit)
long	$-2^{31}...2^{31}-1$ (32-bit)
unsigned long	$0...2^{32}-1$ (32-bit)
long long	$-2^{63}...2^{63}-1$ (64-bit)
unsigned long long	$0...-2^{64}$ (64-bit)
float	IEEE single-precision floating point numbers.
double	IEEE double-precision floating point numbers.
char	An 8-bit value.
boolean	TRUE or FALSE.
octet	An 8-bit value that is guaranteed not to undergo any conversion during transmission.
any	The any type allows the specification of values that can express an arbitrary IDL type.

The any data type allows you to specify that an attribute value, an operation parameter, or an operation return value can contain an arbitrary type of value to be determined at runtime. Type any is described in detail in “The Any Data Type”.

IDL Complex Types

This section describes the IDL data types enum, struct, union, string, sequence, array, and fixed.

Enum

An enumerated type allows you to assign identifiers to the members of a set of values, for example:

```
// IDL
module BankSimple {
    enum Currency {pound, dollar, yen, franc};

    interface Account {
        readonly attribute CashAmount balance;
        readonly attribute Currency
             balanceCurrency;
        ...
    };
};
```

In this example, attribute `balanceCurrency` in interface `Account` can take any one of the values `pound`, `dollar`, `yen`, or `franc`.

Struct

A struct data type allows you to package a set of named members of various types, for example:

```
// IDL
module BankSimple{
        struct CustomerDetails {
            string name;
            short age;
        };

        interface Bank {
            CustomerDetails getCustomerDetails
                (in string name);
            ...
        };
};
```

In this example, the struct `CustomerDetails` has two members. The operation `getCustomerDetails()` returns a struct of type `CustomerDetails` that includes values for the customer name and age.

Union

A union data type allows you to define a structure that can contain only one of several alternative members at any given time. A union saves space in memory, as the amount of storage required for a union is the amount necessary to store its largest member.

All IDL unions are *discriminated*. A discriminated union associates a label value with each member. The value of the label indicates which member of the union currently stores a value.

For example, consider the following IDL union definition:

```
// IDL
struct DateStructure {
    short Day;
    short Month;
    short Year;
};

union Date switch (short) {
    case 1: string stringFormat;
    case 2: long digitalFormat;
    default: DateStructure structFormat;
};
```

The union type `Date` is discriminated by a short value. For example, if this short value is `1`, then the union member `stringFormat` stores a date value as an IDL string. The default label associated with the member `structFormat` indicates that if the short value is not `1` or `2`, then the `structFormat` member stores a date value as an IDL struct.

Note that the type specified in parentheses after the `switch` keyword must be an integer, char, boolean or enum type and the value of each `case` label must be compatible with this type.

String

An IDL string represents a character string, where each character can take any value of the char basic type.

If the maximum length of an IDL string is specified in the string declaration, then the string is *bounded*. Otherwise the string is *unbounded*.

The following example shows how to declare bounded and unbounded strings:

```
// IDL
module BankSimple {
            interface Account {
                // A bounded string with maximum length 10.
                attribute string<10> sortCode;

                // An unbounded string.
                readonly attribute string name;
                ...
            };
};
```

Sequence

In IDL, you can declare a sequence of any IDL data type. An IDL sequence is similar to a one-dimensional array of elements.

An IDL sequence does not have a fixed length. If the sequence has a fixed maximum length, then the sequence is *bounded*. Otherwise, the sequence is *unbounded*.

For example, the following code shows how to declare bounded and unbounded sequences as members of an IDL struct:

```
// IDL
module BankSimple {
    interface Account {
        ...
    };

    struct LimitedAccounts {
        string bankSortCode<10>;
        // Maximum length of sequence is 50.
        sequence<Account, 50> accounts;
    };
    struct UnlimitedAccounts {
        string bankSortCode<10>;
        // No maximum length of sequence.
        sequence<Account> accounts;
    };
};
```

A sequence must be named by an IDL `typedef` declaration before it can be used as the type of an IDL attribute or operation parameter. Refer to "Defining Data Type Names and Constants" on page 45 for details. The following code illustrates this:

```
// IDL
module BankSimple {
    typedef sequence<string> CustomerSeq;

    interface Account {
        void getCustomerList(out CustomerSeq names);
        ...
    };
};
```

Arrays

In IDL, you can declare an array of any IDL data type. IDL arrays can be multi-dimensional and always have a fixed size. For example, you can define an IDL struct with an array member as follows:

```
// IDL
module BankSimple {
    ...

    interface Account {
        ...
    };

    struct CustomerAccountInfo {
        string name;
        Account accounts[3];
    };

    interface Bank {
        getCustomerAccountInfo (in string name,
            out CustomerAccountInfo accounts);
        ...
    };
};
```

In this example, struct `CustomerAccountInfo` provides access to an array of `Account` objects for a bank customer, where each customer can have a maximum of three accounts.

An array must be named by an IDL `typedef` declaration before it can be used as the type of an IDL attribute or operation parameter. The IDL `typedef` declaration allows you define an alias for a data type, as described in "Defining Data Type Names and Constants" on page 45.

The following code illustrates this:

```
// IDL
module BankSimple {
    interface Account {
        ...
    };

    typedef Account AccountArray[100];

    interface Bank {
        readonly attribute AccountArray accounts;
        ...
    };
};
```

Note that an array is a less flexible data type than an IDL sequence, because an array always has a fixed length. An IDL sequence always has a variable length, although it may have an associated maximum length value.

Fixed

The fixed data type allows you to represent number in two parts: a *digit* and a *scale*. The digit represents the length of the number, and the scale is a non-negative integer that represents the position of the decimal point in the number, relative to the rightmost digit.

```
module BankSimple {
            typedef fixed<10,4> ExchangeRate;

            struct Rates {
                        ExchangeRate USRate;
                        ExchangeRate UKRate;
                        ExchangeRate IRRate;
            };
};
```

In this case, the `ExchangeRate` type has a digit of size 10, and a scale of 4. This means that it can represent numbers up to (+/-)999999.9999.

The maximum value for the digits is 31, and scale cannot be greater than digits. The maximum value that a fixed type can hold is equal to the maximum value of a `double`.

Scale can also be a negative number. This means that the decimal point is moved scale digits in a rightward direction, causing trailing zeros to be added to the value of the fixed. For example, fixed `<3,-4>` with a numeric value of `123` actually represents the number `1230000`. This provides a mechanism for storing numbers with trailing zeros in an efficient manner.

Note: Fixed `<3, -4>` can also be represented as fixed `<7, 0>`.

Constant fixed types can also be declared in IDL. The digits and scale are automatically calculated from the constant value. For example:

```
module Circle {
          const fixed pi = 3.142857;
};
```

This yields a fixed type with a digits value of 7, and a scale value of 6.

IDL Pseudo Object Types

CORBA defines a set of pseudo object types that ORB implementations use when mapping IDL to some programming languages. These object types have interfaces defined in IDL but do not have to follow the normal IDL mapping for interfaces and are not generally available in your IDL specifications.

You can use only the following pseudo object types as attribute or operation parameter types in an IDL specification:

```
CORBA::NamedValue
CORBA::Principal
CORBA::TypeCode
```

To use any of these three types in an IDL specification, include the file orb.idl in the IDL file as follows:

```
// IDL
#include <orb.idl>
...
```

This statement indicates to the IDL compiler that types NamedValue, Principal, and TypeCode may be used. The file orb.idl should not actually exist in your system. Do not name any of your IDL files orb.idl.

Defining Data Type Names and Constants

IDL allows you to define new data type names and constants. This section describes how to use each of these features of IDL.

Data Type Names

The typedef keyword allows you define a meaningful or more simple name for an IDL type. The following IDL provides a simple example of using this keyword:

```
// IDL
module BankSimple {
    interface Account {
        ...
    };

    typedef Account StandardAccount;
};
```

The identifier StandardAccount can act as an alias for type Account in subsequent IDL definitions. Note that CORBA does not specify whether the identifiers Account and StandardAccount represent distinct IDL data types in this example.

Constants

IDL allows you to specify constant data values using one of several basic data types. To declare a constant, use the IDL keyword `const`, for example:

```
// IDL
module BankSimple {
    interface Bank {
        const long MaxAccounts = 10000;
        const float Factor = (10.0 - 6.5) * 3.91;
        ...
    };
};
```

The value of an IDL constant cannot change. You can define a constant at any level of scope in your IDL specification.

The CORBA IDL to C++ Mapping

The CORBA Interface Definition Language (IDL) to C++ mapping specifies how to write C++ programs that access or implement IDL interfaces. This chapter describes this mapping in full.

CORBA separates the definition of an object's interface from the implementation of that interface. As described in "Introduction to CORBA IDL" on page 33, IDL allows you to define interfaces to objects. To implement and use those interfaces, you must use a programming language such as C, C++, Java, Ada, or Smalltalk.

The Orbix IDL compiler allows you to implement and use IDL interfaces in C++. The compiler does this by generating C++ constructs that correspond to your IDL definitions, in accordance with the standard CORBA IDL to C++ mapping.

This chapter describes the CORBA IDL to C++ mapping, as defined in the C++ mapping section of the OMG *Common Object Request Broker Architecture*. The purpose of the chapter is to explain the rules by which the Orbix IDL compiler converts IDL definitions into C++ code and how to use the generated C++ constructs.

This chapter contains a lot of detailed technical information that you require when developing Orbix applications. However, you should not try to learn all the technical details at once. Instead, read this chapter briefly to understand the mappings for the main IDL constructs, such as modules, interfaces, and basic types, and the C++ memory management rules associated with the mapping. When writing applications, consult this chapter for detailed information about mapping the specific IDL constructs you require.

Overview of the Mapping

The major elements of the IDL to C++ mapping are:

- An IDL `module` maps to a C++ `namespace` of the same name. Alternative mappings are provided for C++ compilers that do not support the `namespace` construct.
- An IDL `interface` maps to a C++ class of the same name.
- An IDL operation maps to a C++ member function in the corresponding C++ class.
- An IDL attribute maps to a pair of overloaded C++ member functions in the corresponding C++ class. These functions allow a client program to set and read the attribute value.

Note that IDL identifiers map directly to identifiers of the same name in C++. However, if an IDL definition contains an identifier that exactly matches a C++ keyword, the identifier is mapped to the name of the identifier preceded by an underscore. An IDL identifier cannot begin with an underscore.

Mapping for Modules and Scoping

IDL modules map to C++ namespaces, where your C++ compiler supports them. For example:

```
// IDL
module BankSimple {
    struct Details {
        ...
    };
};
```

This maps to:

```
// C++
namespace BankSimple {
    struct Details {
        ...
    };
};
```

Outside of namespace `BankSimple`, the struct `Details` can be referred to as `BankSimple::Details`. Alternatively, a C++ `using` directive allows you to refer to `Details` without explicit scoping:

```
// C++
using namespace BankSimple;
Details d;
```

Alternative Mappings for Modules

Since namespaces have only recently been added to the C++ language, few compilers support them. In the absence of support for namespaces, IDL modules map to C++ classes that have no member functions or data. This allows IDL scoped names to be mapped directly onto C++ scoped names. For example:

```
// IDL
module BankSimple {
    interface Bank {
        ...
        struct Details {
            ...
        };
    };
};
```

This maps to:

```
// C++
class BankSimple {
public:
    ...
    class Bank : public virtual CORBA::Object {
        ...
        struct Details {
            ...
        };
    };
};
```

You can use struct `Details` in C++ as follows:

```
// C++
BankSimple::Bank::Details d;
```

Mapping for Interfaces

Each IDL interface maps to a C++ class that defines a client programmer's view of the interface. This class lists the C++ member functions that a client can call on objects that implement the interface.

Each IDL interface also maps to other C++ classes that allow a server programmer to implement the interface using either the *BOAImpl* or *TIE* approach. However, this chapter describes only the C++ class that describes the client view of the interface, as this class is sufficient to illustrate the principles of the mapping for interfaces.

Consider a simple interface to describe a bank account:

```
// IDL
...
typedef float CashAmount;
...
    interface Account {
        readonly attribute CashAmount balance;
        void deposit (in CashAmount amount);
        void withdraw (in CashAmount amount);
    };
```

This maps to the following IDL C++ class:

```
// C++
class Account : public virtual CORBA::Object {
public:
    virtual CashAmount balance();
    virtual void deposit (in CashAmount amount);
    virtual void withdraw (in CashAmount amount);
};
```

Implicitly, all IDL interfaces inherit from interface `CORBA::Object`. Class `Account` inherits from the Orbix class `CORBA::Object`, which maps the functionality of interface `CORBA::Object`.

Class `Account` defines the client view of the IDL interface `Account`. Conceptually, instances of class `Account` allow a client to access CORBA objects that implement interface `Account`. However, an Orbix program should never create an instance of class `Account` and should never use a pointer (`Account*`) or a reference (`Account&`) to this class.

Instead, an Orbix program should access objects of type `Account` through an interface helper type. Two helper types are generated for each IDL interface: a `_var` type and a `_ptr` type. For example, the helper types for interface `Account` are `Account_var` and `Account_ptr`.

Conceptually, a `_var` type is a managed pointer that assumes ownership of the data to which it points. This means that you can use a `_var` type such as `Account_var` as a pointer to an object of type `Account`, without ever deallocating the object memory. If a `_var` type goes out of scope or is assigned a new value, Orbix automatically manages the memory associated with the existing value of the `_var` type.

A `_ptr` type is more primitive and has similar semantics to a C++ pointer. In fact, `_ptr` types in Orbix are currently implemented as C++ pointers. However, it is important that you do not use this

knowledge because this implementation may change. For example, you should not attempt conversion to `void*`, arithmetic operations and relational operations, including test for equality on `_ptr` types.

The `_var` and `_ptr` types for an IDL interface allow a client to access IDL attributes and operations defined by the interface. Examples of how to use the `_var` and `_ptr` types are provided later in this section.

Mapping for Attributes

Each attribute in an IDL interface maps to two member functions in the corresponding C++ class. Both member functions have the same name as the attribute: one function allows clients to set the attribute's value and the other allows clients to read the value. A readonly attribute maps to a single member function that allows clients to read the value.

Consider the following IDL interfaces:

```
// IDL
interface Account {
    readonly attribute float balance;
    attribute long accountnumber;
    ...
};
```

The following code illustrates the mapping for attributes `balance` and `accountNumber`:

```
// C++
class Account : public virtual CORBA::Object {
public:
    virtual CORBA::Float balance(CORBA::Environment&);
    virtual CORBA::Long
        accountNumber(CORBA::Environment&);
    virtual void accountNumber
        (Long accountNumber, CORBA::Environment&);
    ...
    };
```

Note that the IDL type `float` maps to `CORBA::Float`, while type `long` maps to `CORBA::Long`. "Mapping for Basic Types" on page 57 provides a detailed description of this mapping.

The following code illustrates how a client program could access attributes `balance` and `accountnumber` of an `Account` object:

```
// C++
Account_var aVar;
CORBA::Float bal = 0;
CORBA::Long number = 99;
// Code to bind aVar to an Account object omitted.
...

try {
    // Get value of balance.
    bal = aVar->balance();
    // Set and get value of accountNumber.
    aVar->accountnumber(number);
    number = aVar->accountnumber();
}
```

```
catch (const CORBA::SystemException& se) {
    ...
}
```

Mapping for Operations

Operations within an interface map to virtual member functions of the corresponding C++ class. These member functions have the same name as the relevant IDL operations. This mapping applies to all operations, including those preceded by the IDL keyword `oneway`.

Consider the following IDL interfaces:

```
// IDL
typedef float CashAmount;
....

interface Account {
    void deposit(in CashAmount amount);
    void withdraw(in CashAmount amount);
    ...
};

interface Bank {
    Account create_account(in string name);
};
```

The following code illustrates the mapping for IDL operations:

```
// C++
class Account : public virtual CORBA::Object  {
public:
    virtual void deposit(CashAmount amount);
    virtual void withdraw(CashAmount amount);
    ...
};

class Bank : public virtual CORBA::Object {
public:
    virtual Account_ptr create_account
        (const char* name);
};
```

The IDL operation `create_account()` has an object reference return type; that is, it returns an `Account` object. In the corresponding C++ code for `create_account()`, the IDL object reference return type is mapped to the type `Account_ptr`. Note that you can assign the return value of function `create_account()` to either an `Account_ptr` or an `Account_var` value.

The following code illustrates how a client calls IDL operations on `Account` and `Bank` objects:

```
// C++
Account_var aVar;
Bank_var bVar;

// Code to bind bVar to a Bank object omitted.
...

try {
    aVar = bVar->create_account("Chris");
```

```
    aVar->deposit(100.00);
}
catch (const CORBA::SystemException& se) {
    ...
}
```

"Memory Management for Parameters" on page 78 provides more information about the mapping for operation parameters.

Mapping for Exceptions

A user-defined IDL exception type maps to a C++ class that derives from class `CORBA::UserException` and that contains the exception's data. For example, consider the following exception definition:

```
// IDL
exception CannotCreate {
    string reason;
    short s;
};
```

This maps to the following C++:

```
// C++
class CannotCreate : public CORBA::UserException {
public:
    CORBA::String_mgr reason;
    CORBA::Short s;

    CannotCreate(const char* _reason,
                 const CORBA::Short& _s);
    CannotCreate();
    CannotCreate(const CannotCreate&);
    ~CannotCreate();

    CannotCreate()& operator = (const
            CannotCreate&);
        static CannotCreate*
            _narrow(CORBA::Exception* e);
};
```

The mapping defines a constructor with one parameter for each exception member; this constructor initializes the exception member to the passed-in value. In the example, this constructor has two parameters, one for each of the fields `reason` and `s` defined in the exception.

You can throw an exception of type `CannotCreate` in an operation implementation as follows:

```
// C++
// Server code.
throw CannotCreate("My reason", 13)
```

The default exception constructor performs no explicit member initialization. The copy constructor, assignment operator, and destructor automatically copy or free the storage associated with the exception. Exceptions are mapped similarly to variable length structs in that each member of the exception must be self-managing.

Mapping for Contexts

An operation that specifies a context clause is mapped to a C++ member function in which an input parameter of type `Context_ptr` follows all operation-specific arguments. For example:

```
// IDL
interface A {
    void op(in unsigned long s)
        context ("accuracy", "base");
};
```

This interface maps to:

```
// C++
class A : public virtual CORBA::Object {
public:
    virtual void op(CORBA::ULong s,
        CORBA::Context_ptr IT_c);
};
```

The `Context_ptr` parameter appears before the `Environment` parameter. This order allows the `Environment` parameter to have a default value.

Mapping for Inheritance of IDL Interfaces

This section describes the mapping for interfaces that inherit from other interfaces. Consider the following example:

```
// IDL
interface CheckingAccount : Account {
    void setOverdraftLimit(in float limit);
};
```

The corresponding C++ is:

```
// C++
class CheckingAccount : public virtual Account {
public:
    virtual void setOverdraftLimit(
        CORBA::Float limit);
};
```

A C++ client program that uses the `CheckingAccount` interface can call the inherited `deposit()` function:

```
// C++
CheckingAccount_var checkingAc;

// Code for binding checkingAc omitted.
...

checkingAc->deposit(90.97);
```

Naturally, assignments from a derived to a base class object reference are allowed, for example:

```
// C++
Account_ptr ac = checkingAc;
```

Note that you should not attempt to make normal or cast assignments in the opposite direction—from a base class object reference to a derived class object reference. To make such assignments, you should use the Orbix *narrow* mechanism as described in "Narrowing Object References" on page 54.

Widening Object References

The C++ types generated for IDL interfaces support normal inheritance conversions. For example, for the preceeding `Account` and `CheckingAccount` classes defined the following conversions from a derived class object reference to a base class reference, known as *widenings*, are implicit:

- `CheckingAccount_ptr` to `Account_ptr`
- `CheckingAccount_ptr` to `Object_ptr`
- `CheckingAccount_var` to `Account_ptr`
- `CheckingAccount_var` to `Object_ptr`

Note: There is no implicit conversion between `_var` types. An attempt to widen from one `_var` type to another causes a compile-time error. Instead conversion between two `_var` types requires a call to `_duplicate()`.

Some widening examples are shown in the code below:

```
// C++
CheckingAccount_ptr cPtr = ....;

// Implicit widening:
Account_ptr aPtr = cPtr;

// Implicit widening:
Object_ptr objPtr = cPtr;

// Implicit widening:
objPtr = aPtr;

CheckingAccount_var cVar = cPtr;
// cVar assumes ownership of cPtr.
aPtr = cVar;
    // Implicit widening, cVar retains ownership of cPtr.

objPtr = cVar;
    // Implicit widening, cVar retains ownership of cPtr.

Account_var av = cVar;
    // Illegal, compile-time error, cannot assign
    // between _var variables of different types.

    Account_var aVar = CheckingAccount::_duplicate(cVar);
    // aVar and cVar both refer to cPtr.
    // The reference count of cPtr is incremented.
```

Narrowing Object References

If a client program receives an object reference of type `Account` that actually refers to an implementation object of type `CheckingAccount`, the client can safely convert the `Account` reference to a `CheckingAccount` reference. This conversion gives the client access to the operations defined in the derived interface `CheckingAccount`.

The process of converting an object reference for a base interface to a reference for a derived interface is known as *narrowing* an object reference. To narrow an object reference, you must use the `_narrow()` function that is defined as a `static` member function for each C++ class generated from an IDL interface.

For example, for interface T, the following C++ class is generated:

```
// C++
class T : public virtual CORBA::Object {
    static T_ptr _narrow(CORBA::Object_ptr);
    ...
};
```

The following code shows how to narrow an `Account` reference to a `CheckingAccount` reference:

```
// C++
Account_ptr aPtr;
CheckingAccount_ptr caPtr;

// Code to bind aPtr to an object that implements
// CheckingAccount omitted.
...

// Narrow aPtr to be a CheckingAccount.
if (caPtr = CheckingAccount::_narrow(aPtr))
    ...
else
    // Deal with failure of _narrow().
```

If the parameter passed to `T::_narrow()` is not of class `T` or one of its derived classes, `T::_narrow()` returns a *nil object reference*. The `_narrow()` function can also raise a system exception, and you should always check for this.

Each object reference in an address space has an associated *reference count*. A successful call to `_narrow()` increases the reference count of an object reference by one.

Object Reference Counts and Nil Object References

Each Orbix program may use a single object reference several times. To determine whether an object reference is currently in use in a program, Orbix associates a reference count with each reference. This section describes the Orbix reference counting mechanism and explains how to test for nil object references.

Object Reference Counts

In Orbix, the reference count of an object is the number of pointers to the object that exist *within the same address space*. Each object is initially created with a reference count of one.

You can explicitly increase the reference count of an object by calling the object's `_duplicate()` static member function. The `CORBA::release()` function on a pointer to an object reduces the object's reference count by one, and destroys the object if the reference count is then zero.

For example, consider the following server code:

```
// C++
// Create a new Bank object:
Bank_ptr bPtr = new Bank_i;
// The reference count of the new object is 1.

Bank::_duplicate(bPtr);
// The reference count of the object is 2.
```

```
CORBA::release(bPtr);
// The reference count of the object is 1.
```

Both implementation objects in servers, and proxies in clients have reference counts. Calls to `_duplicate()` and `CORBA::release()` by a client *do not affect* the reference count of the target object in the server. Instead, each proxy has its own reference count that the client can manipulate by calling `_duplicate()` and `CORBA::release()`. Deletion of a proxy (by a call to `CORBA::release()` that causes the reference count to drop to zero) does not affect the reference count of the target object.

A server can delete an object (by calling `CORBA::release()` an appropriate number of times) even if one or more clients hold proxies for this object. If this happens, subsequent invocations through the proxy causes an `CORBA::INV_OBJREF` system exception to be raised.

Some operations implicitly increase the reference count of an object. For example, if a client obtains a reference to the same object many times—for example, using the Naming Service—this results in only one proxy being created in that client's address space. The reference count of this proxy is the number of references obtained by the client.

To find the current reference count for an object, call the function `_refCount()` on the object reference. This function is defined in class `CORBA::Object` as follows:

```
// C++
// In class CORBA::Object.
CORBA::ULong _refCount();
```

You can call this function as follows:

```
// C++
T_ptr tPtr;
...
CORBA::ULong count = tPtr->_refCount();
```

Nil Object References

A nil object reference is a reference that does not refer to any valid Orbix object. Each C++ class for an IDL interface defines a static function `_nil()` that returns a nil object reference for that interface type.

For example, an IDL interface T generates the following C++:

```
// C++
class T : public virtual CORBA::Object {
    static T_ptr _nil(CORBA::Environment&);
    ...
};
```

To obtain a nil object reference for `T`, do the following:

```
// C++
// Obtain a nil object reference for T:
T_ptr tPtr = T::_nil();
```

The function `is_nil()`, defined in the `CORBA` namespace, determines whether an object reference is nil. The function `is_nil()` is declared as:

```
// C++
// In CORBA namespace.
Boolean is_nil(Object_ptr obj);
```

The following call is guaranteed to be true:

```
// C++
CORBA::Boolean result = CORBA::is_nil(T::_nil());
```

Note that calling `is_nil()` is the only CORBA-compliant way in which you can check if an object reference is nil. Do not compare object references using `operator==()`.

Mapping for IDL Data Types

This section describes the mapping for each of the IDL basic types, constructed types, and template types.

Mapping for Basic Types

The IDL basic data types have the mappings shown in the following table:

IDL	C++
short	CORBA::Short
long	CORBA::Long
long long	CORBA::LongLong
unsigned short	CORBA::UShort
unsigned long	CORBA::ULong
unsigned long long	CORBA::ULongLong
float	CORBA::Float
double	CORBA::Double
char	CORBA::Char
boolean	CORBA::Boolean
octet	CORBA::Octet
any	CORBA::Any

Each IDL basic type maps to a typedef in the `CORBA` module; for example, the IDL type `short` maps to `CORBA::Short` in C++. This is because on different platforms, C++ types such as `short` and `long` may have different representations.

The types `CORBA::Short`, `CORBA::UShort`, `CORBA::Long`, `CORBA::ULong`, `CORBA::LongLong`, `CORBA::ULongLong`, `CORBA::Float`, and `CORBA::Double` are implemented using distinguishable C++ types. This enables these types to be used to distinguish between overloaded C++ functions and operators.

The IDL type `boolean` maps to `CORBA::Boolean` which is implemented as a typedef to the C++ type `unsigned char` in Orbix. The mapping of the IDL `boolean` type to C++ defines only the values `1` (`TRUE`) and `0` (`FALSE`); other values produce undefined behavior.

The mapping for type `any` is described in "The Any Data Type".

Mapping for Complex Types

The remainder of this section describes the mapping for IDL types enum, struct, union, string, sequence, fixed, and array. This section also describes the mapping for IDL typedefs and constants.

The mappings for IDL types struct, union, array, and sequence depend on whether these types are *fixed length* or *variable length*. A fixed length type is one whose size in bytes is known at compile time. A variable length type is one in which the number of bytes occupied by the type can only be calculated at runtime.

The following IDL types are considered to be variable length types:

- A bounded or unbounded string.
- A bounded or unbounded sequence.
- An object reference.
- A struct or union that contains a member whose type is variable length.
- An array with a variable length element type.
- A typedef to a variable length type.
- The type `any`.

Mapping for Enum

An IDL `enum` maps to a corresponding C++ `enum`. For example:

```
// IDL
enum Colour {blue, green};
```

This maps to:

```
// C++
enum Colour {blue, green,
    IT__ENUM_Colour = CORBA_ULONG_MAX};
```

The additional constant `IT__ENUM_Colour` is generated in order to force the C++ compiler to use exactly 32 bits for values declared to be of the enumerated type.

Mapping for Struct

An IDL struct maps directly to a C++ struct. Each member of the IDL struct maps to a corresponding member of the C++ struct. The generated struct contains an empty default constructor, an empty destructor, a copy constructor and an assignment operator.

Fixed Length Structs

Consider the following IDL fixed length struct:

```
// IDL
struct AStruct {
    long l;
    float f;
};
```

This maps to:

```
// C++
struct AStruct {
    CORBA::Long l;
    CORBA::Float f;
};
```

Variable Length structs

Consider the following IDL variable length struct:

```
// IDL
interface A {
    ...
};

struct VariableLengthStruct {
    short i;
    float f;
    string str;
    A a;
};
```

This maps to a C++ struct as follows:

```
// C++
struct VariableLengthStruct {
    CORBA::Short i;
    CORBA::Float f;
    CORBA::String_mgr str;
    A_mgr a;
};
```

Except for strings and object references, the type of the C++ struct member is the normal mapping of the IDL member's type.

String and object reference members of a variable length struct map to special *manager classes*. Note these manager (`_mgr`) types are only used internally in Orbix. You *should not* write application code that explicitly declares or names manager classes.

The behavior of manager types is the same as the normal mapping (`char*` for `string` and `A_ptr` for an `interface`) except that the manager type is responsible for managing the member's memory. In particular, the assignment operator releases the storage for the existing member and the copy constructor copies the member's storage.

The implications of this are that the following code, for example, does not cause a memory leak:

```
// C++
VariableLengthStruct vls;
char* s1 = CORBA::string_alloc(5+1);
char* s2 = CORBA::string_alloc(6+1);
strcpy(s1, "first");
strcpy(s2, "second");
vls.str = s1;
vls.str = s2; // No memory leak, s1 is released.
```

Mapping for Union

An IDL `union` maps to a C++ struct. Consider the following IDL declaration:

```
// IDL
typedef long vector[100];
struct S { ... };
interface A;

union U switch(long) {
    case 1: float f;
    case 2: vector v;
    case 3: string s;
    case 4: S st;
    default: A obj;
};
```

This maps to the following C++ struct:

```
// C++
struct U {
public:
    // The discriminant.
    CORBA::Long _d() const;                          (1)

    // Constructors, Destructor, and Assignment.
    U();                                             (2)
    U(const CORBA::Long);                            (2a)
    U(const U&);                                     (3)
    ~U();                                            (4)
    U& operator = (const U&);                        (5)

    // Accessor and modifier functions for members.
    // Basic type member:
    CORBA::Float f() const;                          (6)
    void f(CORBA::Float IT_member);                  (7)

    // Array member:
    vector_slice* v() const;                         (8)
    void v(vector_slice* IT_member);                 (9)

    // String member:
    const char* s() const;                           (10)
    void s(char* IT_member);                         (11)
    void s(CORBA::String_var IT_member);             (12)
    void s(const char* IT_member);                   (13)
```

```
        // Struct member:
        S& st();                                    (14)
        const S& st() const;                        (15)
        void st(const S& IT_member);                (16)

        // Object reference member:
        A_ptr obj() const;                          (17)
        void obj(A_ptr IT_member);                  (18)
        ...
    };
```

The Discriminant

The value of the discriminant indicates the type of the value that the union currently holds. This is the value specified in the IDL union definition. The function `_d()` (1) returns the current value of the discriminant.

Constructors, Destructor and Assignment

The default constructor (2) does not initialize the discriminant and it does not initialize any union members. Therefore, it is an error for an application to access a union before setting it and Orbix does not detect this error. The Orbix IDL Compiler generates an extra constructor (2a) that takes an argument of the same type as the discriminant.

The copy constructor (3) and assignment operator (5) perform a deep-copy of their parameters; the assignment operator releases old storage if necessary and then performs a deep copy. The destructor (4) releases all storage owned by the union.

Accessors and Modifiers

For each member of the union, an accessor function is generated to read the value of the member and, depending on the type of the member, one or more modifier functions are generated to change the value of the member.

Setting the union value through a modifier function also sets the discriminant and, depending on the type of the previous value, may release storage associated with that value. An attempt to get a value through an accessor function that does not match the discriminant results in undefined behavior.

Only the accessor functions for struct, union, sequence, and `any` return a reference to the appropriate type: thus, the value of this type may be modified either by using the appropriate modifier function or by directly modifying the return value of the accessor. Because the memory associated with these types is owned by the union, the return value of an accessor function should not be assigned to a `_var` type. A `_var` type would attempt to assume ownership of the memory.

For a `union` member whose type is an array, the accessor function (8) returns a pointer to the array slice (refer to "Mapping for Array" on page 73). The array slice return type allows for read-write access for array members using `operator[]()` defined for arrays.

For `string` `union` members, the `char*` modifier function (11) first frees old storage before taking ownership of the `char*` parameter; that is, the parameter is not copied. The `const char*` modifier (13) and the `String_var` modifier (12) both free old storage before the parameter's storage is copied.

Since the type of a string literal is `char*` rather than `const char*`, the following code would result in a delete error:

```
// C++
{
    U u;
    u.s("A String");
        // Calls char* version of s. The string is
        // not copied.
} // Error: u destructor tries to delete
  // the string literal "A String".
```

Note:

The string (`char*`) is managed by a `CORBA::String_mgr` whose destructor calls delete. This results in undefined behavior which the C++ compiler is not required to flag.

Thus, an explicit cast to `const char*` is required in the special case where a string literal is passed to a string modifier function.

For object reference `union` members, the modifier function (18) releases the old object reference and duplicates the new one. An object reference return value from the accessor function (17) is not duplicated, because the `union` retains ownership of the object reference.

Example Program

A C++ program may access the elements of a `union` as follows:

```
// C++
U* u;

u = new U;
u->f(19.2);

// And later:
switch (u->_d()) {
    case 1 : cout << "f = " << u->f()
                        << endl; break;

    case 2 : cout << "v = " << u->v()
                        << endl; break;

    case 3 : cout << "s = " << u->s()
                        << endl; break;
    // Do not free the returned string.

    case 4 : cout << "st = " << "x = " << u->st().x
                        << " " << "y = " << u->st().y
                        << endl; break;

    default: cout << "A = " << u->obj() << endl; break;
    // Do not release the returned object
    // reference.
}
```

Mapping for String

IDL strings are mapped to character arrays that terminate with '`\0`' (the ASCII `NUL` character). The length of the string is encoded in the character array itself through the placement of the `NUL` character.

In addition, the `CORBA` namespace defines a class `String_var` that contains a `char*` value and automatically frees the memory referenced by this pointer when a `String_var` object is deallocated, for example, by going out of scope.

The `String_var` class provides operations to convert to and from `char*` values, and `operator[]()` allows access to characters within the string.

Consider the following IDL:

```
// IDL
typedef string<10> stringTen; // A bounded string.
typedef string stringInf; // An unbounded string.
```

The corresponding C++ is:

```
// C++
typedef char* stringTen;
typedef CORBA::String_var stringTen_var;
typedef char* stringInf;
typedef CORBA::String_var stringInf_var;
```

You can define instances of these types in C++ as follows:

```
// C++
stringTen s1 = 0;
stringInf s2 = 0;

// Or using the _var type:
CORBA::stringTen_var sv1;
CORBA::stringInf_var sv2;
```

At all times, a bounded string pointer, such as `stringTen`, should reference a storage area large enough to hold its type's maximum string length.

Dynamic Allocation of Strings

To allocate and free a string dynamically, you must use the following functions from the `CORBA` namespace:

```
// C++
// In namespace CORBA.
char* string_alloc(CORBA::ULong len);
void string_free(char*);
```

Do not use the C++ `new` and `delete` operators to allocate memory for strings passed to Orbix from a client or server. However, you can use `new` and `delete` to allocate a string that is local to the program and is never passed to Orbix.

The `string_alloc()` function dynamically allocates a string, or returns a null pointer if it cannot perform the allocation. The `string_free()` function deallocates a string that was allocated with `string_alloc()`. For example:

```
// C++
{
    char* s = CORBA::string_alloc(10+1);
    strcpy(s, "0123456789");
    ...
    CORBA::string_free(s);
}
```

The function `CORBA::string_dup()` copies a string passed to it: as a parameter

```
// C++
char* string_dup(const char*);
```

Space for the copy is allocated using `string_alloc()`.

By using the `CORBA::String_var` types, you are relieved of the responsibility of freeing the space for a string. For example:

```
// C++
{
    CORBA::String_var sVar = CORBA::string_alloc(10+1);
    strcpy(sVar, "0123456789");
    ...
}   // String held by sVar automatically freed here.
```

Bounds Checking of String Parameters

Although you can define bounded IDL string types, C++ does not perform any bounds checking to prevent a string from exceeding the bound. Since strings map to `char*`, they are effectively unbounded.

Consequently, Orbix takes responsibility for checking the bounds of strings passed as operation parameters. If you attempt to pass a string to Orbix that exceeds the bound for the corresponding IDL string type, Orbix detects this error and raises a system exception.

General Mapping for Sequences

The IDL data type sequence is mapped to a C++ class that behaves like an array with a *current length* and a *maximum length*. A `_var` type is also generated for each sequence.

The maximum length for a bounded sequence is defined in the sequence's IDL type and cannot be explicitly controlled by the programmer. Attempting to set the current length to a value larger than the maximum length given in the IDL specification is undefined. Orbix checks the length against maximum bound and, if this is greater, does nothing.

For an unbounded sequence, the initial value of the maximum length can be specified in the sequence constructor to allow control over the size of the initial buffer allocation. The programmer may always explicitly modify the current length of any sequence.

If the length of an unbounded sequence is set to a larger value than the current length, the sequence data may be reallocated. Reallocation is conceptually equivalent to creating a new sequence of the desired new length, copying the old sequence elements into the new sequence, releasing the original elements, and then assigning the old sequence to be the same as the new sequence. Setting the length to a smaller value than the current length does not result in any reallocation. The current length is set to the new value and the maximum remains the same.

Mapping for Unbounded Sequences

Consider the following IDL declaration:

```
// IDL
typedef sequence<long> unbounded;
```

The IDL compiler generates the following class definition:

```
// C++
class unbounded {
public:
    unbounded();                                        (1)
    unbounded(const unbounded&);                        (2)

    // This constructor uses existing space.
    unbounded(                                          (3)
        CORBA::ULong max,
        CORBA::ULong length,
        CORBA::Long* data,
        CORBA::Boolean release = 0);

    // This constructor allocates space.
    unbounded(CORBA::ULong max);                        (4)

    ~unbounded();                                       (5)
    unbounded& operator = (const unbounded&);           (6)

    static CORBA::Long* allocbuf(                       (7)
        CORBA::ULong nelems);

    static void freebuf(CORBA::Long* data);             (8)

    CORBA::ULong maximum() const;                       (9)

    CORBA::ULong length() const;                       (10)
    void length(CORBA::ULong len);                     (11)

    CORBA::Long& operator[](                           (12)
        CORBA::ULong IT_i);
    const CORBA::Long& operator[](                     (13)
        CORBA::ULong IT_i) const;
};
```

Constructors, Destructor and Assignment

The default constructor (1) sets the sequence length to `0` and sets the maximum length to `0`.

The copy constructor (2) creates a new sequence with the same maximum and length as the given sequence, and copies each of its current elements.

Constructor (3) allows the buffer space for a sequence to be allocated externally to the definition of the sequence itself. Normally sequences manage their own memory. However, this constructor allows ownership of the buffer to be determined by the `release` parameter: `0` (false) means the caller owns the storage, while `1` (true) means that the sequence assumes ownership of the storage. If `release` is `true`, the buffer must have been allocated using the sequence `allocbuf()` function, and the sequence passes

it to freebuf() when finished with it. In general, constructor (3) particularly with the release parameter set to 0, should be used with caution and only when absolutely necessary.

For constructor (3), the type of the data parameter for strings and object references is char* and A_ptr (for interface A) respectively. In other words, string buffers are passed as char** and object reference buffers are passed as A_ptr*.

Constructor (4) allows only the initial value of the maximum length to be set. This allows applications to control how much buffer space is initially allocated by the sequence. This constructor also sets the length to 0.

The destructor (5) automatically frees the allocated storage containing the sequence's elements, unless the sequence was created using constructor (3) with the release parameter set to false. For sequences of strings, CORBA::string_free() is called on each string; for sequences of object references, CORBA::release() is called on each object reference.

Sequence Buffer Management: allocbuf() and freebuf()

The static member functions, allocbuf() (7) and freebuf() (8) control memory allocation for sequence buffers when constructor (3) is used.

The function allocbuf() dynamically allocates a buffer of elements that can be passed to constructor (3) in its data parameter; it returns a null pointer if it cannot perform the allocation.

The freebuf() function deallocates a buffer that was allocated with allocbuf(). The freebuf() function ignores null pointers passed to it. For sequences of array types, the return type of allocbuf() and the argument type of freebuf() are pointers to array slices (refer to "Mapping for Array" on page 73).

When the release flag is set to true and the sequence element type is either a string or an object reference, the sequence individually frees each element before freeing the buffer. It frees strings using string_free(), and it frees object references using release().

Other Functions

The function maximum() (9) returns the total amount of buffer space currently available. This allows applications to know how many items they can insert into an unbounded sequence without causing a reallocation to occur.

The overloaded operators operator[]() (12, 13) return the element of the sequence at the given index. They may not be used to access or modify any element beyond the current sequence length. Before operator[]() is used on a sequence, the length of the sequence must first be set using the modifier function length() (11) function, unless the sequence was constructed using constructor (3).

For strings and object references, operator[]() for a sequence returns a type with the same semantics as the types used for the string and object reference members of structs and arrays, so that assignment to the string or object reference sequence member releases old storage when appropriate.

Unbounded Sequences Example

This section shows how to create the unbounded sequence defined in the following IDL:

```
// IDL
typedef sequence<long> unbounded;
```

You can create an instance of this sequence in any of the following ways:

- Using the default constructor:

  ```
  // C++
  unbounded x;
  ```

  The sequence length is set to `0` and the maximum length is set to `0`. This does not allocate any space for the buffer elements.

- By specifying the initial value for the maximum length of the sequence:

  ```
  // C++
  unbounded y(10);
  ```

  The initial buffer allocation for this sequence is enough to hold ten elements. The sequence length is set to `0`, the maximum is set to `10`.

- Using the copy constructor:

  ```
  // C++
  unbounded c = y;
  ```

  This copies `y`'s state into `c`. The buffer is copied, not shared.

- Dynamically allocating the sequence using the C++ `new` operator:

  ```
  // C++
  unbounded* s1 = new unbounded;
  unbounded* s2 = new unbounded(10);
  ...
  delete s1;
  delete s2
  ```

  By defining a `_var` type, you do not have to explicitly free the sequence when you are finished with it. Like the mapped class, the `_var` type for a sequence provides the `operator[]()`.

  ```
  // C++
  unbounded_var uVar = new unbounded;

  uVar->length(10);
  CORBA::Long i;
  for (i = 0; i<10; i++)
      uVar[i] = i;
  ...
  // Do not call 'delete uVar'.
  ```

- Allocating the buffer space externally to the definition of the sequence itself.

```
// C++
CORBA::Long* data = unbounded::allocbuf(10);
unbounded z(10, 10, data, 1);
CORBA::Long i;
// You can initialize the sequence as follows:
for (i = 0; i<10; i++)
    z[i] = i;
...
z::freebuf(data);
```

In this example, the last parameter to `z`'s constructor is `1`. This indicates that sequence assumes ownership of the buffer. The data buffer is freed automatically when `z` goes out of scope.

If the last parameter were `0`, the data buffer would have to be freed by calling `unbounded::freebuf(data)`.

It is not often necessary to use this form of sequence construction.

Mapping for Bounded Sequences

This section describes the mapping for bounded sequences. For example, consider the following IDL:

```
// IDL
typedef sequence<long, 10> bounded;
```

The corresponding C++ code is as follows:

```
// C++
class bounded {
public:
    bounded();                                          (1)
    bounded(const bounded&);                            (2)
    bounded(CORBA::ULong length,                        (3)
        CORBA::Long* data,
        CORBA::Boolean release = 0);

    ~bounded();                                         (4)

    bounded& operator = (const bounded&);               (5)

    static CORBA::Long* allocbuf(                       (6)
        CORBA::ULong nelems);
    static void freebuf(CORBA::Long* data);             (7)

    CORBA::ULong maximum() const;                       (8)
    CORBA::ULong length() const;                        (9)
    void length(CORBA::ULong len);                     (10)

    CORBA::Long& operator[](                           (11)
        CORBA::ULong IT_i);
    const CORBA::Long& operator[](                     (12)
        CORBA::ULong IT_i) const;
};
```

The mapping is as described for unbounded sequences except for the differences indicated in the following paragraphs.

The maximum length is part of the type and cannot be set or modified.

The `maximum()` function (8) always returns the bound of the sequence as given in its IDL type declaration.

Bounded Sequence Examples

Consider the following IDL declaration:

```
// IDL
typedef sequence<long, 10>  boundedTen;
```

You can declare an instance of `boundedTen` in a variety of ways:

- Using the default constructor:

  ```
  // C++
  boundedTen x;
  ```

  The length of the sequence is set to `0` and the maximum length is set to `10`. Space is allocated in the buffer for `10` elements.

- Using the copy constructor:

  ```
  // C++
  boundedTen c = x;
  ```

  This copies `x`'s state into `c`. The buffer is copied, not shared.

- By dynamically allocating the sequence:

  ```
  // C++
  boundedTen* w = new boundedTen;

  CORBA::Long i;
  w->length(10);
  for (i = 0; i<10; i++)
      (*w)[i] = i;
  ...
  delete w;
  ```

  By defining a `_var` type, you do not have to explicitly free the sequence when you are finished with it. Like the mapped class, the `_var` type for a sequence provides the `operator[]()`. For example:

  ```
  // C++
  boundedTen_var wVar = new boundedTen;

  CORBA::Long i;
  for (i = 0; i<10; i++)
      wVar[i] = i;
  ...
  // Do not call 'delete wVar'.
  ```

- Using constructor (3) as follows:

  ```
  // C++
  CORBA::Long* data = boundedTen::allocbuf(10);
  CORBA::Long i;

  boundedTen z(10, data, 1); // 1 for true.

  // You can initialize the sequence as follows
  // using the overloaded operator[]():
  for (i = 0; i<10; i++)
      z[i] = i;
  ```

  As for unbounded sequences, avoid this form of sequence construction whenever possible. In this example, the `release`

parameter is set to `1` (true) to indicate that sequence `z` is to responsible for releasing the buffer, `data`.

Mapping for Fixed

The fixed type maps to a C++ template class, as shown in the following example:

```
// IDL
typedef fixed <10, 6> ExchangeRate;
const fixed pi = 3.1415926;

// C++
typedef CORBA_Fixed<10, 6> ExchangeRate;
static const CORBA_Fixed
       <(unsigned short)7, (short)6> pi = 3.1415926;
```

The fixed template class is defined as follows:

```
template<unsigned short d, short s> class CORBA_Fixed
{
  public:

    CORBA_Fixed(const int val = 0);
    CORBA_Fixed(const long double val);
    CORBA_Fixed(const CORBA_Fixed<d, s>& val);
    ~CORBA_Fixed();

  operator CORBA_Fixed<d, s> () const;
  operator double() const;

  CORBA_Fixed<d, s>& operator= (const CORBA_Fixed<d, s>& val);
  CORBA_Fixed<d, s>& operator++();
  const CORBA_Fixed<d, s> operator++(int);
  CORBA_Fixed<d, s>& operator--();
  const CORBA_Fixed<d, s> operator--(int);
  CORBA_Fixed<d, s>& operator+() const;
  CORBA_Fixed<d, s>& operator-() const;
  int operator!() const;
  CORBA_Fixed<d, s>& operator+= (const CORBA_Fixed<d, s>& val1);
  CORBA_Fixed<d, s>& operator-= (const CORBA_Fixed<d, s>& val1);
  CORBA_Fixed<d, s>& operator*= (const CORBA_Fixed<d, s>& val1);
  CORBA_Fixed<d, s>& operator/= (const CORBA_Fixed<d, s>& val1);
  const unsigned short Fixed_Digits() const;
  const short Fixed_Scale() const;
```

The class mainly consists of conversion and arithmetic operators to all the fixed types. These types are for use as native numeric types and allow assignment from and to other numeric types.

```
// C++
double rate = 1.4234;
ExchangeRate USRate(rate);

USRate + = 0.1;
cout << "US Exchange Rate = " << USRate << endl;
    // outputs 0001.523400
```

The `Fixed_Digits()` and `Fixed_Scale()` operations return the digits and scale of the fixed type.

A set of global operators for the fixed type is also provided.

Streaming Operators

The streaming operators for fixed are as follows:

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

These operators allow native streaming to `ostreams` and input from `istreams`. This output is padded:

```
// C++
ExchangeRate USRate(1.40);
cout << "US Exchange Rate = " << USRate << endl;
    // outputs 0001.400000
```

Arithmetic Operators

The arithmethic operators for fixed are as follows:

```
CORBA_Fixed<d, s> operator+ (const CORBA_Fixed<d, s>& val1,
                        const CORBA_Fixed<d, s>& val2);
CORBA_Fixed<d, s> operator- (const CORBA_Fixed<d, s>& val1,
                        const CORBA_Fixed<d, s>& val2);
CORBA_Fixed<d, s> operator*  (const CORBA_Fixed<d, s>& val1,
                        const CORBA_Fixed<d, s>& val2);
CORBA_Fixed<d, s> operator/ (const CORBA_Fixed<d, s>& val1,
                        const CORBA_Fixed<d, s>& val2);
```

These operations allow binary arithmetic operations between fixed types. For example:

```
// C++
ExchangeRate USRate(1.453);
ExchangeRate UKRate(0.84);
ExchangeRate diff;

diff = USRate - UKRate;
cout << "difference between US rate and UK rate is "
        << diff << endl;
        // outputs 0000.613000;
```

Logical Operators

The logical operators for fixed are as follows:

```
int operator> (const Fixed<d1, s1>& val1,
                                const Fixed<d2, s2>& val2);
int operator< (const Fixed<d1, s1>& val1,
                                const Fixed<d2, s2>& val2);
int operator>= (const Fixed<d1, s1>& val1,
                                const Fixed<d2, s2>& val2);
int operator>= (const Fixed<d1, s1>& val1,
                                const Fixed<d2, s2>& val2);
int operator== (const Fixed<d1, s1>& val1,
                                const Fixed<d2, s2>& val2);
int operator!= (const Fixed<d1, s1>& val1,
                                const Fixed<d2, s2>& val2);
```

These operators provide logical arithmetic on fixed types. For example:

```
// C++
ExchangeRate USRate(1.453);
ExchangeRate UKRate(0.84);
```

```
if (USRate<= UKRate)
    {
        // Do stuff...
    };
```

Mapping for Array

An IDL array maps to a corresponding C++ array definition. A `_var` type for the array and a `_forany` type, which allows the array to be inserted into and extracted from an `any`, are also generated.

All array indices in IDL and C++ run from `0` to `<size-1>`. If the array element is a `string` or an object reference, the mapping to C++ uses the same rule as for structure members, that is, assignment to an array element releases the storage associated with the old value.

Arrays as Out Parameters and Return Values

Arrays as `out` parameters and return values are handled via a pointer to an *array slice*. An array slice is an array with all the dimensions of the original specified except the first one; for example, a slice of a 2-dimensional array is a 1-dimensional array, a slice of a 1-dimensional array is the element type.

The CORBA IDL to C++ mapping provides a typedef for each array slice type. For example, consider the following IDL:

```
// IDL
typedef long arrayLong[10];
typedef float arrayFloat[5][3];
```

This generates the following array and array slice typedefs:

```
// C++
typedef long arrayLong[10];
typedef long arrayLong_slice;

typedef float arrayFloat[5][3];
typedef float arrayFloat_slice[3];
```

Dynamic Allocation of Arrays

To allocate an array dynamically, you must use functions which are defined at the same scope as the array type. For array `T`, these functions are defined as:

```
// C++
T_slice* T_alloc();
void T_free (T_slice*);
```

The function `T_alloc()` dynamically allocates an array, or returns a null pointer if it cannot perform the allocation. The `T_free()` function deallocates an array that was allocated with `T_alloc()`. For example, consider the following array definition:

```
// IDL
typedef long vector[10];
```

You can use the functions `vector_alloc()` and `vector_free()` as follows:

```
// C++
vector_slice* aVector = vector_alloc();
// The size of the array is as specified
// in the IDL definition. It allocates a 10
// element array of CORBA::Long.
...
vector_free(aVector);
```

Mapping for Typedef

A typedef definition maps to corresponding C++ typedef definitions. For example, consider the following typedef:

```
// IDL
typedef long CustomerId;
```

This generates the following C++ typedef:

```
// C++
typedef CORBA::Long CustomerId;
```

Mapping for Constants

Consider a global, file level, IDL constant such as:

```
// IDL
const long MaxLen = 4;
```

This maps to a file level C++ `static const`:

```
// C++
static const CORBA::Long MaxLen = 4;
```

An IDL constant in an interface or module maps to a C++ `static const` member of the corresponding C++ class. For example:

```
// IDL
interface CheckingAccount : Account {
    const float MaxOverdraft = 1000.00;
};
```

This maps to the following C++:

```
// C++
class CheckingAccount : public virtual Account {
public:
    static const CORBA::Float MaxOverdraft;
};
```

The following definition is also generated for the value of this constant, and is placed in the client stub implementation file:

```
// C++
const CORBA::Float
    CheckingAccount::MaxOverdraft = 1000.00;
```

Mapping for Pseudo-Object Types

For most pseudo-object types, the CORBA specification defines an operation to create a pseudo-object. For example, the pseudo-interface `ORB` defines the operations `create_list()` and `create_operation_list()` to create an `NVList` (an `NVList` describes the arguments of an IDL operation) and operation `create_environment()` to create an `Environment`.

To provide a consistent way to create pseudo-objects, in particular, for those pseudo-object types for which the CORBA specification does not provide a creation operation, Orbix provides static `IT_create()` function(s) for all pseudo-object types in the corresponding C++ class. These functions provide an Orbix-specific means to create and obtain a pseudo-object reference. An overloaded version of `IT_create()` is provided that corresponds to each C++ constructor defined on the class. `IT_create()` should be used in preference to C++ operator `new` but only where there is no suitable compliant way to obtain a pseudo-object reference. Use of `IT_create()` in preference to `new` ensures memory management consistency.

The ***Orbix Programmer's Reference C++ Edition*** gives details of the `IT_create()` functions available for each pseudo-interface. The entry for `IT_create()` also indicates the compliant way, if any, of obtaining an object reference to a pseudo-object.

Memory Management and _var Types

This section describes the `_var` types that help you to manage memory deallocation for some IDL types. The Orbix IDL compiler generates `_var` types for the following:

- Each `interface` type.
- Type `string`.
- All variable length complex data types; for example, an array or sequence of `string`s, and structs of variable data length.
- All fixed length complex data types, for consistency with variable length types.

Conceptually, a `_var` type can be considered as an abstract pointer that assumes ownership of the data to which it points.

For example, consider the following interface definition:

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

The following C++ code illustrates the functionality of a `_var` type for this interface:

```
// C++
{
    // Set aPtr to refer to an object:
    A_ptr aPtr = ...
    A_var aVar = aPtr;

    // Here, aVar assumes ownership of aPtr.
    // The object reference is not duplicated.
```

```
    aVar->op();
    ...
}
    // Here, aVar is released (its
    // reference count decremented).
```

The general form of the `_var` class for IDL type `T` is:

```
// C++
class T_var {
public:
    T_var();                                        (1)
    T_var(T_ptr IT_p);                              (2)
    T_var(const T_var& IT_s);                       (3)
    T_var& operator = (T_ptr IT_p);                 (4)
    T_var& operator = (const T_var& IT_s);          (5)
    ~T_var();                                       (6)
    T* operator->();                                (7)
};
```

Constructors and Destructor

The default constructor (1) creates a `T_var` containing a null pointer to its data or a nil object reference as appropriate. A `T_var` initialized using the default constructor can always legally be passed as an `out` parameter.

Constructor (2) creates a `T_var` that, when destroyed, frees the storage pointed to by its parameter. The parameter to this constructor should never be a null pointer. Orbix does not detect null pointers passed to this constructor.

The copy constructor (3) deep-copies any data pointed to by the `T_var` constructor parameter. This copy is freed when the `T_var` is destroyed or when a new value is assigned to it.

The destructor frees any data pointed to by the `T_var` `string`s and array types are deallocated using the `CORBA::string_free()` and `S_free()` (for array of type `S`) deallocation functions respectively; object references are released.

The following code illustrates some of these points:

```
// C++
{
    A_var aVar = ...
    String_var sVar = string_alloc(10);
    ...
    aVar->op();
    ...
} // Here, aVar is released,
  // sVar is freed.
```

Assignment Operators

The assignment operator (4) results in any old data pointed to by the `T_var` being freed before assuming ownership of the `T*` (or `T_ptr`) parameter. For example:

```
// C++
// Set aVar to refer to an object reference.
A_var aVar = ...

// Set aPtr to refer to an object reference.
A_ptr aPtr = ...
```

```
// The following assignment causes the _ptr
// owned by aVar to be released before aVar
// assumes ownership of aPtr.
aVar = aPtr;
```

The normal assignment operator (5) deep-copies any data pointed to by the `T_var` assignment parameter. This copy is destroyed when the `T_var` is destroyed or when a new value is assigned to it.

```
// C++
{
    T_var t1Var = ...
    T_var t2Var = ...

    // The following assignment frees t1Var and
    // deep copies t2Var, duplicating its
    // object reference.
    t1Var = t2Var;
}

// Here, t1Var and t2Var are released. They both ///
refer to the same object so the reference count
// of the object is decremented twice.
```

Assignment between `_var` types is only allowed between `_var`s of the same type. In particular, no widening or narrowing is allowed. Thus the following assignments are illegal:

```
// C++
// B is a derived interface of A.
A_var aVar = ...
B_var bVar = ...
aVar = bVar; // ILLEGAL.
bVar = aVar; // ILLEGAL.
```

You cannot create a `T_var` from a `const T*`, or assign a `const T*` to a `T_var`. Recall that a `T_var` assumes ownership of the pointers passed to it and frees this pointer when the `T_var` goes out of scope or is otherwise freed. This deletion cannot be done on a `const T*`. To allow construction from a `const T*` or assignment to a `T_var`, the `T_var` would have to copy the const object. This copy is forbidden by the standard C++ mapping, allowing the application programmer to decide if a copy is really wanted or not. Explicit copying of `const T*` objects into `T_var` types can be achieved via the copy constructor for `T`, as shown below:

```
// C++
const T* t = ...;
T_var tVar = new T(*t);
```

operator->()

The overloaded `operator->()` (7) returns the `T*` or `T_ptr` held by the `T_var`, but retains ownership of it. You should not call this function unless the `T_var` has been initialized with a valid `T*` or `T_var`.

For example:

```
// C++
A_var aVar;
// First initialize aVar.
aVar = ... // Perhaps an object reference
           // returned from the Naming Service.
// You can now call member functions.
aVar->op();
```

The following are some examples of illegal code:

```
// C++
A_var aVar;
aVar->op(); // ILLEGAL! Attempt to call function
            // on uninitialized _var.
A_ptr aPtr;
aPtr = aVar; // ILLEGAL! Attempt to convert
             // uninitialized _var. Orbix does
             // not detect this error.
```

The second example above is illegal because an uninitialized `_var` contains no pointer, and thus cannot be converted to a `_ptr` type.

Memory Management for Parameters

When passing operation parameters between clients and objects in a distributed application, you must ensure that memory leakage does not occur. Since main memory pointers cannot be meaningfully passed between hosts in a distributed system, the transmission of a pointer to a block of memory requires the block to be transmitted by value and re-constructed in the receiver's address space. You must take care not to cause memory leakage for the original or the new copy.

This section explains the mapping for parameters and return values and explains the memory management rules that clients and servers must follow to ensure that memory is not leaked in their address spaces.

Passing basic types, enums, and fixed length structs as parameters is quite straightforward in Orbix. However, you must be careful when passing strings and other variable length data types, including object references.

in Parameters

When passing an `in` parameter, a client programmer allocates the necessary storage and provides a data value. Orbix does not automatically free this storage on the client side.

For example, consider the following IDL operation:

```
// IDL
interface A {
    ...
};

interface B {
    void op(in float f, in string s, in A a);
};
```

A client can call operation `op()` as follows:

```
// C++
{
    CORBA::Float f = 12.0;
    char* s = CORBA::string_alloc(4);
    strcpy(s, "Two");
    A_ptr aPtr = ...
    B_ptr bPtr = ...
    bPtr->op(f, s, aPtr);
    ...
    CORBA::string_free(s);
    CORBA::release(aPtr);
    CORBA::release(bPtr);
}
```

On the server side, the parameter is passed to the function that implements the IDL operation. Orbix frees the parameter upon completion of the function call in order to avoid a memory leak. If you wish to keep a copy of the parameter in the server, you must copy it before the implementation function returns.

This is illustrated in the following implementation function for operation `op()`:

```
// C++
void B_i::op(CORBA::Float f, const char* s,
    A_ptr a, CORBA::Environment&) {
    ...
    // Retain in parameters.
    // Copy the string and maybe assign it to
    // member data:
    char* copy = CORBA::string_alloc(strlen(s));
    strcpy(copy, s);
    ...

    // Duplicate the object reference:
    A::_duplicate(a);
}
```

Note: A client program should not pass a NULL or uninitialized pointer for an `in` parameter type that maps to a pointer (`*`) or a reference to a pointer (`*&`).

inout Parameters

In the case of `inout` parameters, a value is both passed from the client to the server and vice versa. Thus, it is the responsibility of the client programmer to allocate memory for a value to be passed in.

In the case of variable length types, the value being passed back out from the server is potentially longer than the value which was passed in. This leads to memory management rules that you must examine on a type-by-type basis.

Object Reference inout Parameters

On the client side, the programmer must ensure that the parameter is a valid object reference that actually refers to an object. In particular, when passing a `T_var` as an `inout` parameter, where `T` is an interface type, the `T_var` should be initialized to refer to some object.

If the client wishes to continue to use the object reference being passed in as an `inout` parameter, it must first duplicate the reference. This is because the server can modify the object reference to refer to something else when the operation is invoked. If this were to happen, the object reference for the existing object would be automatically released.

On the server side, the object reference is made available to the programmer for the duration of the function call. The object referenced is automatically released at the end of the function call. If the server wishes to keep this reference, it must duplicate it.

The server programmer is free to modify the object reference to refer to another object. To do so, you must first release the *existing* object reference using `CORBA::release()`. Alternatively, you can release the existing object reference by assigning it to a local `_var` variable, for example:

```
// C++
// Server code.
void B_i::opInout(CORBA::Float& f,
        char*& s, A_ptr& a,
        CORBA::Environment&) {
    A_var aTempVar = a;
    a = ... // New object reference.
    ...
}
```

Any previous value held in the `_var` variable is properly deallocated at the end of the function call.

String inout Parameters

On the client side, you must ensure that the parameter is a valid `NUL`-terminated `char*`. It is your responsibility to allocate storage for the passed `char*`. This storage must be allocated via `string_alloc()`.

After the operation has been invoked, the `char*` may point to a different area of memory, since the server is free to deallocate the input string and reassign the `char*` to point to new storage. It is your responsibility to free the storage when it is no longer needed.

On the server side, the string pointed to by the `char*` which is passed in may be modified before being implicitly returned to the client, or the `char*` itself may be modified. In the latter case, it is your responsibility to free the memory pointed to by the `char*` before reassigning the parameter. In both cases, the storage is automatically freed at the end of the function call. If the server wishes to keep a copy of the string, it must take an explicit copy of it.

An alternative way to ensure that the storage for an `inout` string parameter is released is to assign it to a local `_var` variable, for example:

```
// C++
// Server code.
void B_i::opInout(CORBA::Float& f,
        char*& s, A_ptr& a,
        CORBA::Environment&) {
    String_var sTempVar = s;
    s = ... // New string.
    ...
}
```

Any previous value held in the `_var` variable is properly deallocated at the end of the function call.

For unbounded strings, the server programmer is free to pass a string back to the client that is longer than the string which was passed in. Doing so would, of course, cause an automatic reallocation of memory at the client side to accommodate the new string.

Sequence inout Parameters

On the client side, you must ensure that the parameter is a valid sequence of the appropriate type. Recall that this sequence may have been created with either '`release = 0`' (false) semantics or '`release = 1`' (true) semantics. In the former case, the sequence is *not* responsible for managing its own memory. In the latter case, the sequence frees its storage when it is destroyed, or when a new value is assigned into the sequence.

In all cases, it is the responsibility of the client programmer to release the storage associated with a sequence passed back from a server as an `inout` parameter.

On the server side, Orbix is unaware of whether the incoming sequence parameter was created with `release = 0` or `release = 1` semantics, since this information is not transmitted as part of a sequence. Orbix must assume that `release` is set to `1`, since failure to release the memory could result in a memory leak.

The sequence is made available to the server for the duration of the function call, and is freed automatically upon completion of the call. If the server programmer wishes to use the sequence after the call is complete, the sequence must be copied.

A server programmer is free to modify the contents of the sequence received as an `inout` parameter. In particular, the length of the sequence that is passed back to the client is not constrained by the length of the sequence that was passed in.

Where possible, use only sequences created with `release = 1` as `inout` parameters.

Type any inout Parameters

The memory management rules for `inout` parameters of type `any` are the same as those for sequence parameters as described above.

There is a constructor for type `CORBA::Any` which has a `release` parameter, analogous to that of the sequence constructors (refer to the chapter "The Any Data Type" on page 173). However, the warning provided above in relation to `inout` sequence parameters does not apply to type `any`.

Other inout Parameters

For all other types, including variable length unions, arrays and structs, the rules are the same.

The client must make sure that a valid value of the correct type is passed to the server. The client must allocate any necessary storage for this value, except that which is encapsulated and managed within the parameter itself. The client is responsible for freeing any storage associated with the value passed back from the server in the `inout` parameter, except that which is managed by the parameter itself. This client responsibility is alleviated by the use of `_var` types, where appropriate.

The server is free to change any value which is passed to it as an `inout` parameter. The value is made available to the server for the duration of the function call. If the server wishes to continue to use the memory associated with the parameter, it must take a copy of this memory.

out Parameters

A client program passes an `out` parameter as a pointer. A client may pass a reference to a pointer with a null value for `out` parameters because the server does not examine the value but instead just overwrites it.

The client programmer is responsible for freeing storage returned to it via a variable length `out` parameter. The memory associated with a variable length parameter is properly freed if a `_var` variable is passed to the operation.

For example, consider the following IDL:

```
// IDL
struct VariableLengthStruct {
    string aString;
};

struct FixedLengthStruct {
    float aFloat;
};

interface A {
    void opOut(out float f,
        out FixedLengthStruct fs,
        out VariableLengthStruct vs);
};
```

The operation `opOut()` is implemented by the following C++ function:

```
// C++
A_i::opOut(
    CORBA::Float& f,
    FixedLengthStruct& fs,
    VariableLengthStruct*& vs,
    CORBA::Environment&) {
    ...
}
```

A client calls this operation as follows:

```
// C++
{
    FixedLengthStruct_var fs;
    VariableLengthStruct_var vs;
    A_var aVar = ...;
    aVar->opOut(fs, vs);
    aVar->opOut(fs, vs); // 1st results freed.
} // 2nd results freed.
```

The client must explicitly free memory if `_var` types are not used.

A fixed-length struct `out` parameter maps to a struct reference parameter. A variable-length struct `out` parameter maps to a reference to a pointer to a struct. Since the `_var` type contains conversion operators to both of these types, the difference in the mapping for `out` parameters for fixed length and variable length structs is hidden. If `_var` types are not used, you must use a different syntax when passing fixed and variable length structs. For example:

```
// C++
{
    //You must allocate memory for a fixed
    //length struct
    FixedLengthStruct fs;

    //No need to initialize memory for a variable
    //length struct
    VariableLengthStruct* vs_p;
    aVar->opOut(fs, vs_p)

    // Use fs and vs_p.
    ...

    // Free pointer vs_p before passing it to
    // A_i::opOut() again.
    delete vs_p;
    aVar->opOut(*fs, vs_p);

    // Use fs and vs_p.
    ...


// Delete memory pointed to by vs_p
    delete vs_p;
}
```

On the server side, the storage associated with `out` parameters is freed by Orbix when the function call completes. The programmer must retain a copy (or duplicate an object reference) to retain the value. For example:

```
// C++
A_i::opOut(
        CORBA::Float& f,
        FixedLengthStruct& fs,
        VariableLengthStruct*& vs,
        CORBA::Environment&) {
    // To retain the variable length struct:
    VariableLengthStruct* myVs =
        new VariableLengthStruct(*vs);
    ...
}
```

In this example, you take a copy of the struct parameter by using the default C++ copy constructor.

A server may not return a null pointer for an `out` parameter returned as a `T*` or `T*&`—that is, for a variable length struct or union, a sequence, a variable length or fixed length array, a string or any.

In all cases, the client is responsible for releasing the storage associated with the `out` parameter when the value is no longer required. This responsibility can be eased by associating the storage with a `_var` type, where appropriate, which assumes responsibility for its management.

Return Values

The rules for managing the memory of return values are the same as those for managing the memory of `out` parameters, with the exception of fixed-length arrays. A fixed-length array `out` parameter maps to a C++ array parameter, whereas a fixed-length array return value maps to a pointer to an array slice. The server should set the pointer to a valid instance of the array. This cannot be a null pointer. It is the responsibility of the client to release the storage associated with the return value when the value is no longer required.

An Example of Applying the Rules for Object References

An important example of the parameter passing rules arises in the case of object references. Consider the following IDL definitions:

```
// IDL
interface I1 {
};
interface I2 {
    I1 op(in I1 par);
};
```

The following implementation of operation `I2::op()` is incorrect:

```
// C++
I1_ptr I2::op(I1_ptr par) {
    return par;
}
```

If the object referenced by the parameter `par` does not exist in the server process's address space before the call, Orbix creates a proxy for this object within that address space. This object initially has a reference count of one. At the end of the call to `I2::op()`, this count is decremented twice—once because `par` is an `in` parameter, and once because it is also a return value. The code therefore tries to return a reference that is found by attempting to access a proxy that no longer exists—with undefined results.

A similar error in reference counts results if the object (or its proxy) referenced by the parameter `par` already exists in the server process's address space.

The correct coding of `I2::op()` is:

```
// C++
I1_ptr I2::op(I1_ptr par) {
    return I1::_duplicate(par);
}
```

Using and Implementing IDL Interfaces

This chapter describes how servers create objects that implement IDL interfaces, and shows how clients access these objects through IDL interfaces. This chapter shows how to use and implement CORBA objects through a detailed description of the banking application introduced in "Getting Started With Orbix".

Overview of an Example Application

In the `BankSimple` example, an Orbix server creates a single distributed object that represents a bank. This object manages other distributed objects that represent customer accounts at the bank.

A client contacts the server by getting a reference to the bank object. This client then calls operations on the bank object, instructing the bank to create new accounts for specified customers. The bank object creates account objects in response to these requests and returns them to the client. The client can then call operations on these new account objects.

This application design, where one type of distributed object acts as a factory for creating another type of distributed object, is very common in CORBA.

The source code for the example described in this chapter is available in the `demos\banksimple` directory of your Orbix installation.

Overview of the Programming Steps

1. Define IDL interfaces to your application objects.
2. Compile the IDL interfaces.
3. Implement the IDL interfaces with C++ classes.
4. Write a server program that creates instances of the implementation classes. This involves:
   i. Initializing the ORB.
   ii. Creating initial implementation objects.
   iii. Allowing Orbix to receive and process incoming requests from clients.
5. Write a client program that accesses the server objects. This involves:
   i. Initializing the ORB.
   ii. Getting a reference to an object.
   iii. Invoking object attributes and operations.
6. Compile the client and server.
7. Run the application. This involves:
   i. Running the Orbix daemon process.
   ii. Registering the server in the Implementation Repository.
   iii. Running the client.

Defining IDL Interfaces

This example uses two IDL interfaces: an interface for the bank object created by the server and an interface that allows clients to access the account objects created by the bank.

The IDL interfaces are called `Bank` and `Account`, defined as follows:

```
// IDL
// In banksimple.idl

module BankSimple {

        typedef float CashAmount;
        interface Account;
            // A factory for bank accounts.
    interface Bank {
       // Create new account with specified name.
       Account create_account(in string name);
       // Find the specified named account.
       Account find_account(in string name);
    };

    interface Account {
       readonly attribute string name;
       readonly attribute CashAmount balance;

       void deposit(in CashAmount amount);
       void withdraw(in CashAmount amount);
    };
    };
```

The server creates a `Bank` object that accepts operation calls such as `create_account()` from clients. The operation `create_account()` instructs the `Bank` object to create a new `Account` object in the server. The operation `find_account()` instructs the `Bank` object to find an existing `Account` object.

In this example, all of the objects (both `Bank` and `Account` objects) are created in a single server process. A real system could use several different servers and many server processes.

For details on how to compile your IDL interfaces, refer to "Compiling IDL Interfaces".

Implementing IDL Interfaces

This section describes in detail the mechanisms enabling you to define C++ classes to implement IDL interfaces. To implement an IDL interface, you must provide a C++ class that includes member functions corresponding to the operations and attributes of the IDL interface. Orbix supports two mechanisms for relating an implementation class to its IDL interface:

- The *BOAImpl* approach.
- The *TIE* approach.

Most server programmers use one of these approaches exclusively, but you can use both in the same server. Client programmers do not need to be concerned with which of these mechanisms is used.

The BOAImpl Approach to Implementing Interfaces

For each IDL interface, Orbix generates a C++ class with the same name. Orbix also generates a second C++ class for each IDL interface, taking the name of the interface with `BOAImpl` appended. For example, it generates the class `AccountBOAImpl` for the IDL interface `Account`, and the class `BankBOAImpl` for the IDL interface `Bank`. To indicate that a C++ class implements a given IDL interface, that class should inherit from the corresponding BOAImpl-class.

Each BOAImpl class inherits from a corresponding IDL Compiler-generated C++ class; for example, `AccountBOAImpl` inherits from `Account`. BOAImpl classes inherit from each other in the same way that the corresponding IDL interfaces do.

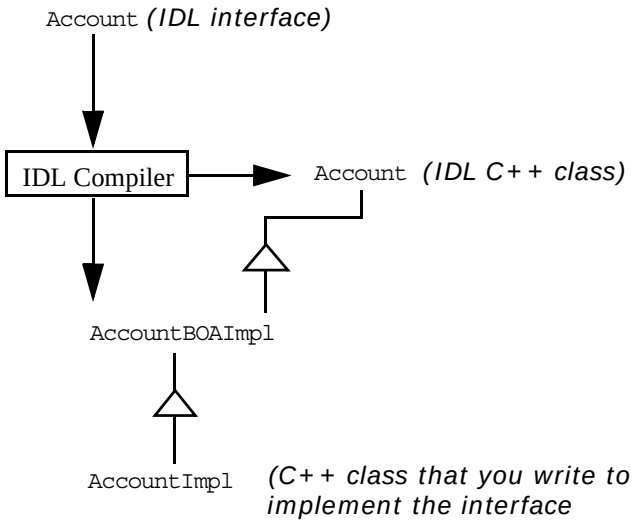


Figure 9: *The BOAImpl Approach to Defining a C++ Implementation Class*

The BOAImpl approach is shown in Figure 9 for the `Account` IDL interface. For simplicity, the fully-scoped name (`BankSimple::Account`) is not used.

The Orbix IDL compiler produces the C++ classes `Account` and `AccountBOAImpl`. You define a new class, `AccountImpl`, that implements the functions defined in the IDL interface. In addition to functions that correspond to IDL operations and attributes, class `AccountImpl` can contain user-defined constructors, a destructor, and private and protected members.

Note: This guide uses the convention that interface `A` is implemented by class `AImpl`. It is not necessary to follow this naming scheme. In any case, some applications might need to implement interface `A` several times.

The TIE Approach to Implementing Interfaces

Using the TIE approach, you can implement the IDL operations and attributes in a class that does *not* inherit from the BOAImpl class. In this case, you must indicate to Orbix that the class implements a particular IDL interface by using a C++ macro to *tie together* your class and the IDL interface.

To use the `TIE` mechanism, the server programmer indicates that a particular class implements a given IDL C++ class by calling a `DEF_TIE` macro, which has the general form:

`DEF_TIE_`*IDL C++ class name* `(`*implementation class name*`)`

Each call to this macro defines a `TIE` class. This class records that a particular IDL C++ class is implemented by a particular implementation class. Consider the macro call:

```
DEF_TIE_Account(AccountImpl)
```

This generates a class named `TIE_Account(AccountImpl)`. Figure 10 shows the TIE approach. For simplicity, the fully scoped name, `BankSimple::Account`, is not used.

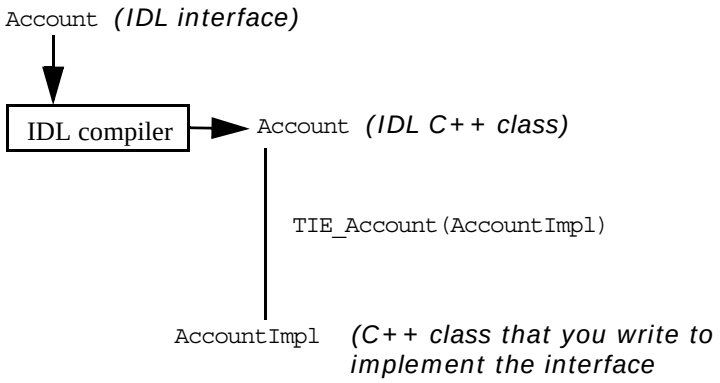


Figure 10: *The TIE Approach to Implementing Interfaces*

`DEF_TIE` macros also work when interfaces are defined in IDL modules. For example, if interface `I` is defined in module `M`, the macros take the following form:

`DEF_TIE_M_Impl (`*implementation class name*`)`
`TIE_M_Impl (`*implementation class name*`)`

For example, interface `Account` is defined in module `BankSimple` and implemented by C++ class `AccountImpl`. The macros thus take the following form:

- `DEF_TIE_BankSimple_Account(BankSimple_AccountImpl)`

  This macro is called in the implementation header file (in this case, `banksimple_accountimpl.h`).

- `TIE_BankSimple_Account(BankSimple_AccountImpl)`

  This macro is called in the implementation file (in this case, `banksimple_bankimpl.cxx`).

Refer to "Using the TIE Approach" on page 95 for more details.

Defining Implementation Classes for IDL Interfaces

This section illustrates both the BOAImpl and TIE approaches. Two implementation classes are required:

BankSimple_BankImpl	Implements the Bank interface.
BankSimple_AccountImpl	Implements the Account interface.

Note: You can automatically generate a skeleton version of the class and function definitions for BankSimple::BankImpl and BankSimple::AccountImpl by specifying the -S switch to the IDL compiler.

The -S switch produces two files. If the IDL definitions are in the file banksimple.idl, the skeleton definitions are placed in the following files:

banksimple_ih	This is the class header file that defines the class. This file declares the member functions that you must implement. It can be renamed to banksimple_bankimpl.h.
banksimple.ic	This is the code file. It gives an empty body for each member function and can be renamed to banksimple_bankimpl.cxx.

You can edit both files to provide a full implementation class. You must add member variables, constructors, and destructors. Other member functions can be added if required. You can use either the BOAImpl or the TIE approach to relate the implementation classes to your IDL C++ classes.

Using the BOAImpl Approach

Using this approach, you should indicate that a class implements a specific IDL interface by inheriting from the corresponding BOAImpl-class generated by the IDL compiler:

```
// C++
// In file banksimple_accountimpl.h

#define BANKSIMPLE_ACCOUNTIMPL_H_
#include "banksimple.hh"

// The Account implementation class.
class BankSimple_AccountImpl :
            public virtual BankSimple::AccountBOAImpl {
    public:
        // IDL operations
        virtual void deposit
            (BankSimple::CashAmount amount,
                CORBA::Environment&);
        virtual void withdraw
            (BankSimple::CashAmount amount,
                CORBA::Environment&);

        // IDL attributes
        virtual char* name(CORBA::Environment&);
        virtual void name
            (const char*  _new_value, CORBA::Environment&);
```

```
        virtual BankSimple::CashAmount balance
            (CORBA::Environment&);

        // C++ operations
        BankSimple_AccountImpl
            (const char* name, BankSimple::CashAmount
                balance);
        virtual ~BankSimple_AccountImpl();

    protected:
        CORBA::String_var m_name;
        BankSimple::CashAmount m_balance;
        ...
};

// C++
// In file banksimple_bankimpl.h.

#define BANKSIMPLE_BANKIMPL_H_
#include <banksimple.hh>

// The Bank implementation class.
class BankSimple_BankImpl : public virtual
   BankSimple::BankBOAImpl {
    public:
        // IDL operations.
        virtual BankSimple::Account_ptr
        create_account(const char* name,
                CORBA::Environment&);
        virtual BankSimple::Account_ptr
        find_account(const char* name,
                CORBA::Environment&);

        // C++ operations.
        BankSimple_BankImpl();
        virtual ~BankSimple_BankImpl();

        protected:
        // This bank stores account in an array in memory.
        static const int MAX_ACCOUNTS;
        BankSimple::Account_var* m_accounts;
        ....
};
```

Note: The BOAImpl class is produced only if the `-B` switch is specified to the IDL compiler.

Classes `BankSimple_BankImpl` and `BankSimple_AccountImpl` redefine each of the functions inherited from their respective BOAImpl classes. They can also add constructors, destructors, member functions and member variables. Virtual inheritance is not strictly necessary in the code shown; it is used in case C++ multiple inheritance is required later. Any function inherited from the BOAImpl class is virtual because it is defined as virtual in the BOAImpl class. Therefore, it is not strictly necessary to explicitly mark them as virtual in an implementation class (for example, `BankSimple_AccountImpl`).

The accounts managed by a bank are stored in a array with members of type `BankSimple::Account_var`.

Outline of the Bank Implementation (BOAImpl Approach)

First, in `BankSimple_BankImpl::create_account()`, you should construct a new `BankSimple::Bank` object. The function `create_account()` corresponds to an IDL operation, and its return value is of type `BankSimple::Account_ptr`:

```
// C++
// In file banksimple_bankimpl.cxx.

// Add a new account.
BaBankSimple::Account_ptr
BankSimple_AccountImpl::create_account
    (const char* name, CORBA::Environment&) {

    int i = 0;
    for ( ; i < MAX_ACCOUNTS&  !CORBA::is_nil(m_accounts[i]);
   ++i)
    { }

    if (i < MAX_ACCOUNTS){
        // Create an account with zero balance.
        m_accounts[i]=new BankSimple_AccountImpl(name, 0.0);
        cout << "create_account: Created with name:" << name <<
   endl;
        return BankSimple::Account::_duplicate(m_accounts[i]);
    }
    else {
        // Cannot create an account, return nil.
        cout << "create_account: failed, no space left!" << endl;
        return BankSimple::Account::_nil();
    }
}
```

You must call `BankSimple::Account::_duplicate()` because Orbix calls `CORBA::release()` on any object returned as an `out`/`inout` parameter or as a return value. The reference count on the new object is initially one, and subsequently calling `CORBA::release()` without first calling `BankSimple::Account::_duplicate()` results in deletion of the object.

Using the BOAImpl approach, the `Bank` implementation code is as follows:

```
// C++
// In file bankSimple_bankImpl.cxx.

// Implementation of the BankSimple::Bank interface.
#include "banksimple_bankimpl.h"
#include "banksimple_accountimpl.h"

// Maximum number of accounts handled by the bank.
const int BankSimple_BankImpl::MAX_ACCOUNTS = 1000;

// BankSimple_BankImpl constructor.
BankSimple_BankImpl::BankSimple_BankImpl():
    m_accounts(new BankSimple::Account_var[MAX_ACCOUNTS]) {

    // Make sure all accounts are nil.
    for (int i = 0; i < MAX_ACCOUNTS; ++i) {
        m_accounts[i] = BankSimple::Account::_nil();
    }
```

```
    // BankSimple_BankImpl destructor.
    BankSimple_BankImpl::~BankSimple_BankImpl() {
        delete [] m_accounts;
    }

    // Add a new account.
    BankSimple::Account_ptr
    BankSimple_AccountImpl::create_account
        (const char* name, CORBA::Environment&) {
    ...
    }

    // Find a named account
    BankSimple::Account_ptr
    BankSimple_BankImpl::find_account
        (const char*  name, CORBA::Environment&) {
    ...
    }
}
```

In this example, the possibility of making the server objects persistent is ignored. You can do this by storing the account and bank data in files or in a database. Refer to the chapter "Loading Objects at Runtime" on page 267 for more details.

Using the TIE Approach

Using the TIE Approach, an implementation class does not have to inherit from any particular base class. Instead, a class implements a specific IDL interface by using the `DEF_TIE` macro.

The DEF_TIE Macro

A version of the `DEF_TIE` macro is available for each IDL C++ class. The macro takes one parameter—the name of a C++ class implementing this interface:

```
// C++
// In file banksimple_accountimpl.h
class BankSimple_AccountImpl {
... // As before.
};

// DEF_TIE Macro call.
DEF_TIE_BankSimple_Account(BankSimple_AccountImpl)
```

This macro call defines a `TIE` class that indicates that class `BankSimple_AccountImpl` implements interface `BankSimple::Account`.

```
// C++
// In file banksimple_bankimpl.h

class BankSimple_BankImpl {
    . . .  // As before.
};

// DEF_TIE Macro call.
DEF_TIE_BankSimple_Bank(BankSimple_BankImpl)
```

This macro call defines a `TIE` class which indicates that class `BankSimple_BankImpl` implements interface `BankSimple::Bank`.

The TIE Class

The `TIE_BankSimple_Account(BankSimple_AccountImpl)` construct is a preprocessor macro call that expands to the name of a C++ class representing the relationship between the `BankSimple::Account` and `BankSimple_AccountImpl` classes. This class is defined by the macro call `DEF_TIE_BankSimple_Account(BankSimple_AccountImpl)`. Its constructor takes a pointer to a `BankSimple_AccountImpl` object as a parameter.

The C++ class generated by calling the macro `TIE_BankSimple_Account(BankSimple_AccountImpl)` has a name that is a legal C++ identifier, but you do not need to use its actual name. You should use the macro call `TIE_BankSimple_Account(BankSimple_AccountImpl)` when you wish to use this class.

The TIE approach gives a *complete* separation of the class hierarchies for the IDL Compiler-generated C++ classes and the class hierarchies of the C++ classes used to implement the IDL interfaces.

Consider an IDL operation that returns a reference to an `Account` object; for example, `BankSimple::Bank::create_account()`. In the IDL C++ class, this is translated into a function returning an `BankSimple::Account_ptr`.

However, using the TIE approach, the actual object to which a reference is returned is of type `BankSimple_AccountImpl`. This is not a derived class of `BankSimple::Account`. Therefore, the server should create an object of type `TIE_BankSimple_Account(BankSimple_AccountImpl)`. This `TIE` object references the `BankSimple_AccountImpl` object, and a reference to the `TIE` object should be returned by the function. This is because the class `TIE_BankSimple_Account(BankSimple_AccountImpl)` is a derived class of class `BankSimple::Account`. All invocations on the TIE object are automatically forwarded by it to the associated `BankSimple_AccountImpl` object.

When you code the server you create the `BankSimple_AccountImpl` object and a `TIE` object. The server should then use the `TIE` object, rather than the `BankSimple_AccountImpl` object directly. A bank's linked list of accounts, for example, should then point to TIE objects, rather than directly pointing to the `BankSimple_AccountImpl` objects.

A `TIE` object automatically delegates all incoming operation calls to its corresponding implementation object. For example, all invocations on a `TIE_Account(BankSimple_AccountImpl)` object are automatically passed to the `BankSimple_AccountImpl` object to which the `TIE` object holds a pointer.

Note: By default, calling `CORBA::release()` on a `TIE` object with a reference count of one also deletes the referenced object. The `TIE` object's destructor calls the `delete` operator on the implementation object pointer it holds. This is usually the desired behavior; however, you can use `CORBA::BOA::propagateTIEdelete()` to specify whether the `TIE` object should be deleted. Refer to the ***Orbix Programmer's Reference C++ Edition*** for more details.

Using the TIE approach, the bank service header file might look as follows:

```
// C++
// In file banksimple_bankimpl.h

#define BANKSIMPLE_BANKIMPL_H_
#include <banksimple.hh>

class BankSimple_BankImpl {
public:
    // IDL-defined operations.
    virtual BankSimple::Account_ptr
    create_account(const char*  name, CORBA::Environment&);

    virtual BankSimple::Account_ptr
    find_account(const char*  name, CORBA::Environment&);
    // C++ operations.
    BankSimple_BankImpl();
    virtual ~BankSimple_BankImpl();

protected:
    // This bank steored accounts in an array of Account_var.
    static const int MAX_ACCOUNTS;
    BankSimple::Account_var* m_accounts;
};

// Indicate that BankSimple_BankImp implements
// IDL interface BankSimple::Account.
DEF_TIE_BankSimple_Account(BankSimple_BankImpl)
```

Outline of the Bank Implementation (TIE Approach)

An outline of the code for `BankSimple_BankImpl::create_account()` is shown below:

```
// C++
// In file banksimple_bankimpl.cxx

BankSimple::Account_ptr BankSimple_BankImpl::create_account
    (const char*  name, CORBA::Environment&) {

    // Ensure that a valid account name is found.
    int i = 0;
    for ( ; i < MAX_ACCOUNTS& & CORBA::is_nil(m_accounts[i])
            ++i ) {
        ...
    }

    if (i < MAX_ACCOUNTS) {
        // Create an account with zero balance.
        m_accounts[i] =
             new TIE_BankSimple_Account(BankSimple_AccountImpl)
                        (new BankSimple_AccountImpl(name, 0.0));
    ...
    }
    else {
        ... // Cannot create account, return nil.
    }
};
```

The `BankSimple::Account_ptr` is initialized to reference a `TIE` object that points in turn to the new `BankSimple_AccountImpl` object.

Note: The object that a TIE object points to *must* be dynamically allocated using C++ operator `new`. By default, when a TIE object is destroyed, it deletes the object that it points to. The object must therefore be dynamically allocated.

Using the TIE approach, the Bank implementation class code is:

```
// C++
// In file banksimple_bankimpl.cxx

// Implementation of the BankSimple::Bank interface.
#include "banksimple_bankimpl.h"
#include "banksimple_bccountimpl.h"

const int BankSimple_BankImpl::MAX_ACCOUNTS = 1000;

// Constructor.
BankSimple_BankImpl::BankSimple_BankImpl():
    m_accounts(new BankSimple::Account_var[MAX_ACCOUNTS]) {

    // Make sure all accounts are nil.
    for (int i = 0; i < MAX_ACCOUNTS; ++i) {
            m_accounts[i] = BankSimple::Account::_nil();
    }
}

// Destructor.
BankSimple_BankImpl::~BankSimple_BankImpl() {
    delete [] m_accounts;
}
// Add a new account.
BankSimple::Account_ptr
    BankSimple_AccountImpl::create_account(const char*name,
                                    CORBA::Enviroment& ) {
    int i = 0;
    for ( ; i < MAX_ACCOUNTS&  !CORBA::is_nil(m_accounts[i]);
                               ++i)
    { }

    if (i < MAX_ACCOUNTS) {
        m_accounts[i]=
            new TIE_BankSimple_Account(BankSimple_AccountImpl)
                        (new BankSimple_AccountImpl(name, 0.0));
        cout << "create_account: Created with name:" << name
             << endl;
        return BankSimple::Account::_duplicate(m_accounts[i]);
    }
    else {
        cout << "create_account: failed, no space left!" << endl;
        return BankSimple::Account::_nil();
    }
}

// Find a named account
BankSimple::Account_ptr
    BankSimple_BankImpl::find_account (const char*  name,
                                    CORBA::Enviroment& ) {
    ... // Same as for BOAImpl approach.
    }
};
```

Developing a Server Program

To develop a server program, you must do the following:

- Create initial implementation objects for these interfaces.
- Make these objects available to clients, by allowing Orbix to receive and process incoming requests from clients.

This section describes how you can write a server `main()` function that creates a `Bank` implementation object and makes this object available to clients.

Writing a Server main() Function

This section shows the `main()` function of the banking application, using both the BOAImpl and the TIE approaches. In this example, the server `main()` function creates an implementation object of type `Bank`.

Using the BOAImpl Approach

The `main()` function for the server shows the creation of a `Bank` object. You could write this as follows:

```
// C++
// In file server.cxx.

#include <it_demo_nsw.h>
#include "banksimple_bankimpl.h"
#inlcude "banksimple_accountimpl.h"

int main(int argc, char*  argv[])) {
   try {
      // Process command line arguments and server options.
         ....
      // initialize the ORB and BOA (CORBA-defined).
      CORBA::ORB_var orb = CORBA::Orb_init (argc, argv, "Orbix");
      CORBA::BOA_var boa = orb->BOA_init(argc, argv, "Orbix_BOA");

      // Set diagnostics as specified on the command line.
      orb->setDiagnostics(serveropt.diagnostics());

      // Set server name (required to export object references).
      orb->setServerName(serveropt.server_name());

      // Indicate server should not quit when clients are connected.
      boa->setNoHangup(1);

      // For correct IOR generation, you must call
      // impl_is_ready() before objects are created.
      boa->impl_is_ready((char*)serveropt.server_name(), 0);

      // Create a new Bank implementation object.
      BankSimple::Bank_var my_bank = new BankSimple_BankImpl;
      // To simplify the use of the naming service, a Naming
      // Service Wrapper (NSW) is provided. Refer to
      // DemoLib/IT_DEMO_NSW* for further details.

      // Define a NSW object and define a name prefix to be
      // used for subsequent operations.
```

```
      IT_Demo_NSW ns_wrapper:
      ns_wrapper.setNamePrefix(serveropt.context());

      // Specify the name that the bank object is known as in
      // the naming service.
      const char* bank_name - "BankSimple.Bank";

      // Create missing contexts and overwrite existing entries in
      // the naming serivce.
      ns_wrapper.setBehaviourOptions
            (IT_Demo_NSW::createMissingContexts);
      ns_wrapper.setBehaviourOption
            (IT_Demo_NSW::overwriteExistingObject)

      // If unbind option is specified, unbind the server's
      // objects from the naming service and exit.
      if (server.opt.unbindns()) {
         cout << "Un-binding objects from the Naming Service"
               << endl;
         ns_wrapper.removeObject(bank_name);
         cout << "exiting..." << endl;
         return 0;
      }

      // If the bind option is specified on the command line,
      // register the server's object with the naming service.
      if (serveropt.bindns()) {
            cout << "Binding objects in the Naming Service"
                  << endl;
            ns_wrapper.registerObject(bank_name, my_bank);
      }

      // Server has completed initialization, and waits for
      // incoming requests.
      boa->impl_is_ready((char*)serveropt.server_name(),
                                    serveropt.timeout());

      // impl_is_ready() returns only when Orbix times-out an
      // idle server.
      cout << "server exiting" << endl;
   }
   catch (const CORBA::Exception& e) {
      cerr <<"Unexpected exception" << e << endl;
      return 1;
   }
   return 0;
}
```

This server initializes a `BankSimple::Bank_var` object reference with a new `BankSimple_BankImpl` object. The `BankSimple_BankImpl` object is created using a default constructor.

Before creating the `Bank` object, the server initializes the ORB and BOA. The server then calls `impl_is_ready()` to indicate that it has completed initialization and is ready to receive operation requests on its objects.

Using the TIE Approach

The implementation of the server `main()` function is similar in the TIE approach. The difference is that the server creates a `TIE` object in addition to a `BankSimple_BankImpl` object:

```
// C++
// In file server.cxx.

#include <BankSimple_BankImpl.h>
#inlcude <BankSimple_AccountImpl.h>
#include <IT_Demo_NSW.h>

int main(int argc, char*  argv[])) {
....
    // Create a new Bank implementation object.
    BankSimple::Bank_var my_bank =
        new TIE_BankSimple_Bank(BankSimple_BankImpl)
                            (new BankSimple_BankImpl);
    ....
    // Wait for imcoming requests.
    boa->impl_is_ready((char*)serveropt.server_name(),
                                    serveropt.timeout());
    cout << "server exiting" << endl;
...
}
```

This server `main()` initializes a `BankSimple::Bank_var` object reference with a new `TIE` object. The `BankSimple_BankImpl` object is created using a default constructor. The accounts managed by a bank are stored in a list with members of type `BankSimple::Account_ptr`. In this case, therefore, the linked list is composed of `TIE` objects.

Initializing the Server

A server is normally coded so that it initializes itself and creates an initial set of objects. It then calls `boa->impl_is_ready()` to indicate that it has completed its initialization and is ready to receive operation requests on its objects.

impl_is_ready()

The `impl_is_ready()` function normally does not return immediately. It blocks the server until an event occurs, handles the event, and re-blocks the server to wait for another event. A server must call `impl_is_ready()`; however, a client must not call this function.

The `impl_is_ready()` function is declared as follows:

```
// C++
// In class CORBA::BOA.
void impl_is_ready (
    const char* server_name = " ",
    CORBA::ULong timeOut =
        CORBA::ORB::DEFAULT_TIMEOUT,
CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
```

When a server is launched by Orbix, the server name is already known to Orbix and therefore does not need to be passed to `impl_is_ready()`. However, when a server is launched manually or externally to Orbix, the server name must be communicated to Orbix before any Orbix services are used.

The normal way to do this is as the first parameter to `impl_is_ready()`. To allow a server to be launched either automatically or manually, you should specify the `server_name` parameter.

By default, servers must be registered with Orbix, using the `putit` command. If an unknown server name is passed to `impl_is_ready()`, the call is rejected. However, you can configure the Orbix daemon (`orbixd`) to allow unregistered servers to be run manually. Refer to the ***Orbix C++ Edition Administrator's Guide*** for details.

If you do not want to specify the `server_name`, but want to specify a non-default `timeOut` or `Environment`, you should pass a zero length string (`""`) for the value of the `server_name` parameter.

The `impl_is_ready()` function returns only when a timeout occurs or an exception occurs while waiting for or processing an event. The `timeout` parameter indicates the number of milliseconds to wait between events.

A timeout occurs if Orbix has to wait longer than the specified timeout for the next event. A timeout of zero indicates that `impl_is_ready()` should time out and return immediately *without* checking for any pending event. A timeout does not cause `impl_is_ready()` to raise an exception.

Note: A server can time out either because it has no clients for the timeout duration, or because none of its clients uses it for that period.

The default timeout can be passed explicitly as `CORBA::ORB::DEFAULT_TIMEOUT`. You can specify an infinite timeout by passing `CORBA::ORB::INFINITE_TIMEOUT`.

Developing a Client Program

From the point of view of the client, the functionality provided by the `BankSimple` example is defined by the IDL interface definitions. A typical client program locates a remote object, gets a reference to the object and then invokes operations on the object.

These three concepts, object location, getting an object reference, and remote invocations, are important concepts in distributed systems:

- *Object location* involves searching for an object in the available nodes.
- *Getting a reference to an object*—establishes the facilities required to make remote invocations possible. In Orbix this involves the implicit creation of a proxy—a reference to the proxy is then returned to the client.
- *Remote invocations* in Orbix occur when normal C++ function calls are made on proxies.

These concepts are illustrated in this section. This client uses the Naming Service Wrappers to find and get a reference to a `Bank` object. Remote function invocations can then be made on the object. Alternatives to using the Naming Service are discussed later in this section.

You should refer to the chapter "Making Objects Available in Orbix" for a detailed discussion.

The main `BankSimple` client program performs initialization and then starts a simple interactive menu, enabling you to call IDL operations on a `Bank`. The client uses the following files to make remote invocations:

- `bankmenu.cxx`

  This calls operations on the `Bank` IDL interface.

- `accountmenu.cxx`

  This calls operations on the `Account` IDL interface.

The code for the client is as follows:

```
// C++
// In file client.cxx

#include <it_demo_streams.h>
#include <it_demo_clientoptions.h>
#include <it_demo_nsw.h>
#include "bankmenu.h"

// Connects to bank object, and runs a simple menu loop
// to call operations on bank or accounts.
int main (int argc,char* argv[]) {
      ...
      // ORB Setup - initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "Orbix");

      // Set the diagnostic level from the options
      orb->setDiagnostics(clientopt.diagnostics());

      // Naming Service Setup
      IT_Demo_NSW ns_wrapper;
      ns_wrapper.setNamePrefix(clientopt.context());

      // Get CORBA object.
      // Specify the object name in the naming service.
      const char* object_name = "BankSimple.Bank";

      // Get a reference to the required object from the NSW.
      CORBA::Object_var obj = ns_wrapper.resolveName(object_name);

      // Narrow the object reference.
      BankSimple::Bank_var bank = BankSimple::Bank::_narrow(obj);
      if (CORBA::is_nil(bank)) {
         cerr << "Object \"" << object_name
              << "\"in the naming service" << endl
              << "\tis not of the expected type."<< endl;

      return 1;
      }

      // Start client menu loop
         BankMenu main_menu(bank);
         main_menu.start();
      }
      ...
}
```

Alternatives to the Naming Service

Using the Naming Service is the CORBA-defined way to establish communications with a particular object. There are three other ways that a client can obtain a reference to an object it needs to communicate with:

- Using a return value or an `out` parameter to an IDL operation call.
- Using the Orbix-specific `_bind()` mechanism, although this is a deprecated feature.
- `Use resolve_initial_references()`

Using a Return Value or an Out Parameter

A client can also receive an object reference as a return value or as an `out` parameter to an IDL operation call. This results in the creation of a proxy in the client's address space. Operation `create_account()`, for example, returns a reference to an `Account` object, and a client that calls this operation can then make operation calls on the new object.

Using _bind()

The following code sample shows how a client could obtain a reference to a `Bank` object using the Orbix-specific `_bind()` operation:

```
try {
    CORBA::ORB_var orb =
        CORBA::ORB_init(argc, argv, "Orbix");
        const char* host = ...;
BankSimple::Bank_var bank =
        BankSimple::Bank::_bind
            ("BankMarker:IT_demo/BankSimple/Bank", host);
        ...
}
catch (const CORBA::Exception& e) {
        cout << "Unexpected exception" << e << endl;
}
```

The bind mechanism is implemented by the static member function `_bind()` of C++ class `BankSimple::Bank`. This function takes several parameters that uniquely identify the location of the specified implementation object in the system. Orbix chooses the named implementation object within the named server on the named host. The value returned by `BankSimple::Bank::_bind()` is a remote object reference. This is a pointer to a proxy object in the client address space. Refer to "Tabular Summary of Parameters to _bind()" on page 124 for further information.

Note: The anonymous bind feature, which allowed the Orbix runtime to choose any random implementation object in the named server, is no longer a feature of Orbix.

In general, a process must use the Naming Service, call `_bind()`, or call `resolve_initial_references()` at least once in order to communicate with objects outside of its address space. However, it should not overuse either these features. For many applications it is better for a server to make its objects known to its clients through IDL interfaces provided by other objects.

Where possible, you should use a combination of the Naming Service, `resolve_initial_references()`, and object references returned through IDL operations to make objects available to clients in your Orbix applications. The Orbix `_bind()` function is convenient, but is not defined in the CORBA standard.

Registering the Server

The last step in developing and installing your application is to register the server with the Implementation Repository.

The Orbix Implementation Repository records each server name and executable filename. Registering a server enables the Orbix daemon (`orbixd`) to launch a server that is not running when one of its objects is used. If the Orbix daemon is configured to allow unregistered servers, server registration is optional, and server that is not known to Orbix can then be run manually. Its call to `CORBA::BOA::impl_is_ready()` must specify its server name. In addition, the server must call to `impl_is_ready()` before any other calls to Orbix.

Every node in a network that runs servers must have access to an Implementation Repository. Repositories can be shared using a network file system.

You can register a server using either the Server Manager GUI tool or run the Orbix `putit` command on the server host as follows:

`putit` *server name server path* [*server command line arguments*]

For example, on UNIX, the `Bank` server might be registered as follows:

```
% putit Bank /usr/users/joe/banker
```

The executable file `/usr/users/joe/banker` is then registered as the implementation code of the server named `Bank` at the current host. The `putit` command does not run the executable; you can execute this explicitly from the shell. Alternatively, it is launched automatically by Orbix in response to an incoming operation invocation.

For more information on registration and activation of servers, refer to the ***Orbix C++ Edition Administrator's Guide***.

Execution Trace for the Example Application

This section considers the events that occur when the `Bank` server and client are run. The TIE approach is used to show the initial trace, and then the BOAImpl approach is discussed.

First a server with the name "`Bank`" is registered in the Implementation Repository. Then, when an invocation arrives from a client, Orbix launches the server using the specified executable file; for example, `/usr/users/joe/banker`.

The server process creates a new `TIE` (of class `TIE_Bank(BankImpl)`) for an object of class `BankSimple_BankImpl`, and waits on `CORBA::BOA::impl_is_ready()`:

```
// C++
// In file server.cxx.

#include "banksimple_bankimpl.h"
```

```
#inlcude "banksimple_accountimpl.h"
#include "IT_Demo_NSW.h"

int main(int argc, char*  argv[])) {
    ....
    // Create a new Bank implementation object.
    BankSimple::Bank_var my_bank =
            new TIE_BankSimple_Bank(BankSimple_BankImpl)
                    (new BankSimple_BankImpl);
    ....
    // Wait for imcoming requests.
    boa->impl_is_ready((char*)serveropt.server_name(),
                               serveropt.timeout());
    cout << "server exiting" << endl;
...
}
```

The state of the server, at the time of the `impl_is_ready()` call, is shown in Figure 11. The server is now waiting for incoming requests. If `impl_is_ready()` times out, the server terminates.

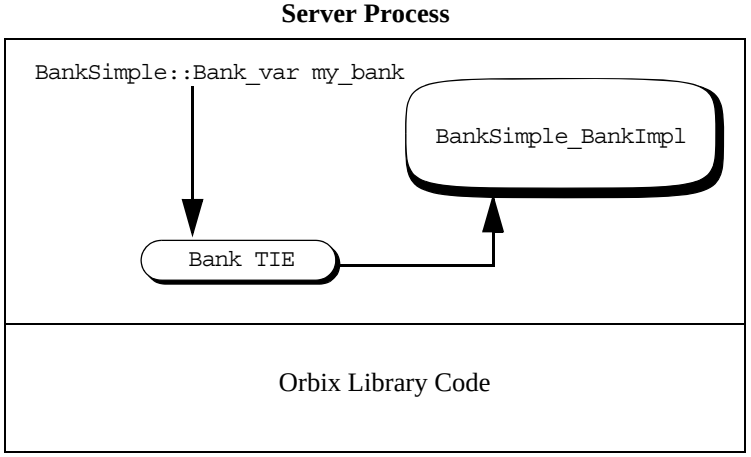


Figure 11: *State of the Server at Launch*

Now consider the client: it first binds to a `Bank` object, using the Naming Service; for example:

```
// C++
// In file client.cxx
...
int main (int argc,char* argv[]) {
      ...
      // Naming Service Setup
      IT_Demo_NSW ns_wrapper;
      ns_wrapper.setNamePrefix(clientopt.context());

      // Get CORBA object.
      // Specify the object name in the naming service.
      const char* object_name = "BankSimple.Bank";

      // Get a reference to the Bank object from the NSW.
      CORBA::Object_var obj = ns_wrapper.resolveName(object_name);

      // Narrow the object reference.
      BankSimple::Bank_var bank = BankSimple::Bank::_narrow(obj);
```

The result is an automatically generated proxy object in the client, which acts as a stand-in for the remote `BankSimple_BankImpl` object in the server. The object reference `bank` within the client is now a remote object reference as shown in Figure 12.

The client programmer is not aware of the `TIE` object. Nevertheless, all remote operation invocations on our `BankSimple_BankImpl` object go via the TIE.

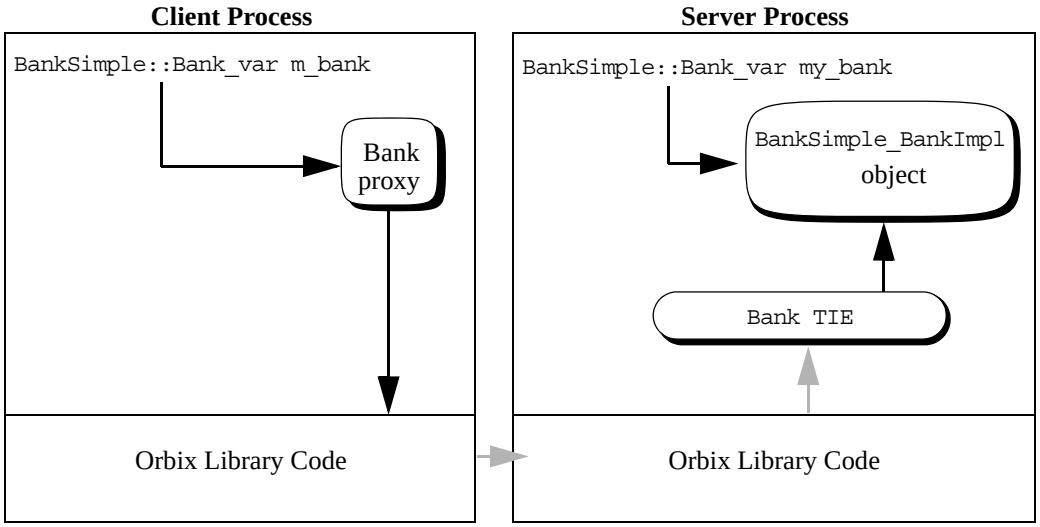


Figure 12: *Client Binds to Bank Object (TIE Approach)*

The client program proceeds by using the client menu to ask the bank to open a new account:

```
// C++
// In file bankmenu.cxx
...
BankSimple::Account_var account
            = m_bank->create_account(name);
```

When the `m_bank->create_account()` call is made, the function `BankSimple::BankImpl::create_account()` is called (via the `TIE`) within the bank server. This generates a new `BankSimple_AccountImpl` object and associated `TIE` object. The `TIE` object is linked into the `BankSimple_BankImpl` object's list of `Account`s.

Finally, `create_account()` returns the `Account` reference back to the client. At the client side, a new proxy is created for the `Account` object, and this is referenced by the `m_account` variable (Figure 13).

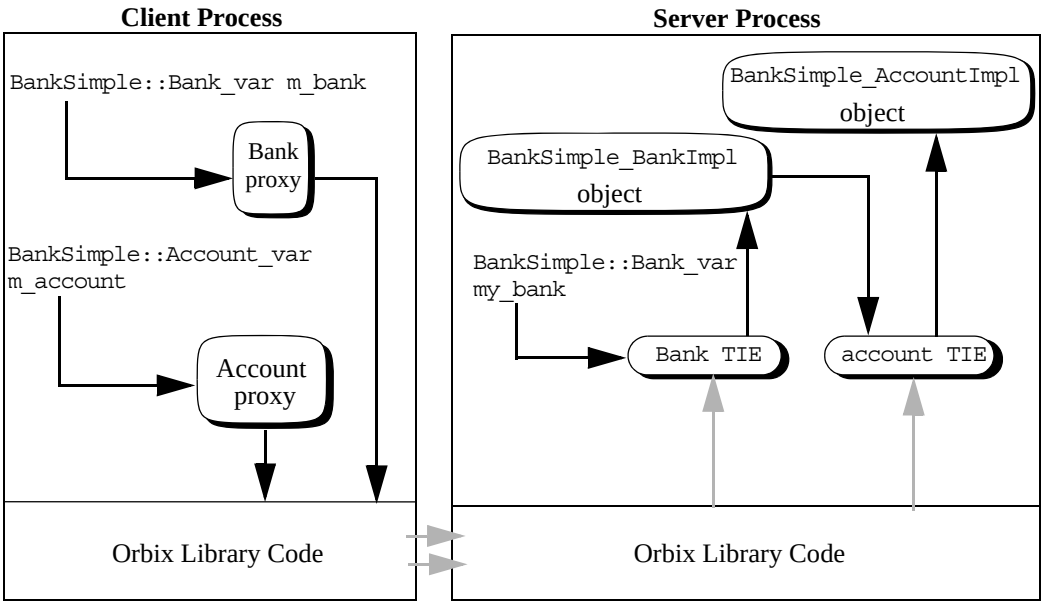


Figure 13: *Client Accesses Account Object (TIE Approach)*

Using the BOAImpl approach, the final diagram is as shown in Figure 14.

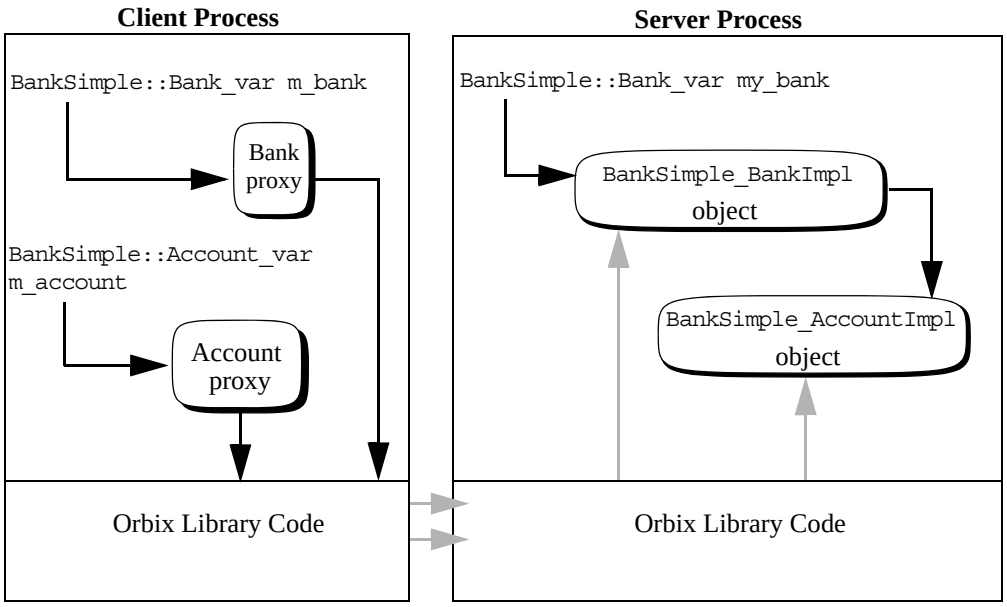


Figure 14: *Client Accesses Account Object (BOAImpl Approach)*

Comparing the TIE and BOAImpl Approaches

This section highlights further ways you can use the TIE and BOAImpl approaches to provide implementation classes, and compares both approaches.

Wrapping Existing Code

Orbix provides a mechanism to achieve application integration for both new and existing applications. An application can allow other code to use its services by providing a number of IDL interfaces and making these available to the overall system. This allows you to write new applications by combining the facilities of existing applications. Because the components of the system are objects whose internals are hidden from their clients, these objects can provide the basis for integrating with legacy systems. Over time, legacy systems can be replaced with newer systems which nevertheless provide the same CORBA interfaces. One aspect of this wrapping of existing code is the ability to implement an IDL interface using existing C++ classes.

Using the TIE Approach

The TIE approach is clear on whether or not it supports wrapping existing code. If the existing C++ class has exactly the correct member functions (each function has exactly the correct name and correct parameter types), you must call the appropriate `DEF_TIE` macro. In addition, you must also add the `CORBA::Environment` parameter to the member functions because existing code would not have this parameter. The existing code may have other functions that do not correspond to IDL attributes or operations in the IDL interface in question. However, if the existing C++ code does not have exactly the correct member functions, the TIE approach cannot be used.

Using the BOAImpl Approach

To use the BOAImpl approach for existing code, you must use C++ multiple inheritance to specify the relationship between the IDL C++ class and the previously written implementation class. Instances of the derived class are then valid implementations of the IDL interface. Figure 15 on page 110 shows how you can use the BOAImpl approach to allow a pre-existing class to implement an IDL interface. The programmer has already implemented a class, `BankAccount`, which provides an implementation of each of the functions of the IDL interface. To indicate that this class implements the IDL interface, a class `BankSimple_AccountImpl` has been defined that inherits from both the BOAImpl-class and the class, `BankAccount`, which provides the functions. Class `BankSimple_AccountImpl` is the class which is said to implement the IDL interface.

This is more difficult to code than the corresponding code for the TIE approach, where a call to the appropriate `DEF_TIE` macro may be all that is required. However, the BOAImpl approach is significantly more flexible in its use of existing code. In particular, the code for class `BankSimple_AccountImpl` can manipulate any call that it receives before passing it on to the code for class

BankAccount. This manipulation can compensate for differences in function names and parameters, and differences in function semantics.

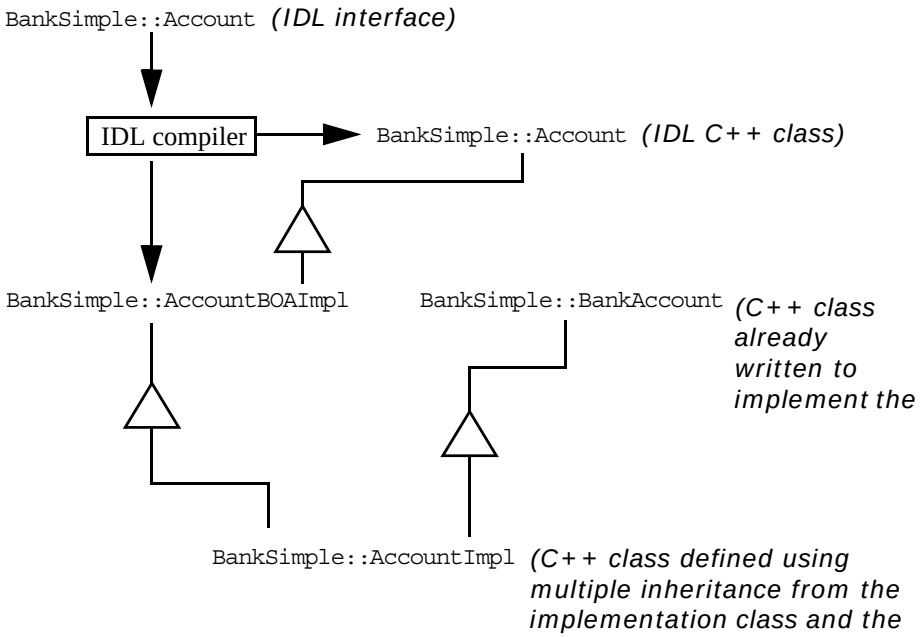


Figure 15: *BOAImpl Approach to Allow an Existing Class to Implement an IDL Interface*

Providing Different Implementations of the Same Interface

Both the BOAImpl and TIE approaches allow you to provide a number of different implementations of the same IDL interface—to provide more than one implementation class for a given IDL interface. This is an important feature, especially in a large heterogeneous distributed system. An object can then be created as an instance of any one of the implementation classes. Client programmers need not be aware of which class has been chosen.

Providing Different Interfaces to the Same Implementation

You can have a C++ implementation class that implements more than one IDL interface. This class must declare all of the operations defined in all of the interfaces it implements. In the TIE approach, this common class is tied to different IDL interfaces using multiple DEF_TIE macro calls.

In the BOAImpl approach, this usually requires an IDL interface that derives from all of the IDL interfaces in question.

Comparison of the BOAImpl and TIE Approaches

This section briefly compares the BOAImpl and TIE approaches to implementing IDL interfaces in C++. In real terms, these do not differ greatly in their power, and it is frequently a matter of personal taste which one is preferred. The TIE and BOAImpl approaches can be freely mixed within the same server.

The TIE approach has a small advantage in that it allows an advanced feature known as “per-object” filtering to be used. This allows you to specify additional code that is to be executed when an invocation is made on a particular object; from the same or a different address space. Both the BOAImpl and the TIE approach enable you to specify additional code to be executed when an attribute or operation invocation is made across an address space boundary; from a client/server to a client/server on the same or a different host.

Refer to “Filtering Operation Calls” for more information on using filters with Orbix.

Making Objects Available in Orbix

A central requirement in a distributed object system is for clients to be able to locate the objects they wish to use. This chapter describes how you can make objects available in servers and locate those objects in clients.

Before using a CORBA object, a client must establish contact with it. To do this, the client must get an *object reference* for the required object. An object reference is a unique value that tells an ORB where an object is and how to communicate with it.

A problem for every CORBA application is how servers can make object references available to clients and how clients can retrieve these references to establish contact with objects. This chapter describes three solutions to this problem:

- Using the CORBA Naming Service.
- Using a basic protocol to transfer object references between servers and clients.
- Using the Orbix `_bind()` function.

These solutions are presented after a brief introduction to how object references work in CORBA.

Identifying CORBA Objects

Every CORBA object is identified by an object reference—a unique value that includes all the information an ORB requires to locate and communicate with the object. When a client gets a reference to an object, the ORB creates a proxy in the client's address space. When the client calls an operation on the proxy, the ORB transmits the request to the target object.

Orbix supports two protocols for communications between clients and servers:

- The CORBA standard Internet Inter-ORB Protocol (IIOP), which is the default protocol.
- The Orbix protocol.

Each of these communication protocols has its own object reference format.

Interoperable Object References

An object that is accessible using IIOP is identified by a CORBA interoperable object reference (IOR). An IOR encodes various pieces of information about an object, including:

- The Internet address of the object's host.
- A port number used to communicate with the object.
- An object reference, in the format of the native ORB protocol.

For example, an IOR for an Orbix object includes the object's full Orbix object reference.

IORs are managed internally by the ORB. It is not necessary for you to know the detailed structure of an IOR. However, you may wish to publish IORs in their string format, as described in "Transferring Object Reference Strings" on page 121.

Orbix Object References

Every object created in an Orbix application has an associated Orbix object reference. This object reference includes the following information:

- An object name that is unique within the object's server. This name is known as the object's *marker*.
- The object's server name.
- The server's host name.

For example, the object reference for a bank account would include the object's marker name, the name of the server that manages the account, and the name of the server's host. The bank server could, if necessary, create and name different bank objects with different names, all managed by the same server.

In more detail, an Orbix object reference is fully specified by the following fields:

- Object marker.
- Server name.
- Server host name.
- IDL interface type of the object.
- Interface Repository (IFR) server in which the interface definition is stored.
- IFR server host.

All Orbix objects inherit the C++ class `CORBA::Object`. This interface supplies several methods common to all object references, including `_object_to_string()`. Given an Orbix object reference, this function produces a string that has the following format:

```
:\host:server_name:marker:IFR_host:IFR_server:IDL_interface
```

Class CORBA::Object also provides access to the individual fields of an object reference string through this set of accessor functions:

```
// C++
// in class CORBA::Object.
const char* _host(CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv()) const;
const char* _implementation(CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv()) const;
const char* _marker(CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv()) const;
const char* _interfaceHost(CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv()) const;
const char* _interfaceImplementation(CORBA::Environment&
              IT_env = CORBA::IT_chooseDefaultEnv()) const;
const char* _interfaceMarker(CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv()) const;
```

In general, the IFR host name (interfaceHost) and IFR server (interfaceImplementation) fields are set to default values. Orbix automatically assigns the server host, server name, and IDL interface fields on object creation, and it is not generally necessary to update these values. Orbix also assigns a marker value to each object, but you can choose alternative marker values in order to name Orbix objects explicitly.

Assigning Markers to Orbix Objects

There are two ways to specify a marker for an object: by setting the marker when creating the object or by calling the modifier function CORBA::Object::_marker(). If you do not specify a marker for an object, Orbix automatically sets the marker value.

The method of assigning a marker when creating an object depends on the approach used to implement the IDL interface:

- If you use the TIE approach, pass a marker name to the second parameter (of type const char*) of a TIE constructor. For example:

  ```
  // C++
  BankSimple::Bank_var bVar = new TIE_Bank
  (BankSimple_BankImpl)
      (new BankSimple_BankImpl, "College_Green");
  ```

- If you use the BOAImpl approach, pass a marker name to the first parameter (of type const char*) of a BOAImpl constructor. For example:

  ```
  // C++
  class BankSimple_BankImpl : public virtual BankBOAImpl
  {
  public:
      BankSimple_BankImpl (const char* marker);
      ...
  };

  BankSimple_BankImpl::BankSimple_BankImpl
      (const char* marker) : BankBOAImpl (marker) {
  }
  ```

Choosing Marker Names

A marker name chosen by Orbix consists of a string composed entirely of numeric characters. You can ensure that your markers are different from those chosen by Orbix by not using strings that consist entirely of numeric characters. Marker names cannot contain the character ':' or the null character.

An object's interface name together with its marker name must be unique within a server. If a chosen marker is already in use when an object is named, Orbix silently assigns a different marker to the object. The object with the original marker will be unaffected. There are two ways to test for this, depending on how a marker is assigned to an object:

- If `_marker(const char*)` is used, you can test for a `false` return value; this indicates a name clash.
- If the marker is assigned when creating a `TIE` or when calling a BOAImpl class constructor, you can test for a name clash by calling the parameterless accessor function `_marker()` on the new object and comparing the marker with the one you tried to assign. This approach is necessary because the return value from the `new` operator is non-zero if there is a name clash.

Using the CORBA Naming Service

The Naming Service allows you to associate abstract names with CORBA objects and allows clients to find those objects by looking up the corresponding names. A server that holds a CORBA object *binds* a name to the object by contacting the Naming Service. To obtain a reference to the object, a client contacts the Naming Service and *resolves* the specified name.

Most CORBA applications make some use of the Naming Service. Each copy of Orbix includes a copy of OrbixNames, the Orbix implementation of the Naming Service, so you can use the Naming Service in any of your applications.

This section provides an overview of the Naming Service and briefly describes how you use the standard interface to the Naming Service. Before using this service, see the ***OrbixNames Programmer's and Administrator's Guide*** for more detailed information.

The Interface to the Naming Service

The programming interface to the Naming Service is defined in IDL. A standard set of IDL interfaces allow you to access all the Naming Service features. OrbixNames, for example, is a normal Orbix server that contains objects implementing these interfaces.

The Naming Service interfaces are defined in the IDL module `CosNaming`:

```
// IDL
module CosNaming {
    // Naming Service IDL definitions.
    ...
};
```

Format of Names in the Naming Service

The Naming Service maintains a database of names and the objects associated with them. In the Naming Service, names can be associated with two types of objects: a *naming context* or an application object. A naming context is an object in the Naming Service within which you can resolve the names of other objects.

The full name of an object, including all the associated naming contexts, is known as a *compound name*. The first component of a compound name gives the name of a naming context, in which the second component is accessed. This process continues until the last component of the compound name has been reached.

A name component is defined as an IDL structure, of type `CosNaming::NameComponent`, that holds two strings:

```
// IDL
// In module CosNaming.
typedef string  Istring;

struct NameComponent {
    Istring id;
    Istring kind;
};
```

A name is a sequence of these structures:

```
typedef sequence<NameComponent> Name;
```

The `id` member of a `NameComponent` is a simple identifier for the object; the `kind` member is a secondary way to differentiate objects and is intended to be used by the application layer. Both the `id` and `kind` members of a `NameComponent` are used to differentiate names.

Making Initial Contact with the Naming Service

The IDL interface `NamingContext`, defined in module `CosNaming`, provides access to most features of the Naming Service. The first step in using the Naming Service is to get a reference to an object of this type.

Each Naming Service contains a special `CosNaming::NamingContext` object, called the root naming context, that acts as an entry point to the service. The root naming context allows you to create new naming contexts, bind names to objects, resolve object names, and browse existing names.

To get a reference to the root naming context, pass the string `NameService` to the following C++ function call on the ORB (the `CORBA::Orbix` object):

```
// C++
// In class CORBA::ORB.
Object_ptr resolve_initial_references(
                                const char* identifier)
```

You can then narrow the returned object reference using the function `CosNaming::NamingContext::_narrow()`. Some configuration is required for this to work, as described in the ***OrbixNames Programmer's and Administrator's Guide***.

Associating Names with Objects

Once you have a reference to the root naming context, you can begin to associate names with objects. The operation `CosNaming::NamingContext::bind()` enables you to bind a name to an object in your application. This operation is defined as:

```
void bind (in Name n, in Object o)
    raises (NotFound, CannotProceed,
            InvalidName, AlreadyBound);
```

To use this operation, you first create a `CosNaming::Name` structure containing the name you want to bind to your object. You then pass this structure and the corresponding object reference as parameters to `bind()`.

Using Names to Find Objects

Given an abstract name for an object, you can retrieve a reference to the object by calling `CosNaming::NamingContext::resolve()`. This operation is defined as:

```
Object resolve (in Name n)
    raises (NotFound, CannotProceed, InvalidName);
```

When you call `resolve()`, the Naming Service retrieves the object reference associated with the specified `CosNaming::Name` value and returns it to your application.

Associating a Compound Name with an Object

If you want to use compound names for your objects, you must first create naming contexts. For example, consider the compound name shown in Figure 16.

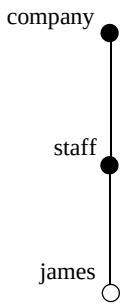


Figure 16: *An Example Compound Name*

To create this compound name:

1. Create a naming context and bind a name with identifier `company` (and no kind value) to it.
2. Create another naming context, in the scope of the `company` context, and bind the name `staff` to it.
3. Bind the name `james` to your application object in the scope of the `staff` context.

The operation `CosNaming::NamingContext::bind_new_context()` enables you to create naming contexts:

```
NamingContext bind_new_context (in Name n)
    raises (NotFound, CannotProceed,
            InvalidName, AlreadyBound);
```

To create a new naming context and bind a name to it, create a `CosNaming::Name` structure for the context name and pass it to `bind_new_context()`. If the call is successful, the operation returns a reference to your newly-created naming context.

Using the Naming Service in Orbix Example Applications

The code examples presented in other chapters of this guide use the Naming Service. To simplify the code, these examples access the Naming Service through a set of wrapper functions. These functions then communicate with the Naming Service using the standard IDL interfaces.

The wrapper functions are defined in the class `IT_Demo_nsw`. You can find the declaration of this class in the file `IT_Demo_nsw.h` in the `demos` directory of your Orbix installation.

The functions are:

`registerObject()`	This function takes two parameters: a string format name and an object reference. It converts the string to a `CosNaming::Name` structure and then binds this name to the specified object. If you specify a compound name, the naming contexts must already exist.
`resolveName()`	This function takes a string format name, converts it to an equivalent `CosNaming::Name` structure and attempts to resolve this name in the Naming Service. It returns a reference to the object associated with the name.
`removeObject()`	This function removes the association between a name and an object in the Naming Service.
`setNamePrefix()`	Function `setNamePrefix()` allows you to shorten the name parameters passed to `registerObject()`, `resolveName()`, and `removeObject()`. The specified string prefix is added to the name parameter passed to each subsequent call to these operations.
`clearNamePrefix()`	This function clears the string prefix added to each name parameter by `setNamePrefix()`.

The functions `registerObject()`, `resolveName()`, and `removeObject()` take object names as parameters. To avoid the creation of `CosNaming::Name` structures directly in sample code, these functions take name parameters in the string format described in the ***OrbixNames Programmer's and Administrator's Guide***. This is a convenient way to format names and is also used by the OrbixNames command-line utilities.

Transferring Object References

There are two ways to pass object references directly between a server and a client:

- Using IDL operation parameters.
- Using the string format of object references.

This section examines each in turn.

Passing Object References as Operation Parameters

IDL operations can return object references as parameters or return values; for example:

```
// IDL
interface Account;

interface Bank {
    // Create a new account.
    Account create_account (in string name);
    // Find an existing account.
    Account find_account   (in string name);
};

interface Account {
    ...
};
```

An object that implements interface `Bank` acts as a *factory* for the creation of `Account` objects. The operations `create_account()` and `find_account()` pass object references to clients as return values.

Of course, to receive an object reference from an operation, a client must first be able to call the operation. This implies that the client already has a reference to some object in the server. A common strategy in CORBA applications is to make one or more server objects available through the Naming Service, or some similar mechanism, and let these act as entry points to other server objects.

In fact, the Naming Service itself uses this strategy. A standard function call, `resolve_initial_references()` returns the root naming context and this object acts as an entry point to all other objects in the service.

Transferring Object Reference Strings

One way to pass an object reference from a server to a client without establishing communications first, is to use object reference strings. As described in "Identifying CORBA Objects" on page 113, you can get the string form of an object reference by calling the function `CORBA::ORB::object_to_string()`. Given the string form of an object reference, an Orbix client can create a proxy for that object by passing the string to the function `CORBA::ORB::string_to_object()`.

One simple protocol for passing an object reference from a server to a client is as follows:

1. The server calls `CORBA::ORB::object_to_string()` to get the string format of an object reference.
2. The server writes this string to a location, for example a file, accessible by both client and server.
3. The client reads the object reference string.
4. The client calls `CORBA::ORB::string_to_object()` to create a proxy.

For example, given an object reference string that identifies a `BankSimple::Bank` object, a client can create a proxy as follows:

```
// C++
// Assign object reference string to bankString.
String_var bankString = ... ;

// Create proxy.
BankSimple::Bank_var bVar =
    CORBA::Orbix.string_to_object(bankString);
```

The function `string_to_object()` is overloaded to allow the individual fields of a stringified object reference to be specified. See the entries for `CORBA::ORB::string_to_object()` in the ***Orbix Programmer's Reference C++ Edition*** for more details.

Binding to Orbix Objects

The Orbix `_bind()` function finds a particular object using specific information about the object's location in a distributed system. For example, when calling `_bind()` you can specify the exact object you require in a particular server on a particular host.

Overview of the _bind() Function

The `_bind()` function is a static member function automatically generated by the IDL compiler for each IDL C++ class. For interface `BankSimple::Bank`, the full declaration of `_bind()` is:

```
// C++
static BankSimple::Bank_ptr _bind
    (const char* markerServer, const char* host, const
       CORBA::Context&, CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
```

Combining default parameters and overloading, `_bind()` can take the following sets of parameters:

- `markerServer`, `host`, `Context`, `Environment` (the last is defaulted).
- `markerServer`, `host`, `Environment` (the last is defaulted).
- `markerServer`, `host`.
- A full object reference as returned by the function `CORBA::ORB::object_to_string()`.

The `_bind()` function that supported polymorphic binds has been deprecated and is now removed from Orbix.

The markerServer Parameter to _bind()

The `markerServer` parameter denotes a specific server name and object within that server. It can be a string of the form:

marker`:`*server_name*

The marker identifies a specific object within the server. The server name is the name with which the server is registered in the Implementation Repository.

The server name must be supplied in all cases except if CORBA::collocated is true, indicating that the _bind is to be a local object not a remote object. In the collocated case, the name of the server in which the bind is being performed, can be null. If the server name is supplied, it should be the name of the calling process. (See "Developing Collocated Clients and Servers".)

The marker must be supplied in all cases. Anonymous bind (not supplying the marker) is deprecated in Orbix 3.3. Clients built with previous versions of Orbix can still use anonymous bind even with Orbix 3.3 servers.

If the string does not contain a '`:`' character, the string is understood to be a marker with no explicit server name. Since colon ('`:`') is used as a separator, it is illegal for a marker or a server name to include a '`:`' character.

The `_bind()` function first looks for the object in the caller's address space if the server name and host name are that of the caller.

Examples of the markerServer parameter that could be used in a call to BankSimple::Bank::_bind() are:

"College_Green:AIB"	The College_Green object at the AIB server.
"College_Green"	Local process; only legal if object collocation is turned on.
"College_Green:"	Local process; only legal if object collocation is turned on.
"College_Green:myBank"	The College_Green object at the myBank server.

Finally, if the markerServer parameter has at least *two* ':' characters within it, it is not treated as a marker:server-name pair, but it is assumed to be the string form of a full object reference. A full object reference string is returned by the function CORBA::ORB::object_to_string(), to which you can pass any Orbix object as a parameter. A call to _bind() with a full object reference string is similar to a call to the function CORBA::ORB::string_to_object().

The host Parameter to _bind()

The host parameter to _bind() specifies the Internet host name or the Internet address of a node on which to find the server. An Internet address is assumed to be a string of the form xxx.xxx.xxx.xxx, where x is a decimal digit. In the collocated case, the host name can be null. (See "Developing Collocated Clients and Servers".)

Example calls to _bind()

This section shows a selection of sample calls to _bind().

```
// C++

// Bind to the "College_Green" object at the "AIB"
// server at node beta, in the internet domain
// "mc.ie".  That object should implement the Bank
// IDL interface.
    BankSimple::Bank_var bVar =
    BankSimple::Bank::_bind ("College_Green:AIB",
         "beta.mc.ie");

// Bind to the College_Green" object in this
// local process.
    CORBA::collocated(true);
    BankSimple::Bank_var bVar =
    BankSimple::Bank::_bind("College_Green");
```

Tabular Summary of Parameters to _bind()

The following table summarizes the rules for a general-form call to `_bind()` :

```
// C++
T1_var tVar;
tVar = T2::_bind("M:S", "H");
```

`T1`	`T1` must be the same or a base type of `T2`.
`T2`	`T2` is an IDL interface name. It is not the name of a server, unless a server is explicitly registered with the same name as an interface. The object that is found must implement interface `T2`.
`M`	`M` is a marker name—the name of an object within the specified server.
`S`	`S` is a server name—a name used previously to register a server in the Implementation Repository.
`H`	`H` is an Internet host name or (if the string is in the format `xxx.xxx.xxx.xxx`, where `x` is a decimal digit) an Internet address. `H` can only be blank when collocated binds are being performed.

Binding and Exceptions

`_bind()` raises a BAD_PARAM System Exception if the rules regarding supplying a marker, server name and host name are not followed.

By default, `_bind()` raises an exception if the desired object is unknown to Orbix. Doing so requires Orbix to *ping* that desired object in order to check its availability. The ping causes the target server process to be activated if necessary, and it confirms that this server recognizes the target object.

(The ping operation is defined by Orbix and it has no effect on the target object. For the Orbix protocol, it is defined by Orbix; for IIOP, it is a LocateRequest.)

You can improve efficiency by reducing the number of remote invocations. To do this, call the function `pingDuringBind()` to disable the ping operation:

```
// C++
// In class CORBA::ORB.
CORBA::Orbix.pingDuringBind(0); // 0 for false.
```

The previous setting is returned. The ***Orbix Programmer's Reference C++ Edition*** provides more details about this function.

When ping is disabled, binding to an unavailable object does not raise an exception at that time unless the parameter rules are violated. Instead, an exception is raised when the proxy object is first used. It is not until the proxy object is used the first time that the marker and interface of the remote object is verified in the specified server.

A program should always check for exceptions when calling `_bind()`, whether or not pinging is enabled.

Exception Handling in Orbix

The implementation of an IDL operation or attribute can throw an exception to indicate that a processing error has occurred. This chapter describes Orbix exception handling in detail, using an example named BankExceptions. This example builds on the concepts illustrated in the BankSimple example in "Getting Started With Orbix" and "Using and Implementing IDL Interfaces".

There are two types of exceptions that an IDL operation can throw:

- *User-defined exceptions*.

  These exceptions are defined explicitly in your IDL definitions, and can only be thrown by operations.

- *System exceptions*.

  These are pre-defined exceptions that all operations and attributes can throw.

This chapter describes user-defined exceptions and system exceptions in turn and shows how to throw and catch these exceptions.

The examples in this chapter, and throughout this guide, assume that your C++ compiler supports C++ exception handling. Orbix no longer supports compilers without native C++ exception handling. The macro support (TRY, CATCH) for non-native C++ exception is eliminated.

An Example of Raising and Handling Exceptions

This chapter extends the BankSimple example so that the `create_account()` operation can raise an exception if the bank cannot create an `Account` object. The source code for the example described in this chapter is available in the `demos\bankexceptions` directory of your Orbix installation.

The exception `CannotCreate` is defined within the `Bank` interface. This defines a string member that indicates the reason that the `Bank` rejected the request:

```
        // IDL
        // In file bankexceptions.idl

        module BankExceptions {
           typedef float CashAmount;
           interface Account;

           interface Bank {
1     // User-defined exceptions.
              exception CannotCreate { string reason; };
              exception NoSuchAccount { string name; };

              Account create_account (in string name)
2             raises (CannotCreate);
              Account find_account (in string name)
                raises (NoSuchAccount);
```

```
    };

    interface Account {
     // User-defined exception.
3   exception InsufficientFunds { };

      readonly attribute string name;
     readonly attribute CashAmount balance;

     void deposit (in CashAmount amount);
     void withdraw (in CashAmount amount)
       raises (InsufficientFunds);
    };
};
```

This IDL is explained as follows:

1. `CannotCreate` and `NoSuchAccount` are user-defined exceptions defined for the `Bank` IDL interface.
2. Operation `BankExceptions::Bank::create_account()` can raise the `BankExceptions::Bank::CannotCreate` exception. It can only raise listed user-defined exceptions. It can raise any system-defined exception.
3. An exception does not need to have any data members.

Note: Read or write access to any IDL attribute can also raise any system-defined exception.

The Generated C++ Code for User-Defined Exceptions

The IDL compiler generates the following C++ definition for the `CannotCreate` user-defined exception from the `Bank` IDL definition:

```
// C++
// In file bankexceptions.hh

class CannotCreate : public CORBA::UserException {
    ...
public:
    static const char* _ex;
    CORBA::String_mgr reason;

    virtual CORBA::Exception* copy() const;
    CannotCreate (const char*  _reason);
    virtual void _throwit ();
    static CannotCreate* CALL_SPEC _narrow
                          (CORBA::Exception* e);
    CannotCreate(const CannotCreate&);
    CannotCreate();
    virtual ~CannotCreate();
    CannotCreate& operator= (const CannotCreate&);
};
```

Exception `BankExceptions::Bank::CannotCreate` is translated into a C++ class with the same name. Each C++ class corresponding to an IDL exception has a constructor that takes a parameter for each member of the exception. Because the `CannotCreate` exception has one member (`reason`, of type `string`), class `BankExceptions::Bank::CannotCreate` has a constructor that allows that single member to be initialized.

Handling Exceptions in a Client

A client (or server) calling an operation that can raise a user exception should handle that exception using an appropriate C++ `catch` clause. All clients should also catch system exceptions. The BankSimple client calls the `create_account()` operation as follows:

```
// C++
// In file bankmenu.cxx

BankMenu::BankMenu(BankExceptions::Bank_ptr bank)
    throw() : m_bank (BankExceptions::Bank::_duplicate(bank))
{ }
    ...

// do_create -- calls create_account and runs an account menu
void BankMenu::do_create() throw(CORBA::SystemException) {

    cout << "Enter account name: " << flush;
    CORBA::String_var name = IT_Demo_Menu::get_string();

    try {
        BankExceptions::Account_var account =
                            m_bank->create_account(name);


        // Start a sub-menu with the returned account reference
        AccountMenu sub_menu(account);
        sub_menu.start();
    }
catch (const BankExceptions::Bank::CannotCreate& cant_create)
    {
        cout << "Cannot create an account, reason:"
                << cant_create.reason << endl;
    }
}
```

The handler for the `BankExceptions::Bank::CannotCreate` exception outputs an error message and exits the program. The parameter to the `catch` clause is passed by reference.

The `operator<<()` function defined on class `SystemException` outputs a text description of the individual system exception raised. This text is read from a standard file, and can be modified for individual installations. Refer to the ***Orbix Programmer's Reference C++ Edition*** for more details.

If the handler for the `BankExceptions::Bank::CannotCreate` exception does not exit the program, you must be careful about the value of the variable `m_bank`. In particular, if an exception occurs in `create_account()`, the return value of this operation call would be undefined, and hence `m_bank` would be undefined (as specified by the C++ exception model. The C++ exception model also specifies that the values of out and inout parameters are undefined if an operation raises an exception).

A simple way to address this is shown in the following code segment, where the nil object reference value is assigned to `m_bank`, and this value is tested for before `m_bank` is used after the `catch` clauses:

```
// C++
// In file bankmenu.cxx

    // Ensure the bank reference is valid.
    if (CORBA::is_nil(m_bank)) {
        cout << "Cannot proceed - bank reference is nil";
    }
    else {
        // Loop printing the menu and executing selections
        ...
        try {
            ...
        }
        catch (const CORBA::SystemException& e) {
            cout << "Unexpected exception:" << e << endl;
        }
    }
}
```

The `is_nil()` function determines whether the object reference is nil. A nil object reference is one that does not refer to any valid Orbix object. The `is_nil()` function, defined in the CORBA namespace, is the only CORBA-compliant way of ascertaining whether an object reference is nil.

Handling Specific System Exceptions

A client may also provide a handler for a specific system exception. For example, to explicitly handle a `CORBA::COMM_FAILURE` exception that might be raised from a call to `create_account()`, the client could write code as follows:

```
// C++

#define EXCEPTIONS
#include "BankMenu.h"
#include <IT_Demo_Menu.h>

BankMenu::BankMenu(BankExceptions::Bank_ptr bank)
   throw() : m_bank (BankExceptions::Bank::_duplicate(bank)) { }
...

void BankMenu::do_create() throw(CORBA::SystemException) {

   cout << "Enter account name:" << flush;
   CORBA::String_var name = IT_Demo_Menu::get_string();

   try {
      BankExceptions::Account_var account =
                    m_bank->create_account(name);

      // Start a sub-menu with the returned account reference
      AccountMenu sub_menu(account);
      sub_menu.start();
   }
```

```
   catch (const CORBA::COMM_FAILURE& se) {
      cout << "Communications failure exception"
           << endl <<& se << endl;
   }
   catch (const CORBA::SystemException& se) {
      cout << "Unexpected system exception"
           << endl <<& se << endl;
   }
   catch(const BankExceptions::Bank::CannotCreate&  cant_create) {
      cout << "Cannot create an account, reason:"
           << cant_create.reason << endl;
   }
}
```

The handler for a specific system exception must appear before the handler for `CORBA::SystemException`. In C++, `catch` clauses are attempted in the order specified, and the *first* matching handler is called. Because of implicit casting, a handler for `CORBA::SystemException` matches all system exceptions (because all system exception classes are derived from class `CORBA::SystemException`). Therefore it should normally appear after all handlers for specific system exceptions. Refer to the ***Orbix Programmer's Reference C++ Edition*** for a list of system exceptions.

You can use the message output by the `operator<<()` function on class `CORBA::SystemException` to determine the type of system exception that occurred. A handler for an individual system exception is only required when specific action is to be taken if that exception occurs.

Throwing Exceptions in a Server

This section shows how to extend the definition of the function `BankExceptions_BankImpl::create_account()` to raise an exception, using the normal C++ `throw` statement. The function `newAccount()` can be coded as follows:

```
// C++
// In file bankexceptions_bankimpl.cxx

BankExceptions::Account_ptr
BankExceptions_BankImpl::create_account (const char*  name,
   CORBA::Environment&) throw(BankExceptions::Bank::CannotCreate)
{
   int empty = -1;
   int exists = -1;
   int i = 0;
   for ( ; i < MAX_ACCOUNTS; ++i) {
      if (CORBA::is_nil(m_accounts[i])) {
         empty = i;
      }
      else if (strcmp(m_accounts[i]->name(), name) == 0) {
         exists = i;
         break;
      }
   }
   // Test for errors and throw an exception if a problem occurs.
   if (exists != -1) {
      cout << "create_account: failed because name exists" << endl;
      throw BankExceptions::Bank::CannotCreate
```

```
            ("Account with same name already exists.");
    }
    else if (empty == -1) {
       cout <<"create_account: failed because no more space"<<endl;
       throw BankExceptions::Bank::CannotCreate
             ("No more space for new accounts.");
    }

    // No errors - create an account with zero balance.
    m_accounts[empty] = new BankExceptions_AccountImpl(name, 0.0);
    cout << "create_account: Created account with name:"
         << name << endl;

    // Duplicate the returned reference.
    return BankExceptions::Account::_duplicate(m_accounts[empty]);
}
```

This code uses the automatically-generated constructor of class `BankExceptions::Bank::CannotCreate` to initialize the exception's `reason` member with the strings "`Account with name already exists`" and "`No more space for new accounts`".

Information Available in System Exceptions

System exceptions have two member functions that you can use in some applications:

- `completed()`
- `minor()`

completed()

The `completed()` function returns an `enum` type that indicates how far the operation or attribute call ed before the exception was raised. The values are:

`COMPLETED_NO`	The system exception was raised before the operation or attribute call began to execute.
`COMPLETED_YES`	The system exception was raised after the operation or attribute call completed its execution.
`COMPLETED_MAYBE`	It is uncertain whether or not the operation or attribute call started execution, and, if it did, whether or not it completed. For example, the status will be `COMPLETED_MAYBE` if a client's host receives no indication of success or failure after transmitting a request to a target object on another host.

minor()

The `minor()` function returns an `unsigned long` value to give more details of the particular system exception raised. For example, if the `COMM_FAILURE` system exception is caught by a client, it can access the minor field of the system exception to determine why this occurred. Each system exception has a set of minor values associated with it, and those for `COMM_FAILURE` include `TIMEOUT` and `STRING_TOO_BIG`.

Throwing a System Exception

In some circumstances you may need to throw a system exception. You can specify the system exception's minor field and completion status using the constructor:

```
// C++
SystemException(ULong minor_id,
    CompletionStatus completed_status);
```

The following line of code illustrates the use of this constructor by throwing a `COMM_FAILURE` exception with minor code `TIMEOUT` and completion status `COMPLETED_NO`:

```
// C++
throw CORBA::COMM_FAILURE(TIMEOUT, COMPLETED_NO);
```

Using Inheritance of IDL Interfaces

This chapter describes how to implement inheritance of IDL interfaces, using an example named BankInherit. This example builds on the concepts illustrated in the BankSimple and BankExceptions examples in "Using and Implementing IDL Interfaces" and "Exception Handling in Orbix", respectively.

You can define a new IDL interface that uses functionality provided by an existing interface. The new interface inherits or derives from the base interface. IDL also supports multiple inheritance, allowing an interface to have several immediate base interfaces. This chapter shows how to use inheritance in Orbix using the BankInherit example.

A version of the source code for the example described in this chapter is available in the `demos\bankinherit` directory of your Orbix installation.

The IDL Interfaces

The IDL for the BankInherit example demonstrates the use of single inheritance of IDL interfaces:

```
// IDL
// In file bankinherit.idl

#include "bankexceptions.idl"

module BankInherit {
    interface CheckingAccount;  // forward reference

// BetterBank manufactures checking accounts.
1   interface BetterBank : BankExceptions::Bank {
2           // New operation to create new checking accounts.
            CheckingAccount create_checking (in string name,
                in BankExceptions::CashAmount overdraft)
                raises(CannotCreate);
    };

    // New CheckingAccount interface.
3   interface CheckingAccount : BankExceptions::Account {
        readonly attribute BankExceptions::CashAmount overdraft;
    };
};
```

This IDL can be explained as follows:

1. `BetterBank` inherits the operations of `BankExceptions::Bank` and adds a new operation to create checking accounts. You do not need to list the account operations from `BankExceptions::Bank` because these are now inherited.
2. The new `create_checking()` operation added to interface `BetterBank` manufactures `CheckingAccount`s.

3. The new interface `CheckingAccount` derived from interface `BankExceptions::Account`. `CheckingAccount` has an overdraft limit, and the implementation allows the balance to become negative.

The Generated C++ Code

The IDL compiler produces the following IDL C++ classes for the `BankInherit` IDL interface:

```
// C++
// The file bankinherit.hh
...
#include <CORBA.h>
...
class CheckingAccount: public virtual
    BankExceptions::Account {
    // Various details for Orbix.
    public:
    // Various details for Orbix.
    virtual BankExceptions::CashAmount overdraft
(CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
};
...
class BetterBank: public virtual BankExceptions::Bank{
    // Various details for Orbix.
    public:
    // Various details for Orbix.
...
    virtual BankInherit::CheckingAccount_ptr
        create_checking(
         const char*  name, BankExceptions::CashAmount
            overdraft, CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
};
```

Implementation Class Hierarchies

The class hierarchy for IDL C++ classes produced by the IDL compiler directly corresponds to the interface hierarchy given in the IDL source files. Figure 17 shows the inheritance hierarchy, using the BOAImpl approach. For simplicity, this diagram omits some details (for example, an implementation class also inherits from its corresponding IDL C++ class).

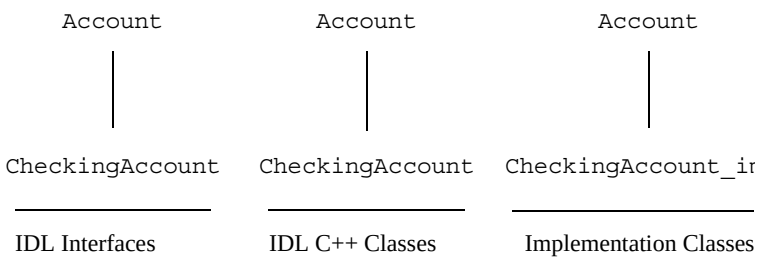


Figure 17: *IDL and Corresponding C++ Class Hierarchies*

The Implementation Classes

You can code the `CheckingAccount` interface using the BOAImpl or the TIE approach.

The BOAImpl Approach

Using the BOAImpl approach, you can implement the `CheckingAccount` IDL interface as follows:

```
// C++
// In file bankinherit_accountimpl.h
...
#include "bankinherit.hh"
#include "bankexceptions_accountimpl.h"

class BankInherit_CheckingAccountImpl : public virtual
    BankExceptions_AccountImpl, public virtual
    BankInherit::CheckingAccountBOAImpl {
public:
    // IDL operation
    virtual void withdraw(
    BankExceptions::CashAmount amount, CORBA::Environment&)
    throw (BankExceptions::Account::InsufficientFunds);

    // IDL attributes
    virtual BankExceptions::CashAmount
    overdraft(CORBA::Environment&)
    throw();

    // C++ operations
    BankInherit_CheckingAccountImpl(
    const char* name,
    BankExceptions::CashAmount balance,
    BankExceptions::CashAmount overdraft)
    throw ();

    virtual ~BankInherit_CheckingAccountImpl() throw();
protected:
    BankExceptions::CashAmount m_overdraft;
    ...
};
```

`BankInherit_CheckingAccountImpl` is the application implementation class. Using the BOAImpl approach, this class inherits from the IDL-generated BOAImpl class.

You can implement the `BetterBank` IDL interface as follows:

```
// C++.
// In file bankinherit_bankimpl.h

#include "bankinherit.hh"
#include "bankexceptions_bankimpl.h"

class BankInherit_BetterBankImpl : public virtual
    BankExceptions_BankImpl, public virtual
    BankInherit::BetterBankBOAImpl {
public:
    // IDL operations
    virtual BankInherit::CheckingAccount_ptr create_checking(
        const char*  name, BankExceptions::CashAmount overdraft,
        CORBA::Environment&)
```

```
    throw(BankExceptions::Bank::CannotCreate);

    // C++ operations
    BankInherit_BetterBankImpl() throw();
    virtual ~BankInherit_BetterBankImpl() throw();
    ...
};
```

`BankInherit_BetterBankImpl` is the application implementation class. Using the BOAImpl approach, this class inherits from the IDL-generated BOAImpl class.

The `create_checking()` operation is implemented as follows:

```
// C++
// In file bankinherit.bankimpl.cxx
...
#include "bankinherit_bankimpl.h"
#include "bankinherit_accountimpl.h"

BankInherit::CheckingAccount_ptr
BankInherit_BetterBankImpl::create_checking (
    const char*  name, BankExceptions::CashAmount overdraft,

   CORBA::Environment&)throw(BankExceptions::Bank::CannotCreate)
   {
    ...
    // Create an account with 0 balance using the BOAImpl
   approach.
    BankInherit::CheckingAccount_var
   newly_created_checkingaccount;
    newly_created_checkingaccount
        = new BankInherit_CheckingAccountImpl(name, 0.0,
   overdraft);
    m_accounts[empty]
        = BankInherit::CheckingAccount::_duplicate(
                          newly_created_checkingaccount);
    ...
    // Duplicate the returned reference.
    return BankInherit::CheckingAccount::_duplicate(
                               newly_created_checkingaccount);
}
```

The return statement is slightly different in `create_checking()` than for `create_account()`. This is because you cannot call `_duplicate()` on a `CheckingAccount` object stored in the `Account` array. The temporary variable `newly_created_checkingaccount` is used to get around this problem.

The TIE Approach

Using the TIE approach, the `CheckingAccount` IDL interface could be implemented as follows:

```
// C++
...
class BankInherit_CheckingAccountImpl :
    public virtual BankExceptions_AccountImpl {
public:
    // Same as for BOAImpl.
    ...
}

// DEF_TIE macro call.
DEF_TIE_BankInherit_CheckingAccount(BankInherit_CheckingAccountImpl)
```

This code is explained as follows:

1. The class `BankInherit_CheckingAccountImpl` inherits from `BankExceptions_AccountImpl` only. It does not need to inherit from the IDL-generated BOAImpl class.
2. Indicates that `BankInherit_CheckingAccountImpl` implements `BankInherit::CheckingAccount`. This generates a TIE class `TIE_CheckingAccount(BankSimple_CheckingAccountImpl)`.

The `BetterBank` IDL interface can therefore be implemented as follows:

```
// C++
...
class BankInherit_BetterBankImpl :
    public virtual BankExceptions_BankImpl {
public:
    // Same as for BOAImpl.
    ...
}

// DEF_TIE macro call.
DEF_TIE__BankInherit_BetterBank(BankInherit_BetterBankImpl);
```

Using the TIE approach, you can implement the `create_checking()` operation as follows:

```
// C++
...
BankInherit::CheckingAccount_ptr
BankInherit_BetterBankImpl::create_checking (
    const char*  name, BankExceptions::CashAmount overdraft,
CORBA::Environment&)throw(BankExceptions::Bank::CannotCreate)
{
    ...
    // Create an account with zero balance using TIE approach.
    BankInherit::CheckingAccount_var
   newly_created_checkingaccount;
    newly_created_checkingaccount
        = new TIE_BankInherit_CheckingAccount(
        BankInherit_CheckingAccountImpl)
        (new BankInherit_CheckingAccountImpl(name, 0.0,
                overdraft));
    ...
}
```

Using Inheritance in a Client

A client can proceed to manipulate `CheckingAccount`s in a similar way to `Account`s in, "Handling Exceptions in a Client" on page 127:

```
// C++
// In file BankMenu.cxx

#include "bankmenu.h"
#include <it_demo_menu.h>

// BankMenu constructor, takes a Bank reference.
BankMenu::BankMenu(BankInherit::BetterBank_ptr bank)
      throw() : m_betterbank(BankInherit::BetterBank::_duplicate(bank))
{}

// BankMenu destructor.
BankMenu::~BankMenu() throw(){
// Nothing to do - Bank_var automatically releases reference
}

// Start main menu loop.
void
BankMenu::start() throw() {
   // Ensure the bank reference is valid.
   if (CORBA::is_nil(m_betterbank)) {
      cout << "Cannot proceed - bank reference is nil";
   }
   else {
   // Loop printing the menu and executing selections
   ...
}
...
// Calls create_checking and runs an account menu.
void BankMenu::do_create_checking() throw(CORBA::SystemException){
   cout << "Enter account name: " << flush;
   CORBA::String_var name = IT_Demo_Menu::get_string();
   ...
   try {
      BankInherit::CheckingAccount_var checkingaccount =
                      m_betterbank->create_checking(name, 100);

      // Start a sub-menu with the returned account reference
      AccountMenu sub_menu(checkingaccount);
      sub_menu.start();
   }
   catch (const BankExceptions::Bank::CannotCreate&cant_create) {
      cout << "Cannot create an account, reason: "
      << cant_create.reason << endl;
   }
}

// Calls find_account and runs an account menu.
void BankMenu::do_find() throw(CORBA::SystemException) {
   // Same as for BankExceptions.
...
}
```

The client implementation is not affected by the approach used to implement the server—either TIE or BOAImpl.

Multiple Inheritance of IDL Interfaces

IDL supports multiple inheritance as shown in the following example:

```
// IDL
module BankSimple {
    typedef float CashAmount;
    interface Account;

    interface Bank {
        ...
    };

    interface Account {
        readonly attribute string name;
        readonly attribue CashAmount balance;

        void deposit(in CashAmount amount);
        void withdraw(in CashAmount amount);
    };

    // Derived from interface Account.
    interface CheckingAccount : BankSimple::Account {
        readonly attribute CashAmount overdraftLimit;
    };

    // Derived from interface Account.
        interface DepositAccount : BankSimple::Account {
    };

    // Indirectly derived from interface Account.
    interface PremiumAccount : CheckingAccount, DepositAccount {
    };
}
```

The corresponding IDL C++ classes use multiple inheritance:

```
    // C++
    // The file "bank.hh".
    #include <CORBA.h>

    class BankSimple::Account :
        public virtual CORBA::Object {
        // As before.
    };

    class CheckingAccount :
        public virtual BankSimple::Account {
        // As before.
    };

    class DepositAccount :
        public virtual BankSimple::Account {
        // ...
    };
```

```
class PremiumAccount :
    public virtual CheckingAccount,
        public virtual DepositAccount {
    // ...
};
```

IDL forbids any ambiguity arising due to name clashes of operations and attributes when two or more direct base interfaces are combined. This means that an IDL interface cannot inherit from two or more interfaces with the same operation or attribute name. You can, however, inherit two or more constants, types or exceptions with the same name from more than one interface. However, these must be qualified with the name of the interface (an IDL-scoped name must be used).

Orbix Connections and Events

Orbix applications need to control how Orbix processes events such as operation calls and establishing connections between clients and servers. To do this, applications communicate with the ORB through a direct API that allows them to configure the behavior of Orbix. This chapter outlines this API and describes how you can use it to adapt the Orbix connection establishment and event processing models.

This chapter acts as a guide to the main connection and event management functions in Orbix. You should read this chapter for an overview of these functions and refer to the ***Orbix Programmer's Reference C++ Edition*** for details of particular functions required in your applications.

Overview of the Direct API to Orbix

On the client-side, the interface to Orbix is presented via the class `CORBA::ORB`. On the server, the class `CORBA::BOA` (a derived class of `CORBA::ORB`) specifies the interface to Orbix, as shown in Figure 18.

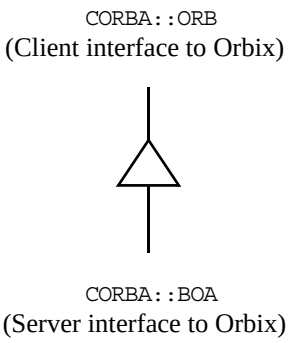


Figure 18: *Interfaces to Orbix on Client and Server*

The acronym BOA stands for *Basic Object Adapter*. An *Object Adapter* is the CORBA term given to the environment in which server applications run. An object adapter provides services such as:

- Registration of servers.
- Instantiation of objects at runtime and creation and management of object references.
- Handling of incoming client calls.
- Dispatching of client requests to server objects.

The BOA or Basic Object Adapter is an object adapter specified by CORBA that must be provided by every ORB. An ORB may optionally provide other object adapters and a server may support a number of object adapters to serve different types of requests.

Refer to the ***Orbix Programmer's Reference C++ Edition*** for the full interface to `CORBA::ORB` and `CORBA::BOA`.

Initializing a Connection to the ORB

The CORBA standard defines how a client or server can obtain a reference to the ORB so that they can communicate with it. The function defined for this purpose is `CORBA::ORB_init()`, which you can use as follows:

```
// C++
...
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "Orbix");
```

`ORB_init()` initializes a client or server's connection to the ORB. It should not be viewed as initializing the ORB itself because the ORB is pervasive rather than just existing within the client or server. You can call any function defined on class `CORBA::ORB` (for example, `string_to_object()`) by using the pointer returned from the `ORB_init()` call.

Servers should carry out a further step, to obtain a reference to the Object Adapter, and in particular to the BOA:

```
// C++
CORBA::BOA_ptr boa = orb->BOA_init(argc, argv, "Orbix_BOA");
```

`CORBA::ORB::BOA_init()` initializes a server's connection to the BOA. An ORB may also provide other object adapters—in this case, it should provide a function to initialize a connection to each.

Functions such as `impl_is_ready()` defined on class `CORBA::BOA` can be called using the object reference returned from the `BOA_init()` call. On the client-side, you do not need to perform these steps although, for compliance to the CORBA standard, you may wish to add them.

Obtaining Initial Object References

Some object services and, in particular, the Interface Repository and the CORBAservices, can only be used by first obtaining a reference to an initial service object. The Naming Service provides a general purpose facility for doing this. When using the Naming Service, you need some way to obtain a reference to an initial Naming Service object.

CORBA addresses this difficulty by providing two operations in interface `ORB`. These provide the facilities of a simplified Naming Service, in which (flat, rather than hierarchical) names can be resolved to obtain initial references to important objects in the system:

```
module CORBA {
    interface ORB {
        ...
        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;

        exception InvalidName {};

        ObjectIdList list_initial_services();

        Object resolve_initial_references
                (in ObjectId identifier)
                raises (InvalidName);
    };
};
```

Only a small group of names are understood by `resolve_initial_references()`, and these are listed by `list_initial_services()`. Currently only strings “NameService” and “InterfaceRepository” are supported. The function `resolve_initial_references()` returns an object reference, which must be narrowed to the correct object type.

Managing Orbix Connections and Events

When an Orbix client first contacts a server, a single communication channel is established between the client-server pair. This connection is then used for all subsequent operation calls from the client to the server. The connection is closed only when either the client or the server exits.

If the server makes operation calls, known as *callbacks*, to objects that exist in the client, the connection usage depends on which communications protocol you are using. If your applications communicate over the Orbix protocol, all communications in both directions use a single client-server connection. However, if your applications use the CORBA Internet Inter-ORB Protocol (IIOP), Orbix opens a second connection *from* the server *to* the client when the server attempts its first callback operation. By default, all IIOP operation calls are only transmitted in one direction across a client-server connection as specified by CORBA. Refer to the chapter “Callbacks from Servers to Clients” for more information on using callbacks in Orbix.

When a connection has been established between a client and server, you must instruct Orbix to process incoming operation calls. Orbix does this by monitoring the file descriptor associated with each client-server connection and responding to events on the file descriptor.

This section highlights some of the Orbix functions that allow you to manage Orbix connection establishment and event processing.

Establishing Connections between Clients and Servers

This section describes the following issues associated with establishing a connection between a client and a server:

- Setting a timeout on a connection attempt.
- Specifying the number of connection attempt retries.
- Filtering bad connection attempts in servers.
- Reconnecting to a server that has crashed and restarted.
- Receiving callbacks from Orbix to application code when connections are opened or closed.

Setting a Timeout on a Connection Attempt

By default, there is a timeout of `30` seconds for establishing connection from a client to a server to confirm that both are operational. This timeout can be changed using the function `CORBA::ORB::connectionTimeout()`.

Under some circumstances, `CORBA::ORB::connectionTimeout()` has no effect. For example, if the server's host is known but is down or unreachable, a TCP/IP connect can block for a considerable time depending on the target operating system. In these circumstances, you can use the function `CORBA::ORB::abortSlowConnects()` to abort connection attempts that exceed the value specified in `connectionTimeout()`.

Specifying Connection Attempt Retries

If a connection cannot be made on the first attempt because the server cannot be contacted, Orbix retries the attempt every two seconds until either the call can be made or until there have been too many retries. You can use the function `CORBA::ORB::maxConnectRetries()` to set the maximum number of retries that should be attempted. The default number is `10`.

Filtering Bad Connection Attempts

By default, an exception is raised if a bad connection is made to a server waiting on the event handling functions (`CORBA::BOA::impl_is_ready()`, `CORBA::BOA::obj_is_ready()`, `CORBA::BOA::processEvents()`). Such bad connections can be caused by, for example, a server that cannot interpret the data that it accesses.

You may wish to allow Orbix to handle such attempts without raising an exception. Refer to the ***Orbix Programmer's Reference C++ Edition*** entry for `CORBA::BOA::filterBadConnectAttempts()` for details.

Reconnecting to a Failed Server

When a server exits while a client is still connected, the next invocation by the client raises a system exception of type `CORBA::COMM_FAILURE`. If the client attempts another invocation, Orbix automatically tries to re-establish the connection.

This default behavior can be changed by passing the value `1` (true) to the function `CORBA::ORB::noReconnectOnFailure()`. Then, all client attempts to contact a server subsequent to closure of the communications channel raises a `CORBA::COMM_FAILURE` system exception.

Receiving Callbacks for New or Closed Connections, and FD Handling Events

Orbix allows a client or server to receive a callback for certain connection and file descriptor (FD) events. Callbacks exist for opening and closing a connection to another Orbix program. Callbacks exist for when the number of FDs used by Orbix reaches soft and hard limits set by the user.

To receive such callbacks, first define a class that inherits from the Orbix class `CORBA::IT_IOCallback`. Class `CORBA::IT_IOCallback` is defined as follows:

```
class IT_IOCallback
{
   public:

      // The following functions are called when an Orbix file
      // descriptor opens or closes.
      virtual void OrbixFDOpen(int fd) {};
      virtual void OrbixFDClose(int fd) {};

      // The following functions are called when activity is
      // detected on a foreign fd.

      // A registered foreign fd is ready for reading.
      virtual void ForeignFDRead(int fd) {};

      // A registered foreign fd is ready for writing.
      virtual void ForeignFDWrite(int fd) {};

      // A registered foreign fd has fired for exceptions.
      virtual void ForeignFDExcept(int fd) {};

      // The following functions are called when the number of FDs
      // used by Orbix hits a soft or hard limit set by the user.

      // The low-watermark (soft limit) of FDs has been consumed
      virtual void AtOrbixFDLowLimit (int numFDsUsed);

      // The hard limit of FDs has been consumed, so Orbix is no
      // longer listening for new connections (which would consume
      // another FD).
      virtual void StopListeningAtFDHigh (int numFDsUsed);

      // Orbix has resumed listening after number of FDs has gone
      // below the hard limit
      virtual void ResumeListeningBelowFDHigh (int numFDsUsed);
};
```

Your sub-class of `CORBA::IT_IOCallback` should override one or more of the `IT_IOCallback` member functions. For example, if you override `OrbixFDOpen()`, this function is called each time a connection to your application is opened. Similarly, if you override `OrbixFDClose()`, this function is called each time a connection to your application is closed. Both `OrbixFDOpen()` and `OrbixFDClose()` receive the file descriptor associated with the relevant Orbix connection as a parameter.

When you implement a derived class of `CORBA::IT_IOCallback`, create an instance of this class and register this object with Orbix by calling `CORBA::ORB::registerIOCallbackObject()` on the ORB. This function is described in the ***Orbix Programmer's Reference C++ Edition***.

The functions `ForeignFDRead()`, `ForeignFDWrite()`, and `ForeignFDExcept()` allow you to integrate Orbix event processing with foreign event processing as described in "Integrating the Orbix Event Loop with Foreign Events" on page 146.

The functions At`OrbixFDLowLimit()`, `StopListeningAtFDHigh()`, and `ResumeListeningBelowFDHigh()`, combined with the configuration variables `IT_FD_WARNING_NUMBER` and `IT_FD_STOP_LISTENING_POINT`, give users the ability to monitor consumption of FDs. Note that FDs are shared by the process and are needed not only by Orbix, but also for database connections and file i/o that the user's code may use. When the number of Orbix FDs reaches `IT_FD_WARNING_NUMBER`, either on the way up or the way down, At`OrbixFDLowLimit()` is called. When the number of Orbix FDs reaches `IT_FD_STOP_LISTENING_POINT`, then `StopListeningAtFDHigh()` is called. Once an Orbix FD is freed up or the number of FDs made available to Orbix is increased, then `ResumeListeningBelowFDHigh()` is called.

Event Processing in Orbix

This section describes the following issues associated with processing Orbix events:

- Orbix event processing functions.
- Integrating the Orbix event loop with foreign events.
- Ensuring that servers process events while clients are connected.
- Setting timeouts on operation calls from clients.

Orbix Event Processing Functions

The function `impl_is_ready()`, defined on class `CORBA::BOA`, allows you to initialize a server and start processing incoming connection attempts and operation calls on existing connections. Class `CORBA::BOA` also provides several other event processing functions that allow you to handle incoming events in a client or in a server that has already been initialized.

The relevant functions are `CORBA::BOA::processNextEvent()`, `CORBA::BOA::processEvents()` and `CORBA::BOA::obj_is_ready()`. You can also test whether or not there is an outstanding event using `CORBA::BOA::isEventPending()`. Refer to the relevant entries in class `CORBA::BOA` of the ***Orbix Programmer's Reference C++ Edition*** for details.

Integrating the Orbix Event Loop with Foreign Events

When you call an Orbix event processing function, Orbix monitors all file descriptors associated with its event loop. This file descriptor set includes each file descriptor associated with an open connection from another Orbix program.

If you wish to integrate Orbix with another system that has an event processing loop, you can do this by adding the file descriptors for the foreign system to the Orbix event loop.

To add foreign file descriptors to the Orbix event loop, call one of the following functions defined in class CORBA::ORB:

```
void addForeignFD
    (const int fd, unsigned char aState);
void addForeignFDSet
    (fd_set& theFDset, unsigned char aState);
```

To remove foreign file descriptors from the Orbix event loop, call one of the following functions:

```
void removeForeignFD
    (const int fd, unsigned char aState);
void removeForeignFDSet
    (fd_set& theFDset, unsigned char aState);
```

There are three sets of foreign file descriptors registered with the Orbix event loop: one set is monitored for reads, another for writes, and the third for exceptions. You can register a file descriptor in one or more of these sets. To do this, specify the values FD_READ, FD_WRITE, FD_EXCEPTION (or any logical combination of these values) in the aState parameter, passed each of the registration functions.

When Orbix detects an event on a foreign file descriptor, it attempts to call a function in your application code. To receive this callback, implement the class CORBA::IT_IOCallback, as described in "Receiving Callbacks for New or Closed Connections, and FD Handling Events" on page 145 and override one of the functions ForeignFDRead(), ForeignFDWrite(), and ForeignFDExcept().

Processing Events while Clients are Still Connected

By default, the event processing functions impl_is_ready(), obj_is_ready(), processEvents() and processNextEvent(), (defined in class CORBA::BOA) time out when a user-defined or defaulted period has elapsed between
events; for example, an incoming operation call, or a connection or disconnection by a client.

Consequently, impl_is_ready() can time out when its clients are idle for a period. A server may prefer to remain active while there are clients connected, active or not. Then the server should make the following call:

```
// C++
CORBA::Orbix.setNoHangup(1); // 1 for true.
```

Refer to the entry for CORBA::BOA::setNoHangup() in the ***Orbix Programmer's Reference C++ Edition*** for full details.

Setting Timeouts on Operation Calls

An operation call that is not defined as oneway can be given a timeout specified in milliseconds. If a reply is not received within the given timeout interval, the invocation fails with a CORBA::COMM_FAILURE exception.

The timeout for a call can be given by setting a value in an Environment, using the following function:

```
// C++
// In class CORBA::Environment.
void timeout(CORBA::Long);
```

For example:

```
// C++
CORBA::Environment& env;
CORBA::Long timeoutValue = ...;
Account_var aVar = ...;
try {
    env.timeout(timeoutValue);
    aVar->deposit(12.00, env);
}
catch (const CORBA::COMM_FAILURE&) {
    cout << "---Timed out after" << timeoutValue
         << "msecs..." << endl;
}

catch (const CORBA::SystemException& se) {
    cout << "Unexpected exception:" << endl
        <<& se;
}
```

The value set by the `CORBA::Environment::timeout()` function remains active until reset for the environment for which it was set.

A timeout can also be specified in a `_bind()` call:

```
// C++
CORBA::Environment& env;
CORBA::Long timeoutValue = ...;
Bank_var bVar;
try {
    env.timeout(timeoutValue);
    bVar = Bank::_bind(":AIB","",env);
}
catch (const CORBA::COMM_FAILURE&) {
    cout << "--- Timed out after " << timeoutValue
        << "msecs..." << endl;
}
```

In this case, the timeout applies to the implicit ping call attempted during binding.

The timeout, if any, in an `Environment` variable can be read using the parameterless function:

```
// C++
// In class CORBA::Environment.
CORBA::Long timeout()
```

As an alternative, timeouts can be set for *all* remote calls by calling the following function on the ORB object:

```
//C++
// In class CORBA::ORB.
unsigned long defaultTxTimeout
    (CORBA::ULong val, CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv())
```

This function returns the previous value. The value set by this function is then used for all remote calls. However, if a timeout is set in an `Environment`, it supersedes any value set globally in the ORB. By default, no call has a timeout, that is, the default timeout is infinite.

If a remote call establishes a connection between the client and server, then there is a separate timeout on connection establishment that can be controlled by the `connectionTimeout()`

function defined in class `CORBA::ORB`. The timeouts specified by `CORBA::ORB::defaultTxTimeout()` or `CORBA::Environment::timeout()` become effective once a connection is established.

Advanced Programming Topics

This chapter presents a number of advanced topics that have not been covered in previous chapters.

The topics covered in this chapter are:

- How to write applications where the client and server are collocated—that is, within the same address space.
- How to determine whether a specific object is local or remote.
- How to obtain a pointer to an implementation class.
- How to raise an exception if the correct proxy code is not available in a client.
- How multiple implementations can be provided for the same IDL interface.
- How an implementation class may implement multiple interfaces.
- How to use the CORBA type `Context`.
- How to modify the level of diagnostic messages displayed by Orbix.

Developing Collocated Clients and Servers

For some applications, it is useful to use IDL to define the interfaces between objects, even if these objects are not distributed. Further, the objects in some applications may or may not be distributed, depending on how the application is configured by its installer. It is useful to be able to write application code that can work efficiently in all these cases.

To address these issues, Orbix supports the collocation of client and server objects within the same address space. When a bind call is made by a client or server, Orbix will first look for the object in the caller's address space, unless the bind call specifies a remote host. If the target object is found in that address space, subsequent calls on the object are very efficient. This is because direct C++ function calls are used from the client to the server application code, and the Orbix runtime is bypassed.

Collocation can be enforced by calling the following function:

```
// C++
// In class CORBA::ORB.
orb->collocated(1);
```

This call controls the lookup mechanism: it prevents binding to objects outside the current process's address space. This function returns the previous setting as a `CORBA::Boolean`.

If the target object cannot be found within the application's address space, Orbix normally tries to locate the object in the same or a different node. However, if collocation is set, Orbix *never* tries to bind to an object outside of the caller's address space. If an object is not found in the caller's address space, `_bind()` raises a `CORBA::INV_OBJREF` system exception.

Calls to `collocated(1)` are normally made during the initialization of a combined client/server. However, collocation can be unset (thereby reinstating remote binding) at any time by calling `collocated(0)`.

Testing for the Presence of Collocation

A program can test whether or not collocation is currently set by making the function call:

```
// C++
// In class CORBA::ORB.
CORBA::Boolean isOn = orb->collocated();
```

This returns a `CORBA::Boolean` value `1` (true) if collocation is set, and returns `0` (false) otherwise.

A program may wish to use this so that it can create local objects if collocation is set, but not create these objects otherwise; in the latter case, it expects these objects to be created and managed by a remote server.

Writing Code for both Collocation and Distribution

The following code works in both the collocated and the distributed case. Either of these two cases can be selected at runtime, perhaps from a command-line switch. The general strategy for a collocated application is to write a mainline that first conducts the usual server-side initialization (and in particular creates target Orbix objects for the server application—here just the `Bank` object), and then continues with the mainline of the client application. In the distributed case, some server, which is not shown here, is instead responsible for creating the target objects.

```
// C++

// Only the TIE approach is shown.
// The BOAImpl approach is very similar.
// Assume we have DEF_TIE_Bank(BankImpl);
...
main(int argc, char** argv) {
    Bank_var localBankVar, remoteBankVar;
    Account_var aVar;

    // Use, for example, the command line arguments
    // to decide whether or not to make this call:
    if (...) {
        orb->collocated(1); // true
        localBankVar =
            new TIE_Bank(BankImpl)(new BankImpl());
    }
    try {
        // The bind to 'srv' is done locally
        // if collocated; else remote bind:
        // Note: In general S for the local process
        // should be different than S for the remote
        // process.
        remoteBankVar = Bank::_bind("M:S", "H");
        aVar = p->newAccount("jack");
        aVar->makeLodgement(100.00);
```

```
        cout << "balance is "
                << aVar->balance() << endl;
    }
    catch (const CORBA::SystemException& se) {
        cout << "Unexpected exception:" << endl
                << se;
    }
    catch (...) {
    }
}
```

An example of collocation is supplied in the `\Micro Focus\Orbix 3.3 \demos\orbixcxx\colocate` directory of your Orbix installation.

Note: The example code shown here assumes that a remote server is responsible for creating the target objects if collocation is not set. Otherwise, it would be necessary to call `impl_is_ready()`.

Determining Locality of Objects

You can use the `CORBA::Object::_isRemote()` function to determine whether or not a reference to an IDL C++ class is remote—that is, whether or not the object it references is in a different address space on the same or a different host). An example of its use is shown below:

```
// C++
// Bank server mainline.
main() {
    Bank_var bVar; // IDL C++ class.

    // The BOAImpl approach.
    bVar = new BankImpl;
    bVar->_marker("College_Green");

    // The TIE approach.
    // bVar = new TIE_Bank(BankImpl)
    //         (new BankImpl(), "College_Green");

    if (!bVar->_isRemote())
       cout << "Object is local (as expected!)");
    // else - IMPOSSIBLE: object *IS* local.
}

// C++
// Client mainline.
main() {
    // Bind to *any* Bank service.
    Bank_var bVar = Bank::_bind("M:S", "H");

    if (bVar->_isRemote())
       cout << "Object is remote (as expected!)"
            << endl;
    // else object is local (or non-existent).
}
```

Determining Hierarchy of Objects

When a reference to a remote object enters a client' s (or server's) address space, Orbix constructs a proxy for that object. This proxy (a normal C++ object) is constructed to execute the proxy code corresponding to the actual interface of the true object it represents.

Consider the base IDL interface:

```
// IDL
interface Account {
  ...
};
```

and the derived IDL interfaces:

```
interface CurrentAccount : Account {
  ...
};
```

and

```
interface PremiumAccount : CurrentAccount {
  ...
};
```

Consider if a CORBA process was linked with the IDL compiler generated proxy code for the `CurrentAccount` object but *not* with the generated proxy code for the `PremiumAccount` object.

If a server object within this process has an operation of the form:

```
// IDL
// In some interface.
void bankOp(in Account_ptr petal);
```

and a reference to a `PremiumAccount` object (the derived interface of `CurrentAccount`) is passed as a parameter to this operation, Orbix sets up a proxy for an `Account` interface in the local address space.

Orbix knows that `PremiumAccount` must be a derived type of `Account`, otherwise the client's invocation of the operation `BankOp()` using a `PremiumAccount` object would not have been allowed.

Now, if the application itself knows of the complete relationship shown above, that is, that a `PremiumAccount` object is also a derived instance of `CurrentAccount`, it is free to attempt a call to `CurrentAccount::_narrow()` on the `Account` proxy object to obtain a `CurrentAccount` proxy object.

However, as it has not been linked with the proxy code for the `PremiumAccount` interface, Orbix does not know of the complete relationship between the types `Account`, `CurrentAccount` and `PremiumAccount`.

If `useRemoteIsACalls` is enabled, Orbix invokes the `_is_a()` operation on the remote object to determine if `PremiumAccount` type is in fact a derived type of the `CurrentAccount` type.

The remote process, knowing the complete relationship between `Account`, `CurrentAccount`, and `PremiumAccount` then returns `TRUE` to indicate that a `PremiumAccount` object is in fact a derived object of the `CurrentAccount` interface.

Orbix can then narrow the `Account` proxy object to a `CurrentAccount` proxy object using the generated proxy code for the `CurrentAccount` interface.

If this is not acceptable, you should call `useRemoteIsACalls()` on the `CORBA::Orbix` object, passing 0 (`FALSE`) for the first parameter. The default setting is 1 (`TRUE`).

Setting the value to 0 (`FALSE`) means that Orbix raises an exception if the local process has not been linked with the proxy code for all required interfaces.

Note: It is not possible to narrow the local `Account` object to a `PremiumAccount` object as the local process has not been linked with the generated proxy code for the `PremiumAccount` object.

Casting from Interface to Implementation Class

This section describes how to cast, when using the BOAImpl approach, from an interface class to an implementation class written by a programmer. Although this is not frequently required, it can be useful in some cases.

Consider interface `Account`, and the C++ implementation class `AccountImpl` defined as follows:

```
// C++
class AccountImpl : public virtual AccountBOAImpl {
    ...
};
```

The overall class hierarchy is shown in Figure 19.

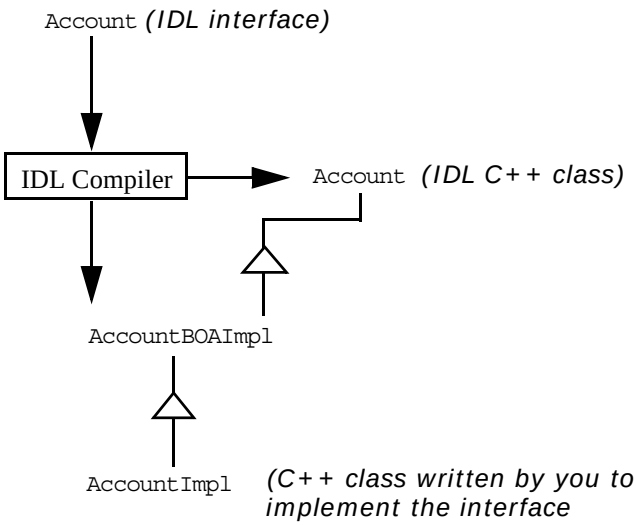


Figure 19: *Casting From Interface Class to Implementation Class Using the BOAImpl Approach*

If you have an object reference for an `Account`, there is a difficulty casting this to a pointer to an `AccountImpl`. C++ prohibits this cast because the inheritance between `AccountBOAImpl` and `Account` is virtual.

Casts from interface to implementation class are not frequently required, because invoking a function on the `Account` object reference is sufficient. However, you can add an extra member function (not defined in the IDL interface) to the implementation class, and this is only available for use if you have a pointer to the implementation class.

Orbix provides a DEREF() macro that, when called on a TIE object, returns a pointer to an implementation object. This macro implicitly calls the function CORBA::Object::_deref(). To cast from an interface to an implementation class using the BOAImpl approach, you should first redefine CORBA::Object:: _deref() function in the implementation class:

```
// C++
class AccountImpl : public virtual AccountBOAImpl{
    ....
    virtual void* _deref() { return this; }
};
```

You can then use the DEREF() macro to achieve the cast as follows:

```
// C++
Account_ptr aPtr = . . . .;
AccountImpl* p_i = (AccountImpl*) DEREF(aPtr);
```

If _deref() is not redefined by AccountImpl, then it inherits an implementation that returns a pointer to the BOAImpl class.

Naturally, the need for the cast could be removed by defining the extra functions as IDL operations in the IDL interface. However, this would make these operations available to remote processes, possibly against the requirements of the application. In addition, some C++ functions cannot be translated into IDL in a straightforward way.

Actions when Proxy Code is Unavailable

When a reference to a remote object enters a client or server address space, Orbix constructs a proxy for that object. This proxy (a normal C++ object) is constructed to execute the proxy code corresponding to the actual interface of the true object it represents.

Hence, if a server object has an operation of the form:

```
// IDL
// In some interface.
void op(in Account a);
```

and if a reference to a remote CurrentAccount (a derived interface of Account) is passed as a parameter to this operation, Orbix tries to set up a proxy for a CurrentAccount in the server address space.

If the server was not linked with the IDL-compiler generated proxy code for CurrentAccount, Orbix instead creates a proxy for an Account in the server address space. This means that, Orbix uses the static rather than the dynamic type of the parameter. The same applies when an object reference enters a client.

If resorting to the static type is unacceptable, call the following function on the ORB object, passing a false value for the first parameter:

```
// C++
// In class CORBA::ORB.
unsigned char resortToStatic(CORBA::Boolean,
    CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
```

This function returns the previous setting; the default setting is true. Setting the value to false means that Orbix raises an exception if the server or client is not linked with the actual proxy code.

Multiple Implementations of an Interface

There may be more than one implementation of the same IDL interface:

- In the BOAImpl approach, you can define multiple classes which inherit from the same BOAImpl-class.
- In the TIE approach, you can declare further relationships using a `DEF_TIE` macro.

For example, in the BOAImpl approach, the following provides a second implementation class of the `Bank` interface:

```
// C++
class BuildingSocietyImpl :
    public virtual BankBOAImpl {
public:
    BuildingSocietyImpl();
    virtual ~BuildingSocietyImpl();

    // Functions for IDL operations.
    Account_ptr newAccount(const char* name,
       CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());

    void deleteAccount(Account_ptr a,
       CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
};
```

In the TIE approach, the following can be used:

```
// C++

class BuildingSocietyImpl {
public:
    BuildingSocietyImpl();
    virtual ~BuildingSociety_i();

    // Functions for IDL operations.
    Account_ptr newAccount(const char* name,
       CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());

    void deleteAccount(Account_ptr a,
       CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
};

DEF_TIE_Bank(BuildingSocietyImpl)
// A class TIE_Bank(BuildingSocietyImpl).
```

Both of the `TIE` classes, `TIE_Bank(BankImpl)` and `TIE_Bank(BuildingSocietyImpl)`, are now derived classes of the IDL C++ class `Bank`.

The argument to the constructor of `TIE_Bank(BankImpl)` must be a `BankImpl*`, and that of `TIE_Bank(BuildingSocietyImpl)` must be a `BuildingSocietyImpl*`:

```
// C++
Bank_ptr b1Ptr = new TIE_Bank(BankImpl)
           (new BankImpl);
Bank_ptr b2Ptr = new TIE_Bank(BuildingSocietyImpl)
           (new BuildingSocietyImpl);
```

Because the two `TIE` classes are derived classes of (the IDL C++ class) `Bank`, the pointers `b1Ptr` and `b2Ptr` can both refer to either of these two `TIE` objects:

```
// C++
b1Ptr = b2Ptr; // OK, b1Ptr now points to a
               // BuildingSocietyImpl TIE.
```

Multiple Interfaces per Implementation

In addition to being able to implement the same IDL interface using two or more different implementation classes, the *same* implementation class can implement two or more IDL interfaces, *even* if these IDL interfaces are *not* themselves related by inheritance. Consider the following two interfaces:

```
// IDL
// An IDL factory for bank accounts.
interface Bank {
    exception Reject { string reason; };

    Account newAccount(in string name)
            raises (reject);
    void deleteAccount(in Account a);
};
// An IDL management interface for accounts.
interface Manager {
    Account firstAccount();
    Account nextAccount();
    void deleteAccount(in Account a);
};
```

Here, `Bank` does not inherit from `Manager`, nor vice versa. The next two sections show how the two interfaces `Bank` and `Manager` can be implemented by the same C++ class, using the TIE approach and the BOAImpl approach, respectively.

Using the TIE Approach

Using multiple interfaces for an implementation is more straightforward in the TIE approach. First you should write a class that provides all of the functions in the two interfaces:

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

    // Functions for IDL operations:
    Account_ptr newAccount(const char* name,
       CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
    void deleteAccount(Account_ptr a,
       CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
    Account_ptr firstAccount
        (CORBA::Environment& IT_env=
        CORBA::IT_chooseDefaultEnv());
    Account_ptr nextAccount
        (CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
};
```

Now class `BigBankImpl` can implement the IDL interfaces `Bank` and `Manager` as follows:

```
// C++
// Indicate that Bank is implemented by BigBankImpl.
DEF_TIE_Bank(BigBankImpl)
// You now have a class TIE_Bank(BigBankImpl).

// Indicate that Manager is implemented by BigBankImpl.
DEF_TIE_Manager(BigBankImpl)
// You now have a class TIE_Manager(BigBankImpl).
```

An instance of `BigBankImpl` acts as an object of type `Bank` when it is accessed through a `TIE` of class `TIE_Bank(BigBankImpl)`. An instance of `BigBankImpl` acts as an object of type `Manager` when it is accessed through a `TIE` of class `TIE_Manager(BigBankImpl)`.

In addition, note that the *same object* can provide both of these interfaces:

```
// C++
// Use the same object to implement
// both Bank and Manager.

// The TIE approach.
Bank_ptr bPtr = new
    TIE_Bank(BigBankImpl)(new BigBankImpl);
Manager_ptr mPtr = new TIE_Manager(BigBankImpl)
        ((BigBankImpl*)DEREF(bPtr));
```

The `DEREF()` macro is applied to a reference to an IDL C++ class; and an explicit type cast is required. If the reference denotes a local object, `DEREF()` returns a reference to that object. If the reference is remote, `DEREF()` returns a reference to the proxy.

You can determine whether or not a reference is remote by using the function `CORBA::Object::_isRemote()`.

Using the BOAImpl Approach

Using the BOAImpl approach, `BigBankImpl` *should not* be defined as follows:

```
// C++
// Incorrect approach:
class BigBankImpl : public virtual BankBOAImpl,
        public virtual ManagerBOAImpl {
    ...
};
```

If this definition is used, it would not be possible to determine whether an object of type `BigBankImpl` was of type `BankBOAImpl` or `ManagerBOAImpl`. This is important if the two interfaces are not related by inheritance.

The natural solution is to define a new IDL interface that inherits from both `Bank` and `Manager`, and for the C++ implementation class to inherit from the BOAImpl class corresponding to that new interface.

If it is not possible to introduce the new IDL interface, you can proceed as follows. Class `BigBankImpl` can inherit from one of the BOAImpl classes, for example `BankBOAImpl`, but it should include functions to implement all of the functions in `Bank` and `Manager`:

```
// C++
class BigBankImpl : public virtual BankBOAImpl {
public:
    BigBankImpl();
    ~BigBankImpl();

    // Functions for Bank IDL operations:
    Account_ptr newAccount(const char* name,
       CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
    void deleteAccount(Account_ptr a,
       CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
    // Functions for Manager IDL operations:
    Account_ptr firstAccount
       (CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
    Account_ptr nextAccount
       (CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
};
```

Calls on the `Bank` interface can go directly to an object of type `BigBankImpl`. However, you need a second object to handle the `Manager` aspects. This object should forward all function invocations to its corresponding `BigBankImpl` object, which implements both the `Bank` and the `Manager` functions. It is clear, therefore, that the TIE approach is easier to use when a single object needs to have more than one *unrelated* interface.

Passing Context Information to IDL Operations

A context is a two-dimensional table that maps identifier strings to value strings. A context may be defined in IDL as part of an operation specification. An operation that specifies a context clause is mapped to a C++ member function that takes an extra parameter (just before the `Environment` parameter). For example, the following interface:

```
// IDL
interface A {
    void op(in unsigned long s)
        context ("accuracy", "base");
};
```

maps to:

```
// C++
class A {
public:
    virtual void op(CORBA::ULong s,
        CORBA::Context_ptr IT_c,
        CORBA::Environment& IT_env =
         CORBA::IT_chooseDefaultEnv());
};
```

Instances of `CORBA::Context` are pseudo-objects. A client can create a `Context` as follows:

```
// C++
CORBA::Context_ptr cPtr =
    CORBA::Context::create_context();
```

This creates an initially empty `Context` object, to which *identifier:string* mappings can be added, and that can be passed to a function that takes a `Context` parameter.

On the server side, Orbix constructs a new `Context` from the value received in the incoming operation request and calls the target object's operation. Orbix releases the context when the call returns. If the server requires that the context be retained after the call, you should use `_duplicate()` to increase the reference count of the context argument passed in the call.

You can obtain the default context for a process by calling `get_default_context()`:

```
// C++
// In class CORBA::ORB::get_default_context().
CORBA::Context_ptr defC;
orb->get_default_context(defC);
```

You must free the `Context` allocated in `defC`.

The default context provides a useful mechanism for sharing context changes between different parts of a program. This context is initially empty.

Context Hierarchies

`Context` objects can be nested in context hierarchies by specifying the `parent` parameter when creating a child `Context`, or by using the `create_child()` function. Figure 20 on page 162 illustrates an example context hierarchy.

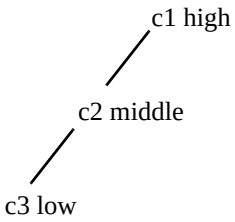


Figure 20: *Hierarchy of Contexts*

A hierarchy may be set up by specifying the parent context in the constructor; a name can also be given to a context:

```
// C++
Context_ptr c1 =
    CORBA::Context::IT_create("high");
Context_ptr c2 =
    CORBA::Context::IT_create("middle", c1);
Context_ptr c3 =
    CORBA::Context::IT_create("low", c2);
```

You must free the `Context` pseudo-object reference returned from the call to `IT_create()`, or alternatively, assign it to a `CORBA::Context_var` variable for automatic management.

CORBA::Context::get_values()

`CORBA::Context` provides a function `get_values()` to retrieve the property values in a `Context`; it is defined as:

```
// C++
// In class CORBA::Context.
CORBA::Status get_values(
    const char* start_scope;
    const Flags op_flags;
    const char* prop_name;
    CORBA::NVList_ptr& values;
    CORBA::Environment& IT_env =
    CORBA::IT_chooseDefaultEnv());
```

You can use the `start_scope` parameter to `get_values()` to specify that the search for the values requested is to be made in a (direct or indirect) parent context of the context on which the call is made. The call searches backwards for a context with the specified name. If this is found, it searches for the specified `Identifier`s in that context.

For example, the following code specifies that the search for identifiers beginning with "sys_" should begin in the context named middle:

```
// C++

CORBA::NVList_ptr listPtr =
            CORBA::NVList::IT_create();
if (!(c3->get_values("middle",
            0, "sys_*", listPtr)))
    // Handle the error.
else {
    // Iterate through the NVList pointed
    // to by listPtr:
}
```

Alternatively, you could code this example to detect an exception raised by get_values() if no entry is found.

If zero is passed as the first parameter to get_values(), the search begins in the context that is the target of the call. If no matching identifiers are found, get_values() returns zero (false).

CORBA::Context::get_values() has a parameter of type Flags. When the null (zero) flag is passed to get_values(), searching of identifiers propagates upwards to parent contexts. If the Flags parameter passed to get_values() is CORBA::CTX_RESTRICT_SCOPE, searching is restricted to the specified start scope or target Context object. Refer to the entry for class CORBA::Flags in the ***Orbix Programmer's Reference C++ Edition*** for more details.

Receiving Diagnostic Messages from Orbix

Orbix enables you to control the output of runtime diagnostic messages on both the client and server. You can set three levels of diagnostics as follows:

Level	Output
0	No diagnostics
1	Simple diagnostics (this is the default)
2	Full diagnostics

Refer to the entry for CORBA::ORB::setDiagnostics() in the ***Orbix Programmer's Reference C++ Edition*** for more details.

Part III

Dynamic Orbix C++ Programming

In this part

This part contains the following:

The TypeCode Data Type

The IDL pseudo-object type TypeCode is used in CORBA to describe arbitrary complex IDL types at runtime. This chapter describes how you can manipulate TypeCode values.

The IDL data type `TypeCode` is used for two main purposes in CORBA systems:

- To describe the contents of an IDL value of type `any`. The `TypeCode` data type forms an important part of the mapping from IDL type `any` to C++ type `CORBA::Any`. This is described in detail in "The Any Data Type" on page 173.
- As a parameter from some of the operations of the Interface Repository. This is described in "The Interface Repository" on page 209.

In an IDL specification, you can use a `TypeCode` as an attribute type or as the type of a parameter or return value to an operation. To make the `TypeCode` data type available, your IDL must include the following directive:

```
#include <orb.idl>
```

The IDL type `TypeCode` maps to a `CORBA::TypeCode_ptr` parameter in the C++ generated from your IDL definitions. The IDL `TypeCode` interface is implemented by the Orbix C++ class `CORBA::TypeCode`.

Overview of the TypeCode Data Type

This section describes the standard IDL interface `CORBA::TypeCode`, as well as the C++ class `CORBA::TypeCode`.

Each `TypeCode` consists of the following:

- A *kind.*

  The kind specifies the overall classification of the `TypeCode`: for example, whether it is a basic type, a struct, a sequence, and so on.
- A sequence of *parameters*.

  The parameters give the details of the type definition and are of type `CORBA::Any`. For example, the IDL type `sequence<long,20>` has the kind `tk_sequence` and has two parameters—the first parameter is a `CORBA::Any` that contains a `TypeCode` for `long`, the second parameter is a `CORBA::Any` that contains the value `20`.

The IDL interface for TypeCode is shown below. Refer to the ***Orbix Programmer's Reference C++ Edition*** for a full description of this interface. It includes an operation kind() to query the kind of a TypeCode and an operation parameter() to access individual parameters of a TypeCode.

```
// IDL
// In module CORBA.
enum TCKind {
    tk_null, tk_void, tk_short, tk_long, tk_ushort,
    tk_ulong, tk_float, tk_double, tk_boolean,
    tk_octet, tk_any, tk_TypeCode, tk_Principal,
    tk_char, tk_objref, tk_struct, tk_union,
    tk_enum, tk_string, tk_sequence, tk_array,
    tk_alias, tk_except, tk_longlong, tk_ulonglong,
    tk_longdouble, tk_wchar,tk_wstring, tk_fixed,
    tk_opaque
}
exception Bounds {};

pseudo interface TypeCode {
    TCKind kind();
    long param_count();
    any parameter(in long index) raises(Bounds);
    boolean equal(in TypeCode tc);
};
```

The C++ signatures of these IDL operations are as follows:

```
// C++
TCKind kind(CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv()) const;
CORBA::Any parameter(CORBA::Long index,
    CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv()) const;
```

The parameter() operation raises the exception Bounds if an attempt is made to access a non-existent parameter. The number of parameters that a TypeCode has varies with the kind of the TypeCode. This number is returned by the param_count() operation of the TypeCode interface. The generated signature of this operation is as follows:

```
// C++
CORBA::Long param_count(CORBA::Environment&
    IT_env = CORBA::IT_chooseDefaultEnv()) const;
```

The parameters of each kind of TypeCode are listed in detail in the entry for CORBA::TypeCode in the ***Orbix Programmer's Reference C++ Edition***. Some examples are as follows:

- A TypeCode of kind tk_struct has one parameter giving the struct name and then two parameters for each member of the struct: the first giving the member's name and the second giving its TypeCode. A struct with N members has 2N+1 parameters. Each parameter is contained in a CORBA::Any.
- A TypeCode of kind tk_string has one parameter—an integer giving the maximum length of the string. A zero length indicates an unbounded string. The parameter is contained in a CORBA::Any.

Implementation of TypeCode in Orbix

The IDL type `TypeCode` is implemented by the C++ class `CORBA::TypeCode`. An IDL operation with a parameter of type `TypeCode` is translated into a C++ member function with a parameter of type `CORBA::TypeCode_ptr`. A declaration for the object that it references can be generated by the IDL compiler from named type definitions that appear in an IDL file—that is, from the following types:

```
interface
typedef
struct
union
enum
```

CORBA::TypeCode_ptr Constants

For each user-defined type that appears in an IDL file, a `CORBA::TypeCode_ptr` can be generated. The `TypeCode_ptr` points to a `TypeCode` constant generated by the IDL compiler. These constants have names of the form `_tc_<type>` where `<type>` is the user-defined type. For example, consider the following IDL specification:

```
interface Interesting {
    typedef long longType;
    struct Useful {
        longType l;
    };
};
```

The following `CORBA::TypeCode_ptr` constants are generated for this definition:

```
_tc_Interesting
_tc_longType
_tc_Useful
```

Note: These definitions are only generated if you specify the `-A` switch to the Orbix IDL compiler.

A number of predefined `CORBA::TypeCode` object reference constants are always available to allow the user to access `TypeCodes` for standard types. Refer to the entry for `CORBA::TypeCode` in the ***Orbix Programmer's Reference C++ Edition*** for a complete list. The following are some examples:

```
CORBA::_tc_float is an object reference for a float TypeCode.
CORBA::_tc_string is an object reference for a string TypeCode.
CORBA::_tc_TypeCode is an object reference for a TypeCode TypeCode.
```

TypeCode Public Members

The C++ class `CORBA::TypeCode` defines the following public members:

- Constructors:

  ```
  CORBA::TypeCode();
  CORBA::TypeCode()(const CORBA::TypeCode&);
  ```

- A destructor.

- operator=(), which allows assignment of objects of type CORBA::TypeCode.
- Function equal(), which allows comparison of objects of type CORBA::TypeCode.
- operator==() and operator!=, which make it easier to compare objects. These operators are specific to the Orbix implementation of TypeCode.
- Function kind(), which returns a value of the enumerate type TCKind.
- Function param_count(), which returns the number of parameters of the CORBA::TypeCode.
- Function parameter(), which returns an individual parameter. This takes the parameter index (the first parameter is at index -1).

CORBA::TypeCode::IT_create()

In addition to the public members listed above, the following function is provided in the public interface to class CORBA::TypeCode:

```
static CORBA::TypeCode_ptr IT_create(
    const TypeCode_ptr& tc,
    CORBA::Environment& IT_env =
    CORBA::IT_chooseDefaultEnv());
```

IT_create() is provided by Orbix to initialize a TypeCode pseudo-object reference, because the CORBA standard does not specify a way to obtain a TypeCode pseudo-object reference. Use of IT_create() is recommended in preference to C++ operator new() in order to ensure memory management consistency.

Examples of Using TypeCode

This section explains the following examples of using TypeCode in Orbix:

- Use of TypeCode in type CORBA::Any.
- Use of TypeCode when querying the Interface Repository.

Use of TypeCode in Type CORBA::Any

Consider an example IDL definition:

```
// IDL
struct Example {
    long l;
};
```

If you compile this definition with the IDL compiler -A switch, the CORBA::TypeCode_ptr constant _tc_Example is generated.

Assume the following IDL operation:

```
// IDL
interface Bar {
    void op(in any a);
};
```

A client program invokes the IDL operation `op()` as follows:

```
// C++
// Client code.
Bar_var bVar;
...
CORBA::Any a ;
// Initialize a. (Not shown in this chapter.)
...
bVar->op(a);
```

On the server-side, you can query the actual type of the parameter to `op()`. For example:

```
// C++
// Server code.
void Bar_i::op(const CORBA::Any& a,
               CORBA::Environment&) {
    CORBA::TypeCode_ptr t = a.type();
    if(t->equal(_tc_Example)) {
        cerr << "Don't like struct Example!"
            << endl;
    }
    else...    // Continue processing here.
}
```

This is one of the most common uses of `TypeCodes`—namely, the runtime querying of type information from a `CORBA::Any`.

Use of TypeCode when Querying the Interface Repository

The Orbix Interface Repository maintains information about IDL type definitions, allowing information about definitions to be determined at runtime. The `kind()` and `parameter()` member functions of class `CORBA::TypeCode` can be used to query the Interface Repository.

For example, when querying information about an operation of an interface, the number of its arguments can be found, and then the `TypeCode` of each argument can be determined. You can use the functions `kind()` and `parameter()` on each `TypeCode` to determine the details of the type of each argument. The chapter "The Interface Repository" on page 209 describes the use of the Interface Repository in detail, including examples of using `TypeCode`.

The Any Data Type

This chapter explains the IDL type any and the corresponding C++ class CORBA::Any, using an example. IDL type any indicates that a value of an arbitrary type can be passed as a parameter or a return value.

This chapter discusses different means of constructing and interpreting an `any`. It first discusses the use of `operator<<=` (left-shift assign operator) and operator `>>=` (right-shift assign operator). This approach is CORBA-defined, and is both the simplest to use and the most type-safe. However, there are situations where these operators cannot be used. This chapter also describes alternative mechanisms for constructing and interpreting an `any`.

Consider the following interface:

```
// IDL

interface AnyDemo {
    void passSomethingIn (in any any_type_parameter);
    ...
}
```

A client can construct an `any` to contain a type that can be specified in IDL, and then pass this in a call to operation `passSomethingIn()`. An application receiving an `any` must determine the type of value stored by the any and then extract the value.

The IDL type `any` maps to the C++ class `CORBA::Any`. Refer to the ***Orbix Programmer's Reference C++ Edition*** for the full specification of this class. This class contains some private member data, accessible via public accessor functions, that store both the type of the `any` and its value. The type is stored as a `CORBA::TypeCode`, and the value is stored as a `void*`.

Inserting Data into an Any with operator<<=()

The C++ class `CORBA::Any` contains a number of left-shift assign operators (`<<=`) that enable you to assign a value to an `any`. An overloaded version of `operator<<=()` is provided for each of the basic IDL types such as `long`, `unsigned long`, `float`, `double`, `string` and so on. In addition, the Orbix IDL compiler can generate such an operator for each user-defined type that appears in an IDL specification.

Note: Operators for user-defined type are generated only if the `-A` switch is passed to the IDL compiler. Refer to "Orbix IDL Compiler Options" on page 319 for more details.

The IDL definition for the example used in this chapter is as follows:

```
// IDL
// In file anydemo.idl

// Illustrates user-defined types and anys.
typedef sequence<long> LongSequence;
...
interface AnyDemo {
    // Takes in any type that can be specified in IDL.
    void passSomethingIn (in any any_type_parameter);

    // Passes out any type specified in IDL.
    void getSomethingOut (out any any_type_parameter);

    // Passes in an any type and passes out an any
    // containing a different type.
    void passSomethingInOut (
        inout any any_type_parameter);
};
```

A version of the code for the example described in this chapter is available in `demos\anydemo` directory of your Orbix installation.

Inserting a Basic Type

Orbix provides a pre-defined overloaded version of `operator<<=()` for basic IDL types such as `long`, `unsigned long`, `float`, `double` `string` and so on.

Assume that a client programmer wishes to pass an `any` containing an IDL `short` (or, in C++, a `CORBA::Short`) as the parameter to the `passSomethingIn()` operation. The client can use the following operator, which is a standard member of the class `CORBA::Any`:

```
void operator<<=(CORBA::Short s);
```

Inserting a User-Defined Type

If the client wishes to pass a more complex user-defined type, such as `LongSequence` (in file `anydemo.idl`), it can use the following generated operators:

```
void operator<<=(CORBA::Any& a,
           const LongSequence& t);
```

Using this operator, you can write the following code:

```
// C++
// In file anydemo_menu.cxx.

// Builds an any containing a sequence of type
// LongSequence and then calls passSomethingIn.
void AnyDemoMenu::do_send_sequence() {
    try {
        CORBA::Any a;

        // Build a sequence of length 2.
        LongSequence sequence_to_insert(2);
        sequence_to_insert.length(2);
```

```
            // Insert a value into the sequence.
            sequence_to_insert[0] = 1;
            sequence_to_insert[1] = 2;

            // Use operator<<=() to insert the sequence
            // into the any.
            a<<=sequence_to_insert;

            // Print out the contents of the sequence.
            cout << "Call passSomethingIn with sequence
                contents:" << sequence_to_insert[0]
                    << " "
                << sequence_to_insert[1] << endl
                    << endl;

            // Now invoke passSomethingIn.
            m_any_demo->passSomethingIn (a);
        }
        catch (const CORBA::SystemException& se) {
        ...
        }
    }
```

These operators provide a type-safe mechanism for inserting data into an `any`. The correct operator is called based on the type of the value being inserted. Furthermore, if an attempt is made to insert a value that has no corresponding IDL type, this results in a compile-time error.

Using the left-shift assign operator to insert a value into an `any` sets both the value of the `CORBA::Any` and the `CORBA::TypeCode` property for the `CORBA::Any`.

Each left-shift assign operator makes a copy of the value being inserted; for example, in the case of object references, `_duplicate()` is used. The `CORBA::Any` is then responsible for the memory management of the copy. Previous values held by the `CORBA::Any` are properly deallocated; for example, using `CORBA::release()` in the case of object references.

Refer to "Other Ways to Construct and Interpret an Any" for details of how to insert boolean, char, array and octet values.

Interpreting an any with operator>>=()

The C++ class `CORBA::Any` contains several right-shift assign operators (`>>=`) that enable you to extract the value stored in an `any`. These operators correspond to the basic IDL types such as `long`, `unsigned long`, `float`, `double`, `string` and so on. As with `operator<<=()`, the IDL compiler can generate an `operator>>=()` for each user-defined type that appears in an IDL specification. These additional operators are only generated if the `-A` switch is specified to the IDL compiler.

Interpreting a Basic Type

The following example illustrates the use of the right-shift assign operators to extract the value stored in an `any`. Each `operator>>=()` returns a `CORBA::Boolean` value to indicate whether or not a value of the required type can be extracted from the `any`. Each `operator>>=()` returns `1` if the `any` contains a value whose `CORBA::TypeCode` matches the type of the right-hand parameter; and returns `0` otherwise. You can extract a value as follows:

```
// C++
// In file anydemo_menu.cxx.

// Shows how an any is passed as an out parameter.
void AnyDemoMenu::do_get_any() {

1       CORBA::Any* any_type_parameter;

        cout << "Call getSomethingOut" << endl;
        m_any_demo->getSomethingOut(any_type_parameter);

        // Assumes that the server passes a string.
        char*  extracted_string = 0;

2       if (*any_type_parameter >>= extracted_string) {
        // Print out the string.
        cout << "any out parameter contains a string with
        value :" << extracted_string << endl << endl;
        }
        else {
        // Error message.
        cout << "unexpected value contained in any"
        << endl;
        }
}
```

This code is explained as follows:

1. The `CORBA::Any` variable retains ownership of the memory it returns when `operator>>=()` is called. Because the memory is managed by the `CORBA::Any` type there is no need for you to manage the memory.
2. The function `operator>>=()` is used to interpret the contents of the `any` parameter. If successful, the operator causes the extracted pointer to point to the memory storage managed by the `any`.

Interpreting a User-Defined Type

More complex, user-defined types can also be extracted using the right-shift operators generated by the IDL compiler. For example, the `LongSequence` IDL type from "Inserting a User-Defined Type":

```
// IDL
typedef sequence<long> LongSequence;
```

You can extract a `LongSequence` from a `CORBA::Any` as follows:

```
void AnyDemoMenu::do_get_any() {
    CORBA::Any* any_type_parameter;

    cout << "Call getSomethingOut" << endl;

    m_any_demo->getSomethingOut(any_type_parameter);

    LongSequence*  extracted_sequence = 0

    if (*any_type_parameter>>= extracted_sequence) {
        cout << "any out parameter contains a sequence
        with value :" << extracted_sequence << endl
        << endl;
    }
    else {
        cout << "unexpected value contained in any"
        << endl;
    }
}
```

The generated right-shift operator for user-defined types takes a pointer to the generated type as the right-hand parameter. If the call to the operator is successful, this pointer points to the memory managed by the `CORBA::Any`.

No attempt should be made to delete or otherwise free the memory managed by the `CORBA::Any`. Extraction into a `_var` variable violates this rule, because the `_var` variable attempts to assume ownership of the memory. Furthermore, it is an error to attempt to access the storage associated with a `CORBA::Any` after the `CORBA::Any` variable has been deallocated.

Other Ways to Construct and Interpret an Any

This section presents a number of other ways to construct and interpret an `any`. You should use the `>>=` and `<<=` operators wherever possible, but there are occasions when you must use a more complex approach.

Inserting Values at Construction Time

Instead of creating a `CORBA::Any` variable using the default constructor, and then inserting a value using `operator<<=()`, an application can specify the value and its type when the `CORBA::Any` is being constructed. This alternative constructor has the following signature:

```
// C++
Any(CORBA::TypeCode_ptr tc, void* value,
    CORBA::Boolean release = 0);
```

This is not used normally, because it is more difficult to use than `operator<<=()`, and because it is not type-safe. Specifically, the type of the value passed to the `value` parameter may not match the type passed in parameter `tc`. A mismatch is not detected because the `value` parameter is of type `void*` and this leads to subsequent errors.

However, there are some types that must be inserted in this way, for example bounded strings. Both bounded and unbounded IDL `string`s map to `char*` in C++, and hence both cannot be inserted using `operator<<=()`. This operator is used to insert unbounded strings only. A `CORBA::Any` containing a bounded string must be created using a specific constructor. You can use the function `CORBA::Any::replace()` to make assignments. Refer to "Low Level Access to a CORBA::Any" on page 179 for more details.

For example, you can construct a `CORBA::Any` variable to contain a bounded string as follows:

```
// C++
// In file anydemo_menu.cxx.

// Insert a bounded string into an any using the
// constructor.
void AnyDemoMenu::do_send_bounded_string() {
    try {
        // Allocate the correct memory for the string.
1       char* string_to_insert =
        CORBA::string_alloc(string_length);
        strcpy(string_to_insert,"Making Software
                    Work Together (TM)");

        // Call to constructor.
2           CORBA::Any a(_tc_BoundedString,&
                string_to_insert, 1);

        // Invoke passSomethingIn as normal.
        cout << "Call passSomethingIn with string
                value :"
            << string_to_insert << endl << endl;
        m_any_demo->passSomethingIn (a);
    }
    catch (const CORBA::SystemException& se) {
        cerr << "System exception: Call
                passSomethingIn
        with a string failed" << endl;
        cerr <<& sysEx;
    }
    catch (const CORBA::Exception& se) {
        cerr << "Exception: Call passSomethingIn
         with a string failed" << endl;
        cerr <<& sysEx;
    }
    catch (...) {
        cerr << "Unexpected exception: Call
        passSomethingIn with a string failed"
                << endl;
    }
}
```

This code is explained as follows:

1. Because this example uses a bounded string, you must ensure that the string is allocated the appropriate amount of memory. The constant `string_length` is defined in `anydemo.idl`.
2. The first parameter to the `CORBA::Any` constructor is a pseudo-object reference for a `CORBA::TypeCode`. In this case, the constant `_tc_BoundedString` is passed. This constant is generated by the IDL compiler.

   The second parameter is a pointer to the value to be inserted into the `CORBA::Any`; in this case `string_to_insert`. This value should be of the type specified by the `CORBA::TypeCode_ptr` parameter. The behavior is undefined if the `CORBA::TypeCode_ptr` and the `value` parameters do not agree. When constructing `CORBA::Any`s for `string` types, the second parameter is of type `char**`.

   The third parameter, `release`, specifies which code assumes ownership of the memory occupied by the value in the `CORBA::Any` variable (`string_to_insert`). If this is `1` (true), the `CORBA::Any` assumes ownership of the storage pointed to by the `value` parameter. If this parameter is `0` (false), the caller must manage the memory associated with the `value`. The default is zero.

   In this example, the `CORBA::Any` assumes ownership of the memory associated with the variable `string_to_insert`: the application code is not required to free this memory.

Low Level Access to a CORBA::Any

Class `CORBA::Any` provides three type-unsafe functions enabling low level access to an `Any`. These are defined as follows:

```
// C++
void replace(CORBA::TypeCode_ptr, void* value,
             CORBA::Boolean release = 0);

CORBA::TypeCode_ptr type() const;

const void* value() const;
```

replace()

The `replace()` function is only intended for use with types that cannot use the type-safe operator interface. It can be used at any time after construction of a `CORBA::Any` to replace the existing `CORBA::TypeCode` and value. Like the various `<<=` operators, it releases the previous `CORBA::TypeCode` and if necessary, deallocates the storage previously associated with the `value`. The `release` parameter has the same semantics as the `release` parameter of the `CORBA::Any` constructor described in "Inserting Values at Construction Time" on page 177.

type()

The `type()` function returns an object reference for a `CORBA::TypeCode` that describes the type of the `CORBA::Any`. As with all object references, the caller must release the reference when it is no longer needed, or assign it to a `CORBA::TypeCode_var` variable for automatic management.

value()

The `value()` function returns a pointer to the data stored in the `CORBA::Any`, or, if no value is stored, it returns the null pointer. This value may be cast to the appropriate C++ type depending on the `CORBA::TypeCode` of the `CORBA::Any`. The rules for the actual C++ type returned for each different IDL type are listed in the entry for `CORBA::Any` in the ***Orbix Programmer's Reference C++ Edition***.

If the `CORBA::Any` contains an object reference for an object whose type is unknown at compile time, the `type()` function returns a reference for a `CORBA::TypeCode` object that is equal to the `_tc_object` typecode constant. The `value()` function returns a `void*` that can be cast to a `CORBA::Object_ptr*`.

Example of Using type() and value()

The following example determines the type of an any by comparing the contents of the any with the typecode constant for a bounded string:

```
// C++
// In file anydemo_impl.cxx.

void AnyDemoImpl::passSomethingIn(
    const CORBA::Any& any_type_parameter,
    CORBA::Environment& )
    throw(CORBA::SystemException) {
    ...

    CORBA::TypeCode_ptr type= any_type_parameter.type();

    // Checks if the any contains a bounded string.
    if (type->equal(_tc_BoundedString)) {

    // Returns a void pointer to the bounded string.
        char** any_contents =
            (char**)any_type_parameter.value();
        const char* bounded_string = *any_contents;

        // Print out the contents.
        cout << "passSomethingIn extracted a bounded
            string of length " << strlen(bounded_string)
            << " and value " << bounded_string << endl
            << endl;
    }
    else {
        // Error message.
        cout << "passSomethingIn: unexpected value"
        << endl;
    }
}
```

Orbix defines a typecode constant for each built-in type, and you can instruct the IDL compiler to generate typecode constants for each user-defined type. This is discussed in more detail in "The TypeCode Data Type" on page 167.

Refer to the ***Orbix Programmer's Reference C++ Edition*** for more details on the `replace()`, `type()` and `value()` functions.

Inserting and Extracting Array Types

Recall that IDL arrays are mapped to regular C++ arrays. This presents a problem for the type-safe operator interface to CORBA::Any. C++ array parameters decompose to a pointer to their first element, so you cannot use the operators to insert or extract arrays of different lengths.

Nevertheless, arrays can be inserted and extracted using the operators, because a distinct C++ type is generated for each IDL array—specifically to help with insertion and extraction into or out of CORBA::Any variables. The name of this type is the name of the array followed by the suffix "_forany".

The following example shows type-safe manipulation of arrays and CORBA::Anys:

```
// IDL
typedef long longArray[2][2];

// C++
longArray_forany m_array = { {14, 15}, {24, 25} };

// Insertion:
CORBA::Any a;
if (a <<= m_array) {
    cout << "Success!" << endl;
}

// Extraction:
longArray_forany extractedValue;
if (a >>= extractedValue) {
    cout << "Element [1][2] is "
            << extractedValue[1][2] << endl;
}
```

These types, like the array _var types, provide an operator[]() to access the array members, but the _forany types do not delete any storage associated with the array when they are themselves destroyed. This is a good match for the semantics of operator>>=(). The CORBA::Any retains ownership of the memory returned by the operator. There is therefore no memory leak in this code sample.

Inserting and Extracting boolean, octet and char

The standard CORBA IDL to C++ mapping does not require that the IDL types boolean, octet and char map to distinct C++ types. Therefore, it is not possible to insert and extract each of these using operator<<=() and operator>>=(). Remember that the overloaded right-shift and left-shift assignment operators are distinguished based on the type of the right-hand argument.

In Orbix, the types boolean and octet map to the same underlying C++ type (unsigned char). Type char maps to a different type (C++ char), so a separate operator could have been provided for it, but this would not be CORBA compliant.

The distinction is achieved by using helper types that are nested within the C++ class CORBA::Any. These helper types are structs; refer to the entry for CORBA::Any in the ***Orbix Programmer's***

Reference C++ Edition for details on their syntax. Left-shift and right-shift assignment operators are provided for each of these helper types.

These helper classes can be used as follows:

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

// Insert a boolean into the CORBA::Any a:
CORBA::Boolean b = 1;
a<<=CORBA::Any::from_boolean(b);

// Extract the boolean.
CORBA::Boolean extractedValue;
if (a>>=CORBA::Any::to_boolean(extractedValue)){
    cout<<"Success!"<<endl;
}
// Insert an octet into the CORBA::Any a:
CORBA::Octet o = 1;
a<<=CORBA::Any::from_octet(o);

// Extract the octet from a:
CORBA::Octet extractedValue;
if (a>>=CORBA::Any::to_octet(extractedValue)) {
    cout<<"Success!"<<endl;
}

// Insert a char into the CORBA::Any a:
CORBA::Char c='b';
a<<=CORBA::Any::from_char(c);

// Extract the char from a:
CORBA::Char extractedValue;
if (a>>=CORBA::Any::to_char(extractedValue)) {
    cout<<"Success!"<<endl;
}
```

Any Constructors, Destructor and Assignment

In addition to the functionality already described, the C++ class `CORBA::Any` also contains the following:

- A default constructor.

  This creates a `CORBA::Any` with a `CORBA::TypeCode` of kind `tk_null` and no value.
- A copy constructor.

  This calls `_duplicate()` on the `CORBA::TypeCode_ptr` of its `CORBA::Any` parameter and deep copies the parameter's value.
- A constructor for setting the type and value of an `CORBA::Any` for untyped values.

  This is described in "Inserting Values at Construction Time" on page 177.

- An assignment operator.

  This releases its own `CORBA::TypeCode_ptr` and deallocates the memory associated with its current value, if any. It then duplicates the `CORBA::TypeCode_ptr` of its `CORBA::Any` parameter and deep copies the parameter's value.
- A destructor.

  This calls `CORBA::release()` on the `CORBA::TypeCode_ptr` and deallocates the memory associated with the value, if any.

Any as a Parameter or Return Value

The mappings for IDL `any` operation parameters and return value are illustrated by the following IDL operation:

```
// IDL
any op(in any a1, out any a2, inout any a3);
```

This maps to:

```
// C++
CORBA::Any* op(const CORBA::Any& a1,
       CORBA::Any*& a2, CORBA::Any& a3);
```

Because both return values and `out` parameters map to pointers to `CORBA::Any`, a `CORBA::Any_var` class is provided that manages the memory associated with this pointer. The `CORBA::Any_var` class calls the C++ operator `delete` on its associated `CORBA::Any*` when it is itself destroyed; for example, by going out of scope.

Dynamic Invocation Interface

In a normal Orbix client program, the IDL interfaces that the client can access are determined when the client is compiled. The Dynamic Invocation Interface (DII) allows a client to call operations on IDL interfaces that were unknown when the client was compiled.

IDL is used to describe interfaces to CORBA objects, and the Orbix IDL compiler generates the necessary support to enable clients to make calls to remote objects. Specifically, the IDL compiler automatically builds the appropriate code to manage proxies, to dispatch incoming requests within a server, and to manage the underlying Orbix services.

Using this approach, the IDL interfaces that a client program can use are determined when the client program is compiled. Unfortunately, this is too limiting for a small but important subset of applications. These application programs and tools require that they can use an indeterminate range of interfaces: interfaces that perhaps were not even conceived at the time the applications were developed. Examples include browsers, gateways, management support tools and distributed debuggers.

Orbix supports the CORBA *Dynamic Invocation Interface* (DII), this allows an application to issue requests for any interface, even if that interface was unknown when the application was compiled.

The DII allows invocations to be constructed at runtime by specifying the target object reference, the operation or attribute name and the parameters to be passed. A server receiving an incoming invocation request does not know whether the client that sent the request used the normal, static approach or the dynamic approach to compose the request.

Using the DII

This chapter uses a banking example to demonstrate the use of the DII. This example has the following IDL definitions:

```
// IDL
interface Account {
    readonly attribute float balance;

    void makeDeposit(in float f);
    void makeWithdrawal(in float f);
};

interface Bank {
    exception Reject {
        string reason;
    };

    // Create an account.
    Account newAccount(in string owner,
        inout float initialBalance) raises (Reject);
```

```
        // Delete an account.
        void deleteAccount(in Account a);
    };
```

To help illustrate the use of the DII the operation `Bank::newAccount()` has been extended to take an `inout` parameter denoting the initial balance.

The examples that follow show how you can make dynamic invocations by constructing a `Request` object and then causing the specified operation or attribute call to be made. The examples make the equivalent of the following call to operation `newAccount()`:

```
// C++
Bank_var bankVar = ...
CORBA::Float initialBalance = 1000.00;
bankVar->newAccount("Chris", initialBalance);
```

Programming Steps in Using the DII

To make an invocation using the DII, do the following:

1. Get a reference to the target object.
2. Construct a `Request` object.
3. Populate the `Request` object with information about the invocation, including the object reference, the name of the operation or attribute to be called, and the parameters to the operation.
4. Invoke the request.
5. Retrieve the results of the operation.

There are two ways to use the DII:

- Using *CORBA-defined functions*
- Using the *Orbix stream-like interface*

The Orbix stream-like interface to the DII is easier to use than the CORBA-defined functions, but this interface is not CORBA-compliant.

There are two common types of client program that use the DII:

- A client interacts with the Interface Repository to determine a target object's interface, including the name and parameters of one or all of its operations and then uses this information to construct DII requests.
- A client, such as a gateway, receives the details of a request to be made. In the case of a gateway, this may arrive as part of a network package. The gateway can then translate this into a DII call, without checking the details with the Interface Repository. If there is any mismatch, the gateway will receive an exception from Orbix, and can report an error to the caller.

Some client programs also use the DII to call an operation with *deferred synchronous* semantics, which is not possible using normal static operation calls. Deferred synchronous calls are described in "Invoking a Deferred Synchronous Request" on page 192.

The CORBA Approach to Using the DII

The first step in using the DII is to obtain a reference to the target object for the request. You can do this using any of the standard methods described in "Making Objects Available in Orbix".

Note: All IDL interfaces inherit from type `CORBA::Object`, so every object reference can be represented using type `CORBA::Object_ptr`. Some client programs may use a user-defined object reference type, but most clients that use the DII use the most-general type, `CORBA::Object_ptr`.

The remainder of this section describes how you create and invoke a request with the CORBA-compliant approach to using the DII.

Setting up a Request

There are two CORBA-compliant ways to construct a `Request` object:

1. Using the function `_request()` defined in class `CORBA::Object`. This is declared as:

```
// C++
typedef char* Identifier;

CORBA::Request_ptr _request(Identifier
     operation, CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
```

   A program that uses the function `_request()` must be linked with the Interface Repository client library, as described in "The Interface Repository".

2. Using the function `_create_request()` also defined in class `CORBA::Object`. This is declared as:

```
// C++
CORBA::Status _create_request(
                CORBA::Context_ptr ctx,
                const char* operation,
                CORBA::NVList_ptr arg_list,
                CORBA::NamedValue_ptr& result,
                CORBA::Request_ptr& request,
                CORBA::Flags req_flags,
                CORBA::Environment& IT_env =
                CORBA::IT_chooseDefaultEnv());
```

Setting up a Request Using _request()

You can set up a request by invoking `_request()` on the target object and specifying the name of the operation that is to be invoked. You can then populate the `Request` object with the parameters to the call.

Creating the Request Object

To create a Request object, first obtain an object reference to the target object. Call _request() on the target object as follows:

```
// C++
// Get object reference.
CORBA::Object_var target =
       CORBA::Orbix::string_to_object(refStr);

// Create a Request object
// for operation newAccount().
CORBA::Request_var request =
       target->_request("newAccount");
```

The function _request() takes the name of the target operation or attribute as a parameter. If you wish to call a *get* or *set* function for an attribute, then prefix the attribute name with _get_ or _set_ as required.

Adding the Parameters to the Request Object

There are two steps in adding the parameters to the Request object:

1. Call the function CORBA::Request::arguments() to get an empty list of name-value pairs corresponding to the parameters of the operation to be called. This list is of type CORBA::NVList, which is a list of CORBA::NamedValue objects.
2. Add a CORBA::NamedValue object to the list for each operation parameter value. The CORBA::NamedValue object stores the name of the parameter and the corresponding value, represented as type CORBA::Any.

You can get the empty parameter list for a request and create a CORBA::NamedValue object for each parameter as follows:

```
// C++
CORBA::NamedValue_ptr ownerArg;
CORBA::NamedValue_ptr balanceArg;

    ownerArg = req->arguments()->add(CORBA::ARG_IN);
    balanceArg = req->arguments()->add(CORBA::ARG_INOUT);
```

The function CORBA::NVList::add() creates a CORBA::NamedValue and adds it to the operation parameter list. It returns a CORBA::NamedValue_ptr for the newly created object. This object does not yet contain the required parameter value.

You must specify the parameter passing mode when creating each of the CORBA::NamedValue objects. Specify these modes in the order in which the parameters appear in the IDL definition for the operation.

The parameter passing modes are as follows:

CORBA::ARG_IN	Input parameters (IDL in).
CORBA::ARG_OUT	Output parameters (IDL out).
CORBA::ARG_INOUT	Input/output parameters (IDL inout).

To set the parameter values, get a pointer to the CORBA::Any in each CORBA::NamedValue object in the parameter list and update it with the corresponding value. To get the CORBA::Any value, use the CORBA::NamedValue::value() function.

For example, to update the first parameter to the operation `newAccount()` do the following:

```
CORBA::Any* ownerValue = ownerArg->value();

// Insert the parameter value:
*ownerValue <<= "Chris";
```

To update the second parameter do the following:

```
CORBA::Any* balanceValue = balanceArg->value();

// Insert the parameter value:
*balanceValue <<= 1000.00;
```

At this point, the request has been constructed and is ready to be invoked.

Adding a Context Parameter to the Request

If the IDL operation has an associated IDL `context` clause, then you can add a `Context` object can to the request. To do this, use the operation `ctx()` defined on class `Request`. This function is described in the entry for class `Request` in the ***Orbix Programmer's Reference C++ Edition***.

Setting up a Request Using _create_request()

Another way to set up a request is to first create a list object, of type `CORBA::NVList`, containing the values of the operation parameters and then invoke `_create_request()` on the target object, passing the request details to this function.

Creating a List of Parameter Values

There are two steps in creating a list of parameter values:

1. Create an empty list of name-value pairs to contain the parameters. This list is of type `CORBA::NVList`, which is a list of `CORBA::NamedValue` objects.
2. Add a `CORBA::NamedValue` object to the list for each operation parameter value. The `CORBA::NamedValue` object stores the name of the parameter and the corresponding value, represented as type `CORBA::Any`.

Create a `CORBA::NVList` and prepare the list to hold the parameter values as follows:

```
// C++
CORBA::NVList_ptr argList;
CORBA::NamedValue_ptr ownerArg;
CORBA::NamedValue_ptr balanceArg;

if (CORBA::Orbix.create_list(2, arglist) {
    ownerArg = argList->add(CORBA::ARG_IN);
    balanceArg = argList->add(CORBA::ARG_INOUT);
}
```

The function `CORBA::NVList::add()` is described in "Adding the Parameters to the Request Object" on page 188.

The `CORBA::NVList` object assumes ownership of the memory for each `CORBA::NamedValue` object in the list. You should not release the `CORBA::NamedValue_ptr` returned from `CORBA::NVList::add()` and you should not assign the result to an `_var` variable.

To set the parameter values, insert each value into the CORBA::Any associated with the corresponding CORBA::NamedValue object, as described in "Adding the Parameters to the Request Object" on page 188:

```
CORBA::Any* ownerValue = ownerArg->value();
CORBA::Any*  balanceValue = balanceArg->value();

// Insert the parameter values.
*ownerValue <<= "Chris";
*balanceValue <<= 1000.00;
```

Creating the Request Object

The function _create_request() is defined in class CORBA::Object as follows:

```
// C++
CORBA::Status _create_request(
            CORBA::Context_ptr ctx,
            const char* operation,
            CORBA::NVList_ptr arg_list,
            CORBA::NamedValue_ptr& result,
            CORBA::Request_ptr& request,
            CORBA::Flags req_flags,
            CORBA::Environment& IT_env =
                CORBA::IT_chooseDefaultEnv());
```

The parameters to this function are as follows:

ctx	A pointer to the Context object to be sent in the request, if the operation has an associated IDL context clause.
operation	The name of the operation to be called. If you wish to call a *get* or *set* function for an attribute, specify the name of the attribute preceded by the string *_get_* or *_set_*.
arg_list	The parameters to the operation.
result	The location for the return value.
request	The pointer to the new Request object to be created.
req_flags	The flags for the request.
env	The Environment parameter for exception handling.

The return type Status is a typedef for CORBA::ULong. Function _create_request() returns a non-zero value to indicate success and a zero value to indicate failure.

When calling _create_request(), you initialize parameters ctx, operation, arg_list, and req_flags in advance. You do not need to initialize parameters result or request.

Once you call _create_request(), you must specify the TypeCode of the operation return value. To do this, call CORBA::Request::set_return_type() on the Request object, passing the TypeCode constant associated with the return type.

The example shown below constructs a `Request` for operation `newAccount()`:

```
// C++
CORBA::Request_ptr request;
CORBA::NVList_ptr argList;
CORBA::NamedValue_ptr result;

// Add parameter values to argList.
...

// Construct the Request object.
if(target->_create_request(
    CORBA::Context::_nil(), "newAccount", argList,
    result, request, 0)) {
        request->set_return_type(_tc_Account);

        // Invoke the request.
        ...
}
```

Using the Interface Repository when Setting Up a Request

Both CORBA-compliant methods of setting up a `Request` object require that you create a `CORBA::NVList` object containing the values of the operation parameters. If you have obtained a description of an operation from the Orbix Interface Repository, as described in "The Interface Repository", an alternative way to create the `CORBA::NVList` object is available.

An operation is described in the Interface Repository by an object of type `CORBA::OperationDef`. The function `CORBA::ORB::create_operation_list()` is defined as follows:

```
// C++
CORBA::Status create_operation_list(
    CORBA::OperationDef_ptr operation,
    CORBA::NVList_ptr& new_list,
    CORBA::Environment& IT_env =
                CORBA::IT_chooseDefaultEnv());
```

Call this function on the `CORBA::Orbix` object, passing a `CORBA::OperationDef` object that describes the target operation and an empty `CORBA::NVList` object. This function updates the `CORBA::NVList` object with one element for each argument. Each element is initialized with the correct parameter passing mode, the name of the argument, and an initial value of type `CORBA::Any`. The value of the `CORBA::Any` is not initialized.

To call `CORBA::ORB::create_operation_list()`, a client must be linked against the Interface Repository client library, as described in "The Interface Repository".

Invoking a Request

Once the parameters are inserted, you can invoke a request as follows:

```
// C++
try {
    if (request->invoke())
        // Call to invoke() succeeded.
    else
        // Call to invoke() failed.
}
catch (const CORBA::SystemException& se) {
    cout << "Unexpected Exception" <<& se << endl;
}
```

Exceptions are handled in the same manner as for static function invocations. However, user-defined exceptions are not currently supported.

Invoking a Request for a Oneway Operation

The function `CORBA::Request::invoke()` calls the target operation and blocks the client until the operation returns. You can not use invoke to call a oneway operation. Instead, you must use the function `CORBA::Request::send_oneway()`.

For example, if the `Request` object `request` was set up for a oneway operation call, then you could invoke `send_oneway()` as follows:

```
// C++
try {
    request->send_oneway();
}
catch (const CORBA::SystemException& se) {
    cout << "Unexpected Exception" <<& se << endl; }
```

Note: You can also use `send_oneway()` to invoke a normal, non-`oneway`, operation. The effect of this is that the client is not blocked while the operation call is being processed, but all return values, `out`, and `inout` parameters are discarded. This functionality is rarely required.

Invoking a Deferred Synchronous Request

The DII allows you to make operation calls using *deferred synchronous* semantics. Using these semantics, a client can call an operation, continue processing in parallel with the operation, and then retrieve the operation results when required.

To use this method of invoking a request, do the following:

1. Invoke the request by calling `CORBA::Request::send_deferred()`.
2. Continue processing in parallel with the operation.
3. If you wish to check if the result of the operation is available, call the function `CORBA::Request::poll_response()` on the `Request` object. This function returns a non-zero value if a response has been received.
4. To get the result of the operation, call `CORBA::Request::get_response()` on the `Request` object.

For more details on the functions CORBA::Request::send_deferred(), CORBA::Request::poll_response(), and CORBA::Request::get_response(), see the entry for class CORBA::Request in the ***Orbix Programmer's Reference C++ Edition***.

Invoking Multiple Requests Simultaneously

Two functions defined on class CORBA::ORB allow you to invoke multiple DII requests simultaneously. To call multiple oneway operations simultaneously, invoke the function CORBA::ORB::send_multiple_requests_oneway() on the CORBA::Orbix object. To call multiple deferred synchronous operations, call CORBA::ORB::send_multiple_requests_deferred() on the same object. These functions are described in the entry for class CORBA::ORB in the ***Orbix Programmer's Reference C++ Edition***.

Retrieving the Results of a Request

When you invoke a request, the values of the out and inout parameters are automatically modified within the CORBA::NVList that contains the parameter values. The function CORBA::Request::arguments() returns this list. To get the parameter values, do the following:

1. Call arguments() on the Request object to get the parameter list. This function returns a CORBA::NVList_ptr.
2. Use the function CORBA::NVList::item() to return an element at a particular index in the list and get the CORBA::NamedValue objects associated with the out and inout parameters.
3. Call CORBA::NamedValue::value() to get a pointer to the CORBA::Any value for each parameter.
4. Extract the parameter values from the CORBA::Any.

The function CORBA::NVList::item() is described in the entry for class CORBA::NVList in the ***Orbix Programmer's Reference C++ Edition***.

To get the return value of the operation, call the function result() on the Request object. This function is defined in class CORBA::Request as follows:

```
// C++
CORBA::NamedValue_ptr result(CORBA::Environment&
        IT_env = CORBA::IT_chooseDefaultEnv());
```

This function returns a reference to a CORBA::NamedValue. Before calling this function, you must create the CORBA::NamedValue object as follows:

```
CORBA::NamedValue_ptr nv =
        CORBA::NamedValue::IT_create();
```

Use the value() function defined on CORBA::NamedValue to extract the CORBA::Any containing the return value of the operation, as for out and inout parameters.

Getting Information About a Request Object

Given a `Request` object, you can get the operation name and the target object reference using the functions `CORBA::Request::operation()` and `CORBA::Request::target()`, respectively. "Filtering Operation Calls" provides an example in which these functions are required.

The Orbix-Specific Approach to Using the DII

As in the CORBA-compliant approach to using the DII, the first step in using the Orbix-specific approach is to obtain a reference to the target object for the request. You can do this using any of the standard methods described in "Making Objects Available in Orbix".

The remainder of this section describes how you create and invoke a request with the Orbix stream-like interface to the DII.

Setting Up a Request

Orbix allows you to instantiate a `Request` object using the normal C++ mechanisms. For example, you can create a `Request` object as follows:

```
// C++
CORBA::Object_ptr target;
// Get a reference for the target object.
...

CORBA::Request request(target);
```

The `Request` constructor used in this example takes the target object reference as a parameter.

The next step is to set the target operation name. To do this, call the function `CORBA::Request::setOperation()` on the `Request` object, for example:

```
// C++
request.setOperation("newAccount");
```

Once you set the operation name, you must specify the `TypeCode` of the operation return value. To do this, call `CORBA::Request::set_return_type()` on the `Request` object, passing the `TypeCode` constant associated with the return type. For example, to set the return type to `Account`, call `CORBA::Request::set_return_type()` as follows:

```
//C++
request.set_return_type(_tc_Account);
```

You can then insert the values of the operation parameters into the request. Orbix allows you to do this as if `Request` object were an I/O stream. Class `CORBA::Request` supports `operator<<()` for all of the IDL basic types, except `octet`.

For example, to insert the parameters for operation `newAccount()`, do the following:

```
// C++
CORBA::Float initialBalance = 1000.00

request << "Chris";
request << CORBA::inoutMode << initialBalance;
```

The parameters must be inserted in the correct order. Orbix dynamically type-checks the values when the request arrives at the remote object.

The default parameter passing mode is `in`. You can specify the parameter passing mode using one of the following manipulators:

`CORBA::inMode`	Input parameters (IDL `in`).
`CORBA::outMode`	Output parameters (IDL `out`).
`CORBA::inoutMode`	Input/output parameters (IDL `inout`).

Using a manipulator changes the parameter attribute mode *for all subsequent parameters* for this `Request` object or until another manipulator is used.

Adding a Context Parameter to the Request

You can also use `operator<<()` to specify a `Context` object to be passed in a request. Use this operator to pass the `Context` object as the last parameter to the request, as if the `Context` object were an IDL `in` parameter.

Invoking a Request

Once you insert the operation parameters, you can invoke the request as described in "Invoking a Request" on page 192. For example, the most common way to invoke a request is to call `CORBA::Request::invoke()` as follows:

```
// C++
try {
    if (request->invoke())
        // Call to invoke() succeeded.
    else
        // Call to invoke() failed.
}
catch (const CORBA::SystemException& se) {
    cout << "Unexpected Exception" <<& se << endl;
}
```

Resetting a Request Object

If you wish to invoke several DII requests in a single program, you can use several `Request` variables, using the appropriate operation settings for each. Alternatively, you can use a single `Request` variable and reset this variable for each request.

To reset an existing `Request` object, call `CORBA::Request::reset()`. You can then set new values for the target object, for example as follows:

```
// C++
request.reset();
request.setTarget(aPtr);
request.setOperation("makeDeposit");
```

You can also do this as follows:

```
// C++
request.reset(aPtr, "makeDeposit");
```

You can then insert new operation parameters into the request. You should also set the request return type, as described in "Creating the Request Object" on page 190.

Retrieving the Results of a Request

When the operation returns, you can examine the return value and output parameters. If there are any `out` and `inout` parameters, these are modified by the call and no special action is required to access their values. For example, after calling `invoke()` on a request to operation `newAccount()`, the actual parameter `initialBalance` is updated automatically.

To get the operation return value, use the extraction operator, `operator>>()`, as follows:

```
// C++
Account_ptr aPtr;
CORBA::Object_ptr oPtr;

try {
    // Call newAccount() using Request request.
    ...

    // Extract the return value.
    request >> oPtr;
    if (aPtr = Account::_narrow(oPtr)) {
        // Use the returned Account object reference.
        ...
    }
}
catch (const CORBA::SystemException& se) {
    cout << "Unexpected System Exception"
        << se << endl;
}
catch (...) {
    cout << "Unexpected exception << endl;
    ...
}
```

Note: `operator>>()` is used to extract just the return value from the request and *not* to extract the output parameters.

Additional Information About operator<<()

As a further example of operator<<(), consider the following IDL operation:

```
// IDL
long op(in long i, inout float f, out char c);
```

You can insert the parameters for this operation as follows:

```
// C++
CORBA::Request request;
CORBA::Long i = 4L;
CORBA::Float f1 = 8.9;
CORBA::Char ch;

request << i
        << CORBA::inoutMode << f1
        << CORBA::outMode << ch;
```

Note that parameters to operator<<() are passed by reference, so you must write:

```
<< f1
```

rather than:

```
<< &f1
```

Input (in) parameters are *not* copied into the request argument list; so if the values of the variables are changed between their insertion and invocation, the new values are transmitted. In other words, operator<<() uses "call by reference" semantics. Care must be taken to ensure that the parameters remain in existence and have the desired values when the invocation of the Request is actually made. An example of such an error would be to insert a local variable within a function and to return from the function before the Request invocation is made.

Parameters inserted using operator<<() are, by default, nameless. However, you can explicitly give the parameter a name, using CORBA::arg():

```
// C++
// Insert parameter "height".
request << CORBA::arg("height") << 65;
```

The naming of parameters does not remove the requirement that parameters must be inserted in the proper order. However, if the same parameter name is used again, its previous value is replaced with the new value.

Note: arg affects only a single use of operator<<(). The manipulators inMode, outMode, and inoutMode affect *all subsequent* uses of operator<<() on a given Request object until the next mode change.

Inserting and Extracting Octets

An octet *cannot* be inserted into or extracted from a Request using operators << and >>.

This restriction arises because both IDL octet and boolean map to the same underlying C++ type. Since the type boolean is used more frequently than octet, operator<<(unsigned char) and operator>>(unsigned char) assume that their parameter is a

boolean; and this assumption may lead to conversion errors between heterogeneous machines if the parameter is in fact an octet.

To insert an octet into a Request, use the function CORBA::Request::insertOctet():

```
// C++
CORBA::octet o = 0xA2;
request.insertOctet(o);
```

Use the function CORBA::Request::extractOctet(octet&) to extract an octet return value.

Inserting and Extracting User-Defined Types

Two manipulators, CORBA::insert and CORBA::extract, allow you to insert and extract user-defined IDL types into and out of a Request object.

The use of these manipulators for structs is illustrated in the code segment below:

```
// IDL
struct Example {
    long m1;
    char m2;
};

// C++
CORBA::Request request;
Example e;
e.m1 = 27;
e.m2 = 'c';
request << CORBA::insert(
        _tc_Example, &e, CORBA::inMode);
```

CORBA::insert uses the CORBA::TypeCode constant generated by the IDL compiler for each user-defined type. In this case, _tc_Example is the TypeCode for the IDL struct Example. Refer to "The TypeCode Data Type" for a full explanation of TypeCodes.

User-defined IDL types can be extracted from a Request using the CORBA::extract manipulator:

```
// C++
CORBA::Request request;
st s1;
request >> CORBA::extract(_tc_Example, &s1);
```

The CORBA::insert and CORBA::extract manipulators also work for primitive types.

Inserting and Extracting Arrays

To insert an array of basic types into a Request, one of the following functions should be called on the Request object:

encodeCharArray()	encodeOctetArray()
encodeShortArray()	encodeUShortArray()
encodeLongArray()	encodeULongArray()
encodeFloatArray()	encodeBooleanArray()

Each is defined in class CORBA::Request and takes a pointer to the first element of the array, and the array length (as a CORBA::ULong).

To extract an array, one of the following functions should be called on the `Request` object:

`decodeCharArray()`	`decodeOctetArray()`
`decodeShortArray()`	`decodeUShortArray()`
`decodeLongArray()`	`decodeULongArray()`
`decodeFloatArray()`	`decodeBooleanArray()`

Each takes a pointer which is updated to point to the first element of the array, and a reference to a `CORBA::ULong` which is updated to hold the length of the array.

Restrictions on Some Compilers

On most compilers, a `CORBA::Float` can be inserted as follows:

```
// C++
CORBA::Float f = ....;
r << f;
```

However, for some compilers, it is necessary to cast the `CORBA::Float` as it is being inserted:

```
// C++
r << (CORBA::Float)f;
```

Otherwise it may be implicitly cast to a C++ `double`.

The latter form needs to be used when writing portable code.

Similarly, some compilers require an explicit cast to insert object references:

```
// C++
CORBA::Object_ptr o = ...
r << (CORBA::Object_ptr)o;
```

Dynamic Skeleton Interface

The Dynamic Skeleton Interface (DSI) is the server-side equivalent of the DII. It allows a server to receive an operation or attribute invocation on any object, even one with an IDL interface that is unknown at compile time. The server does not need to be linked with the skeleton code for an interface to accept operation invocations on that interface.

Instead, a server can define a function that is informed of an incoming operation or attribute invocation. That function determines the identity of the object being invoked. The name of the operation and the types and values of each argument must be provided by the user. The function can then carry out the task that is being requested by the client, and construct and return the result.

Just as use of the DII is significantly less common than use of the normal static invocations, use of the DSI is significantly less common than use of the static interface implementations. A client is not aware that a server is, in fact, implemented using the DSI; it simply makes IDL calls as normal.

To process incoming operation or attribute invocations using the DSI, a server must make a call to the ORB to indicate that it wishes to use the DSI for a specified IDL interface. The same server can use the static interface implementations to handle operation or attribute invocations on other interfaces: however, it cannot use the DSI and static implementation on the same interface.

Uses of the DSI

The DSI has been explicitly designed to help programmers write gateways. Using the DSI, a gateway can accept operation or attribute invocations on any specified set of interfaces and pass them to another system. A gateway can be written to interface between CORBA and some non-CORBA systems. The gateway would need to know the protocol rules of non-CORBA system but it would be the only part of the CORBA system that would require this knowledge. The rest of the CORBA system would continue to make IDL calls as usual.

The IIOP protocol allows an object in one ORB to invoke on an object in another ORB. Non-CORBA systems do not need to support this protocol. One way to interface CORBA to such systems is to construct a gateway using the DSI. This gateway would appear as a CORBA server that contains many CORBA objects. In reality, the server would use the DSI to trap the incoming invocations and translate them into calls to the non-CORBA system. A combination of the DSI and DII allows a process to be a bidirectional gateway. The process can receive messages from the non-CORBA system and use the DII to make

CORBA calls. It can use the DSI to receive requests from the CORBA system and translate these into messages in the non-CORBA system.

Other uses of the DSI are also possible. For example, a server can contain a very large number of non-CORBA objects that it wishes to make available to its clients. One way to achieve this is to provide an individual CORBA object to act as a front-end for each non-CORBA object. However, in some cases this multiplicity of objects may cause too much overhead.

Another way is to provide a single front-end object that can be used to invoke on any of the objects, probably by adding a parameter to each call that specifies which non-CORBA object is to be manipulated. This would of course change the client's view because the client would not be able to invoke on each object individually, treating it as a proper CORBA object.

The DSI can be used to achieve the same space saving as achieved when using a single front-end object, but clients can be given the view that there is one CORBA object for each underlying object. The server would indicate that it wished to accept invocations on the IDL interface using the DSI, and, when informed of such an invocation, it would identify the target object, the operation or attribute being called, and the parameters (if any). It would then make the call on the underlying non-CORBA object, receive the result, and return it to the calling client.

Using the DSI

To use the DSI, you must perform the following steps in your server program:

1. Create one or more objects that have the `CORBA::DynamicImplementation` interface, and register these with Orbix.
2. Register each of these objects to handle requests for a specified IDL interface.

Creating CORBA::DynamicImplementation Objects

The IDL interface `CORBA::DynamicImplementation` is defined as follows:

```
// Pseudo IDL
// In module CORBA.
pseudo interface DynamicImplementation {
        void invoke(inout ServerRequest request,
                          inout Environment env);
};
```

The single operation, `invoke()`, is informed of incoming operation and attribute requests. It can use the `ServerRequest` parameter to determine what operation or attribute is being invoked and on what object. This parameter is also used to obtain `in` and `inout` parameters, and to return `out` and `inout` parameters and the return value to the caller. It can also be used to return an exception to the caller. An implementation of `invoke()` is known as a Dynamic Implementation Routine (DIR).

Interface DynamicImplementation is invisible to clients. In particular, the interfaces that they use do not inherit from it. If they were to inherit from DynamicImplementation, then the fact that the DSI is used at the server-side would not be transparent to the clients.

Registering CORBA::DynamicImplementation Objects

Once an instance of DynamicImplementation has been created, it must be registered to handle requests of a specified interface by calling the setImpl() operation on the CORBA::Orbix object:

```
// IDL
// In module CORBA.
interface BOA {
    ...
    void setImpl(in ImplementationDef implDef,
                      in DynamicImplementation impl);
    ...
};
```

The ServerRequest object that is passed to DynamicImplementation::invoke() is created by Orbix once it receives an incoming request and recognizes it as one that is to be handled by the DSI. This means that an instance of DynamicImplementation has been registered to handle the target interface.

The ServerRequest Data Type

The ServerRequest type is defined in IDL as follows:

```
// Pesudo IDL
// In module CORBA.
pseudo interface ServerRequest {
    Identifier op_name();
    Context ctx();
    any result();
    void params(inout NVList parms);

    // The following are Orbix specific.
    readonly attribute Object target;
    readonly attribute Identifier operation;
        // operation is the same as op_name()
    attribute NVList arguments;
        // arguments is closely related to params()
    attribute any exception;
    attribute Environment env;
};
```

Instances of this interface are pseudo-objects; this means that references to them cannot be transmitted through IDL interfaces.

Because this is a recent addition to the CORBA standard, it was necessary to make Orbix-specific extensions to it to address some inconsistencies in the standard, and also to provide compatibility between type Request and ServerRequest.

The attributes and operations of `ServerRequest` have the following meanings:

`target`	This is an object reference to the target object. Naturally, the target object will not actually exist as a normal CORBA object, so this is actually an object (of a derived type of `CORBA::Object`) that is created by Orbix temporarily for the duration of the call. The operations on this object can be used to determine the marker of the target object, and its interface name.
`operation` / `op_name()`	This attribute or operation gives the name of the operation being invoked.
`arguments` / `params()`	This attribute or operation allows the `invoke()` operation to specify the types of incoming arguments. The attribute `arguments` is explained in detail later in this section.
`result`	This allows the `invoke()` operation to return the result of an operation or attribute call to the caller. In C++, the result is given as a pointer to a `CORBA::Any` that holds the value to be returned to the caller.
`exception`	This allows the `invoke()` operation to return an exception to the caller. In C++, the exception is given as a pointer to a `CORBA::Any` that holds the exception to be returned to the caller.
`env`	This returns the environment parameter (of type `CORBA::Environment`) associated with the call.
`ctx`	This returns the context associated with the call.

There are some special rules determining how you can call these attributes and operations:

`operation` / `op_name()`	This attribute/operation must be called at least once in *each* execution of the `invoke()` function.
`arguments` / `params()`	This attribute/operation must be called exactly once in *each* execution of the `invoke()` function.
`result`	This must be called once for operations with non-void return types and not at all for operations with `void` return types. If it is called, the `exception` attribute cannot be used.
`exception`	This can be called at most once. If it is called, the `result` attribute cannot be used.
`ctx`	This can be called at most once. If it is called, it must be called before the `arguments`/`params()` attribute/operation is called.

The other attributes, `target`, `operation` and `env`, can be used at any time, and any number of times.

Example of Using the DSI

To implement the Dynamic Implementation Routine, `invoke()`, you should first declare a class that inherits from `CORBA::DynamicImplementation`; for example:

```
// C++
class myDSI :
    public CORBA::DynamicImplementation {
public:
    virtual void invoke(CORBA::ServerRequest&);
};
```

You must create an instance of this and register it using `CORBA::BOA::setImpl()`.

```
// C++
{
    myDSI myDSIinstance;
    CORBA::Orbix.setImpl("interfaceName",
             myDSIinstance);
    ...
}
```

The following pseudo-code gives an outline of how to implement a simple version of `invoke()`. It explicitly tests for operations called "`firstOp`" and "`secondOp`". An outline of the code for "`firstOp`" is shown:

```
// C++
void myDSI::invoke(CORBA::ServerRequest& rSrvReq,
    CORBA::Environment& env,
        CORBA::Environment& IT_env =
            CORBA::IT_chooseDefault_Env())
{
    CORBA::Object_ptr theTarget = rSrvReq.target();
    // Use _marker() to determine the marker of the
    // target object.

    const char* pOpName = rSrvReq.op_name();[1]

    try {
        if (strcmp(pOpName,"firstOp") == 0 ) {
            // Access the in and inout parameters and
            // set up variables that will hold the
            // out parameters. Both steps are achieved
            // using params(), explained later.

            // Carry out the required actions.
            // If anything goes wrong, use exception()
            // to pass an exception back to the caller.

            // Prepare to send the reply to the caller.
            // First construct a CORBA::Any object to
            // hold the value.
            CORBA::Any* pResult = new CORBA::Any;
            // Secondly, insert the value into pResult,
            // using operator<<=().
            pResult <<= 24;
```

1. `OpName` then holds the name of the invoked operation; if an attribute, say `attr`, is called, the name will be "`_set_attr`" or "`_get_attr`".

```
            // Then use result() to give the result back:
                rSrvReq.result(pResult);
            }
            else if ( strcmp(pOpName, "secondOp") == 0 ) {
                // Similar code as before.
            }
        }
        catch (...) {
            // Use exception() to pass an exception
            // back to the caller.
            // Note that CORBA forbids invoke()
            // raising an exception.
        }
    }
```

Some real implementations of `invoke()` may not have a set of strings to compare using `strcmp()`, but instead may need to look up some configuration table, or determine how to proceed in some other way.

Example of Using params()

In the first example of using `params()`, it is assumed that there are two arguments to the operation called, both `in` parameters of type `short`, and named "`n`" and "`m`", respectively. There is also a return value of type `long`.

The code to call `params()` and `result()` can then be as follows:

```
// C++
// Build the argument list.
CORBA::NVList_ptr pArgList;

if (CORBA::Orbix.create_list(2, pArglist) {

    CORBA::Short valueOf_n = 0;
    CORBA::Short valueOf_m = 0;
    CORBA::Any* pFirstAny = new CORBA::Any
            (CORBA::_tc_short, &valueOf_n, 0);
    CORBA::Any pSecondAny = new CORBA::Any
            (CORBA::_tc_short, &valueOf_m, 0);
    pArgList->add_value("n", *pFirstAny,
        CORBA::DSI_ARG_IN);
    pArgList->add_value("m", *pSecondAny,
        CORBA::DSI_ARG_IN);

    // Give the prepared arg. list to the ServerRequest.
    rSrvReq.params(pArgList );

    // Now, valueOf_n contains the value or parameter n.
    // And valueOf_m contains the value of parameter m.

    // Prepare the space for the reply:
    CORBA::Long pValue;
    // Then execute the required code for the operation
    // that the client has called. Put the final value
    // in pValue.

    // Create the result.
    CORBA::Any* pResult = new CORBA::Any;
    pResult <<= pValue;
    rSrvReq.result(pResult, IT_env);
    ...
}
```

In the second example of using `params()`, it is assumed that there are two arguments to the operation that has been called, the first, named "`n`", is an `out` parameters of type `short`, and the second, named "`m`", is an `inout` parameter of type `long`. There is no return value. The code to call `params()` can then be as follows:

```
// C++
// Build the argument list.
CORBA::NVList_ptr pArgList;

if (CORBA::Orbix.create_list(2, pArglist) {
    CORBA::Short valueOf_n = 0;
    CORBA::Long valueOf_m = 0;
    CORBA::Any* pFirstAny = new CORBA::Any
            (CORBA::_tc_short, &valueOf_n, 0);
    CORBA::Any* pSecondAny = new CORBA::Any
            (CORBA::_tc_long, &valueOf_m, 0);
    pArgList->add_value
        ("n", *pFirstAny, CORBA::DSI_ARG_OUT);
    pArgList->add_value
        ("m", *pSecondAny, CORBA::DSI_ARG_INOUT);

    // Give the prepared arg. list to the ServerRequest:
    rSrvReq.params(pArgList);
    ...
}
```

Once this code has been executed, the proper action of `invoke()` can be carried out. During that time, the incoming value of the second parameter, `m`, is available in `valueOf_m`. The values that `valueOf_n` and `valueOf_m` have at the end of the function call will be passed back to the caller (as the `out` and `inout` parameters, `n` and `m`, respectively).

The Interface Repository

This chapter describes the Interface Repository, the component of Orbix that provides persistent storage of IDL modules, interfaces and other IDL types. Orbix programs can query the Interface Repository at runtime to obtain information about IDL definitions.

There are several ways to use the Interface Repository in your Orbix applications. For example, you can iterate through the Interface Repository to browse or list its contents. Alternatively, given an object reference, the object's type and all information about that type can be determined at runtime by calling functions defined by the Interface Repository.

Such facilities are important for some tools, such as:

- Browsers that allow you to determine that types that have been defined in the system, and to list the details of chosen types.
- CASE tools that aid software design, writing and debugging.
- Application level code that uses the Dynamic Invocation Interface (DII) to invoke on objects whose types were unknown at compile time. This code may need to determine the details of the object being invoked in order to construct the request using the DII.
- A gateway that requires runtime type information about the type of an object being invoked.

The Interface Repository provides a set of IDL interfaces to browse and list its contents, and to determine the type information for a given object.

Configuring the Interface Repository

Before writing applications to read the contents of the Interface Repository, you must first install and configure the repository as described in the ***Orbix C++ Edition Administrator's Guide***.

Orbix implements the Interface Repository using a standard Orbix server named `IFR`. To install the Interface Repository, you must run the Orbix daemon process and register this server.

Orbix provides a command-line utility, called `putidl`, that allows you to add IDL definitions to the Interface Repository. The Orbix GUI tools also includes graphical interface to the Interface Repository. Refer to the ***Orbix C++ Edition Administrator's Guide*** for more details.

Runtime Information about IDL Definitions

The Interface Repository maintains full information about the IDL definitions that have been passed to it. A program can use the Interface Repository to browse through the set of modules and interfaces, determining the name of each module, the name of each interface and the full definition of that interface. A program

can also find a full IDL definition if given the name of a module, interface, attribute, operation, struct, union, enum, typedef, constant or exception.

For example, given any object reference, you can use the Interface Repository to determine all of the information about that interface. In particular, you can determine:

- The module in which the interface was defined, if any.
- The name of the interface.
- The interface's attributes, and their definitions.
- The interface's operations, and their full definition, including parameter, context and exception definitions.
- The interface's base interfaces.

A short example at the end of this chapter demonstrates the use of the Interface Repository.

The Structure of Interface Repository Data

The data in the Interface Repository is best viewed as a set of CORBA objects where one object is stored in the repository for each IDL type definition. Objects in the Interface Repository support one of the following IDL interface types, reflecting the IDL constructs they describe:

`Repository`	The type of the repository itself, in which all of its other objects are nested.
`ModuleDef`	The interface for a `ModuleDef` definition. Each module has a name and can contain definitions of any type (except `Repository`).
`InterfaceDef`	The interface for an `InterfaceDef` definition. Each interface has a name, a possible inheritance declaration, and can contain definitions of type attribute, operation, exception, typedef and constant.
`AttributeDef`	The interface for an `AttributeDef` definition. Each attribute has a name and a type, and a mode that determines whether or not it is readonly.
`OperationDef`	The interface for an `OperationDef` definition. Each operation has a name, a return value, a set of parameters and, optionally, `raises` and `context` clauses.
`ConstantDef`	The interface for a `ConstantDef` definition. Each constant has a name, a type and a value.
`ExceptionDef`	The interface for an `ExceptionDef` definition. Each exception has a name and a set of member definitions.
`StructDef`	The interface for a `StructDef` definition. Each `struct` has a name, and also holds the definition of each of its members.
`UnionDef`	The interface for a `UnionDef` definition. Each union has a name, and also holds a discriminator type and the definition of each of its members.

EmumDef	The interface for an EnumDef definition. Each enum has a name, and also holds its list of member identifiers.
AliasDef	The interface for a typedef statement in IDL. Each alias has a name and a type that it maps to.
PrimitiveDef	The interface for primitive IDL types. Objects of this type correspond to a type such as short and long, and are pre-defined within the Interface Repository.
StringDef	The interface for a string type. Each string type records its bound. Objects of this type do not have a name. If they have been defined using an IDL typedef statement, then they will have an associated AliasDef object. (Objects of this type correspond to bounded strings.)
SequenceDef	The interface for a sequence type. Each sequence type records its bound (a value of zero indicates an unbounded sequence type) and its element type. Objects of this type do not have a name. If they have been defined using an IDL typedef statement, then they will have an associated AliasDef object.
ArrayDef	The interface for an array type. Each array type records its length and its element type. Objects of this type do not have a name. If they have been defined using an IDL typedef statement, then they will have an associated AliasDef object. Each ArrayDef object represents one dimension; multiple ArrayDef objects are required to represent a multi-dimensional array type.

In addition, the following abstract types (types without direct instances) are defined:

```
IRObject
IDLType
TypedefDef
Contained
Container
```

Understanding these types is the key to understanding how to use the Interface Repository.

Containment Relationships

You can interrogate any object of these types to determine their definitions. They are organized in a natural manner according to the IDL interface. For example, each InterfaceDef object is said to contain objects representing the interface's constant, type, exceptions, attribute, and operation definitions. The outermost object is of type Repository.

The containment relationships between the Interface Repository types are as follows:

A Repository can contain:

```
ConstantDef
TypedefDef
ExceptionDef
```

```
InterfaceDef
ModuleDef
```

A `ModuleDef` can contain:

```
ConstantDef
TypedefDef
ExceptionDef
ModuleDef
InterfaceDef
```

An `InterfaceDef` can contain:

```
ConstantDef
TypedefDef
ExceptionDef
AttributeDef
OperationDef
```

Objects of type `ModuleDef`, `InterfaceDef`, `ConstantDef`, `ExceptionDef`, and `TypedefDef` can also appear outside of any module, directly within a repository.

You can determine the full interface definition given any object of the Interface Repository types. For example, `InterfaceDef` defines operations or attributes to determine an interface's name, its inheritance hierarchy, and the description of each operation and each attribute.

Refer to "Containment interface types" for more information.

Simple Types

The Interface Repository defines the following simple IDL definitions:

```
// IDL
// In module CORBA.
typedef string Identifier;
typedef string ScopedName;
typedef string RepositoryId;
typedef string VersionSpec;

enum DefinitionKind {
    dk_none, dk_all,
    dk_Attribute, dk_Constant,
    dk_Exception, dk_Interface,
    dk_Module, dk_Operation, dk_Typedef,
    dk_Alias, dk_Struct, dk_Union, dk_Enum,
    dk_Primitive, dk_String, dk_Sequence, dk_Array,
    dk_Repository
};
```

An `Identifier` is a simple name that identifies modules, interfaces, constants, typedefs, exceptions, attributes, and operations.

A `ScopedName` gives an entity's name relative to a scope. A `ScopedName` that begins with "`::`" is an *absolute scoped name*. This is a name that uniquely identifies an entity within a repository. For example, the name `::Finance::Account::withdraw`. A `ScopedName` that does not begin with "`::`" is a *relative scoped name*. This is a name that identifies an entity relative to some other entity. For example, `withdraw` within the entity with the absolute scoped name `::Finance::Account`.

A `RepositoryId` is a string that uniquely identifies an object within a repository, or globally within a set of repositories if more than one is being used. The object can be a constant, exception, attribute, operation, structure, union, enumeration, alias, interface or module.

Type `VersionSpec` is used to indicate the version number of an Interface Repository object. This means that it allows the Interface Repository to distinguish two or more versions of a definition, each with the same name but with details that evolve over time. However, the Interface Repository is not required to support such versioning. It is not required to store more than one definition with any given name. The Orbix Interface Repository currently does not support versioning.

Each Interface Repository object has an attribute (called `def_kind`) of type `DefinitionKind` that records the kind of the Interface Repository object. For example, the `def_kind` attribute of an `interfaceDef` object is `dk_interface`. The enumerate constants `dk_none` and `dk_all` have special meanings when searching for objects in a repository.

Abstract Interfaces in the Interface Repository

There are five abstract interfaces defined for the Interface Repository, as follows:

- `IRObject`
- `IDLType`
- `TypedefDef`
- `Contained`
- `Container`

These are of key importance in understanding the basic structure of the Interface Repository, and provide basic functionality for each of the concrete interface types.

Class Hierarchy and Abstract Base Interfaces

The Interface Repository defines five abstract base interfaces (interfaces that cannot have direct instances). These are used to define the other Interface Repository types:

`IRObject`	This is the base interface of all Interface Repository objects. Its only attribute defines the kind of an Interface Repository object.
`IDLType`	All Interface Repository interfaces that hold the definition of a type directly or indirectly inherit from this interface.
`TypedefDef`	This is the base interface for all Interface Repository types that can have names (except interfaces): structures, unions, enumerations and aliases (results of IDL `typedef` definitions).
`Contained`	Many Interface Repository objects can be contained in others and these all inherit from `Contained`. The exact meaning of containment is explained later.
`Container`	Some Interface Repository interfaces, such as `Repository`, `ModuleDef` and `InterfaceDef`, can contain other Interface Repository objects. These interfaces inherit from `Container`.

The interface hierarchy for all of the Interface Repository interfaces is shown in Figure 21 on page 216.

The Interface IRObject

The interface `IRObject` is defined as follows:

```
// IDL
// In module CORBA.
interface IRObject {
    // read interface
    readonly attribute DefinitionKind def_kind;

    // write interface
    void destroy ();
};
```

This is the base interface of all Interface Repository types. The attribute `def_kind` is useful because it provides a simple way of determining the type of an Interface Repository object. Other than defining an attribute and operation, and acting as the base interface of other interfaces, `IRObject` plays no further role in the Interface Repository.

You can delete an Interface Repository object by calling its `destroy()` operation. This also deletes any objects contained in the target object. It is an error to call `destroy()` on a `Repository` or a `PrimitiveDef` object.

Containment in the Interface Repository

Definitions in the IDL language have a nested structure. For example, a module can contain definitions of interfaces, and the interfaces themselves can contain definitions of attributes, operations and many others. Consider the following fragment of IDL:

```
// IDL
module Finance {
    interface Account {
        readonly attribute float balance;
        void deposit(in float amount);
        void withdraw(in float amount);
    };
    interface Bank {
        Account create_account();
    };
};
```

In this example the module `Finance` (represented in the Interface Repository as a `ModuleDef` object) contains the two interface definitions `Bank` and `Account` (each represented by an individual `InterfaceDef` object). These two interfaces contain further definitions. For example, the interface `Account` contains a single attribute and two operations.

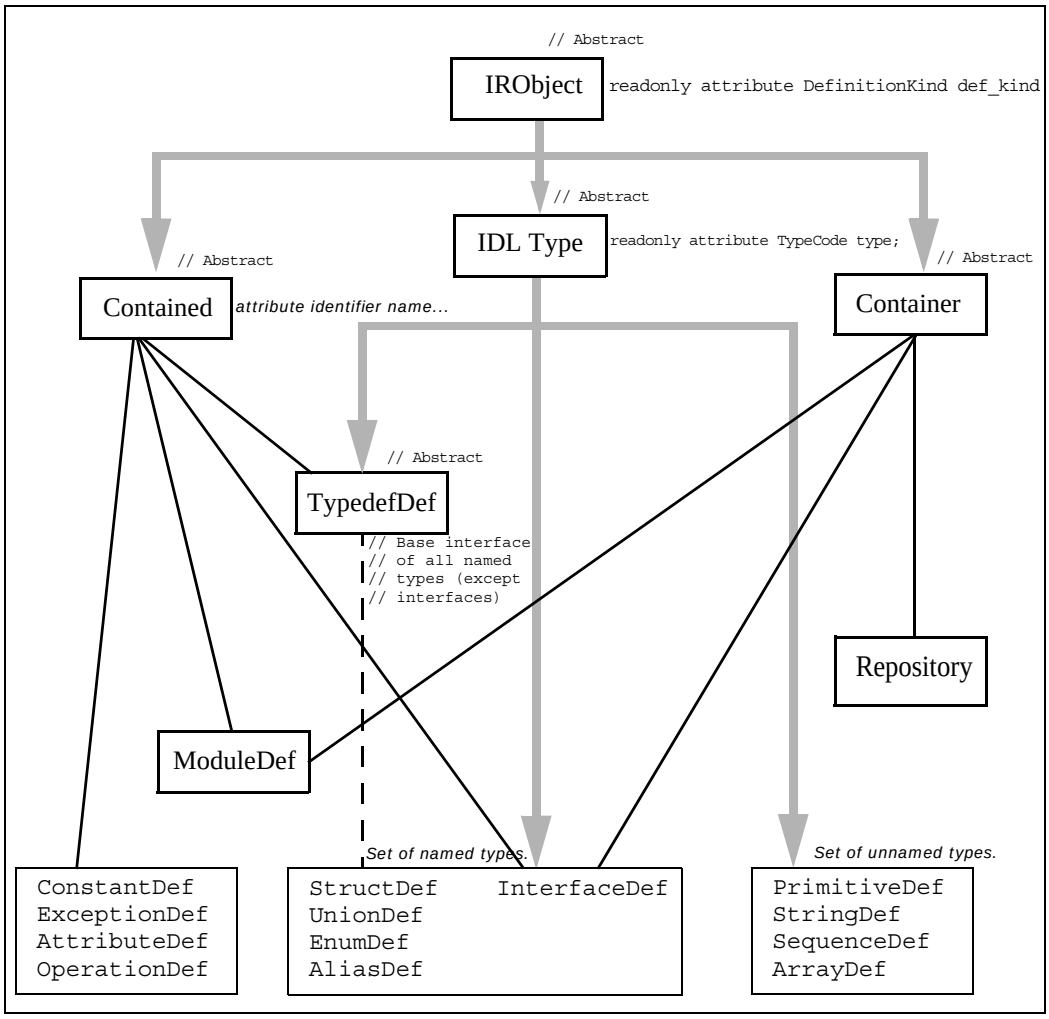


Figure 21: *The Hierarchy for Interface Repository Interfaces*

The notion of containment is basic to the structure of the IDL definitions, and the Interface Repository specification abstracts the properties of containment. For example, an Interface Repository object (such as a `ModuleDef` or `InterfaceDef` object) that can contain further definitions also needs a function to list its contents. Similarly an Interface Repository object that can be contained within another Interface Repository object may want to know the identity of the object it is contained in. This leads naturally to the definition of two abstract base interfaces `Container` and `Contained`, which group together common operations and attributes. Most of the objects in the repository are derived from one or both of `Container` or `Contained` (the exceptions are instances of `PrimitiveDef`, `StringDef`, `SequenceDef`, and `ArrayDef`).

You can access a considerable part of the structure of the Interface Repository by using the operations and attributes of `Container` and `Contained`. Understanding containment is crucial to understanding most of the Interface Repository functionality.

Containment interface types

The interfaces that use containment are of three different types:

- Interfaces that inherit only from `Container`.
- Interfaces that inherit from both `Container` and `Contained`.
- Interfaces that inherit only from `Contained`.

These are as follows:

Inheriting From	Interface
`Container`	`Repository`
`Container` and `Contained`	`ModuleDef`, `InterfaceDef`
`Contained`	`ConstantDef`, `ExceptionDef`, `AttributeDef`, `OperationDef`, `StructDef`, `UnionDef`, `EnumDef`, `AliasDef`, `TypedefDef`

The `Repository` itself is the only interface that can be a pure `Container`. There is only one `Repository` object per Interface Repository server and it has all of the other definitions nested inside it.

Objects of type `ModuleDef` and `InterfaceDef` can create additional layers of nesting and therefore they derive from both `Container` and `Contained`.

The remaining types of objects have a simpler structure and derive just from `Contained`. The last interface, `TypedefDef`, is unique in that it is an abstract interface.

The Contained Interface

This section is limited to a discussion of the basic attributes and operations of interface `Contained`. An outline of the `Contained` interface is as follows:

```
//IDL
typedef Identifier string;

interface Contained : IRObject {
    // Incomplete list of operations and attributes...
    ...
    attribute Identifier name;
    ...
    readonly attribute Container defined_in;
    ...
    struct Description {
        DefinitionKind kind;
        any value;
    };
    Description describe();
};
```

A basic attribute of any contained object is its `name`. The attribute `name` has the type `Identifier` that is just a `typedef` for a string. For example, the module `Finance` is represented in the repository by a `ModuleDef` object. The inherited `ModuleDef::name` attribute resolves to the string `"Finance"`. Similarly, an `OperationDef` object

representing `withdraw` has an `OperationDef::name` that resolves to `"withdraw"`. The `Repository` object itself evidently has no name, because it does not inherit from `Contained`.

Another basic attribute is `Contained::defined_in` that stores an object reference to the `Container` in which the object is defined. This attribute is all that is needed to express the idea of containment for a `Contained` object. The attribute `defined_in` stores a uniquely defined `Container` reference because a given definition appears only once in IDL. However, because of the possibility of inheritance between interfaces, a given object may be contained in more than one interface. In the following example, interface `CurrentAccount` is derived from interface `Account`:

```
//IDL
// in module Finance
interface CurrentAccount : Account {
    readonly attribute overDraftLimit;
};
```

The attribute balance is contained in interface `Account` and also contained in interface `CurrentAccount`. However, the result of querying `AttributeDef::defined_in()` for the `balance` attribute will always return an object for `Account`. This is because the definition of attribute `balance` appears in the base interface `Account`.

A `Contained` object may include more than just containment information. For example, an `OperationDef` object has a list of parameters associated with it and details of the return type. The operation `Contained::describe()` provides access to these details by returning a generic `Description` structure (discussed later).

The Container Interface

Some of the basic definitions for interface `Container` are as follows:

```
//IDL
typedef sequence<Contained> ContainedSeq;
enum DefinitionKind {dk_name, dk_all,
    dk_Attribute, dk_Constant, dk_Exception,
    dk_Interface, dk_Module, dk_Operation,
    dk_Typedef, dk_Alias, dk_Struct, dk_Union,
    dk_Enum, dk_Primitive, dk_String, dk_Sequence,
    dk_Array, dk_Repository};

interface Container : IRObject {
    // Incomplete list of operations and attributes
    ...
    ContainedSeq contents(
        in DefinitionKind limit_type,
        in boolean exclude_inherited);
    ...
};
```

Container::contents()

The `contents()` operation is the most basic operation associated with a `Container`. It returns a sequence of `Contained` objects that belong to the `Container`. By using `contents()`, it is possible to browse a `Container` and descend nested layers of containment. Once the appropriate `Contained` object has been found, the details

of its definition can be found by invoking `Contained::describe()` to obtain a detailed `Description` of the object. The use of `Container::contents()` coupled with `Contained::describe()` provides a basic way of browsing the Interface Repository. However, there are a number of approaches to browsing the Interface Repository that can be more efficient. These more sophisticated search operations are discussed in "Retrieving Information about IDL Definitions" on page 225.

The arguments to operation `contents()` make use of `DefinitionKind`. This is an `enum` type that is used to tag the different kinds of repository objects. In addition to the interfaces for concrete repository objects there are three additional tags: The tag `dk_none` matches no repository object, the tag `dk_all` matches any repository object, and the tag `dk_Typedef` matches any one of `dk_Alias`, `dk_Struct`, `dk_Union`, or `dk_Enum`. The arguments to `contents()` can be described as follows:

`limit_type`	A tag of type `DefinitionKind` that can be used to limit the list of contents to certain kinds of repository objects. A value of `dk_all` lists all objects.
`exclude_inherited`	This argument is only relevant if the `Container` happens to be an `InterfaceDef` object. In the case of an `InterfaceDef`, it determines whether or not inherited definitions should be included in the contents listing. `TRUE` indicates they should be excluded while `FALSE` indicates they should be included.

The returned value is then a sequence of `Contained` objects that match the given criteria.

There are a number of additional operations of the interface `Container` that enable efficient searching of the repository. Refer to the ***Orbix Programmer's Reference C++ Edition*** for details.

Containment Descriptions

The containment framework reveals which definitions are made within which interface or module. However, each repository object, besides the possible property of being a `Contained` or `Container`, also retains the details of an IDL definition. Calling `describe()` on a `Contained` object returns a `Description` struct holding these details.

Both interfaces `Contained` and `Container` define their own version of a `Description` struct which are, respectively, `Contained::Description` and `Container::Description`. The `Container::Description` structure differs slightly from the `Contained::Description`.

Consider the following fragment of the IDL interface for Container:

```
//IDL
interface Container : IRObject {
    // Incomplete listing of interface
    ...
    struct Description {
        Contained contained_object;
        DefinitionKind kind;
        any value;
    };
    typedef sequence<Description> DescriptionSeq;
    DescriptionSeq describe_contents(
        in DefinitionKind limit_type,
        in boolean exclude_inherited,
        in long max_returned objects);
    ...
};
```

Note the extra member contained_object appearing in this Description structure.

Container::describe_contents()

The Container::Description is used by describe_contents(). This operation effectively combines calling contents() on the Container plus calling describe() on each of the returned objects.

The arguments to describe_contents() are as follows:

limit_type	A tag of type DefinitionKind which may be used to limit the list of contents to certain kinds of repository objects. A value of dk_all lists all objects.
exclude_inherited	This argument is only relevant if the Container happens to be an InterfaceDef object. For the case of an InterfaceDef it determines whether or not inherited definitions should be included in the contents listing. TRUE indicates they should be excluded while FALSE indicates they should be included.
max_returned_objects	Specifies the maximum length of the sequence returned.

The describe_contents() operation returns a sequence of Description structures, one for each of the Contained objects found.

The Description structure itself serves as a wrapper for the detailed description that is specific to a repository object. For example, the interface OperationDef inherits the operation OperationDef::describe.

OperationDesription

Associated with the OperationDef interface is the struct OperationDescription. This has the following structure:

```
// IDL
struct OperationDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode result;
    OperationMode mode;
    ContextIdSeq contexts;
    ParDescriptionSeq parameters;
    ExcDescriptionSeq exceptions;
};
```

This structure is not returned directly by the operation OperationDef::describe(). Initially it returns a Contained::Description wrapper. The first layer is tagged by Description::kind, which equals dk_Operation, and the substance of the Description is in the Description::value. The second layer is given by the value, which is an any. Inside the any there is a TypeCode _tc_OperationDescription and the value of the any is the OperationDescription structure itself.

The various members of the OperationDescription structure have the following meaning:

name	The name of the operation as it appears in the definition. For example, the operation Account::makeWithdrawal would have the name "makeWithdrawal".
id	The id is just a RepositoryId for the OperationDef object. A RepositoryId is basically a particular way of naming repository objects.
defined_in	The member defined_in gives the RepositoryId for the parent Container of the OperationDef object.
version	The version of type VersionSpec is used to indicate the version number of an Interface Repository object. This would allow the Interface Repository to distinguish two or more versions of a definition which have the same name but with details that evolve over time. The Orbix Interface Repository currently does not support versioning.
result	The TypeCode of the result returned by the defined operation.
mode	The mode specifies whether the operation is normal (OP_NORMAL) or oneway (OP_ONEWAY).
contexts	The member contexts is of type ContextIdSeq that is a typedef for a sequence of strings. The sequence lists the context identifiers specified in the context clause of the operation.
parameters	The member parameters is a sequence of ParameterDescription structures that give details of each parameter to the operation.

exceptions	The member exceptions is a sequence of ExceptionDescription structures giving details of the exceptions specified in the raises clause of the operation.

The OperationDescription provides all of the information present in the original definition of the operation. As is the case with many aspects of the Interface Repository, the CORBA specification provides for more than one way of accessing this information. The interface OperationDef also defines a number of attributes that allow direct access to the members of the above structure. Frequently, in a distributed environment it is more convenient to obtain the complete description in a single step. This is why the OperationDescription structure is provided.

Only those repository interfaces that inherit from Contained have an associated description structure, and of those, not all have a unique description structure. Specifically, the interfaces EnumDef, UnionDef, AliasDef, and StructDef all use a similar sort of description called TypeDescription.

The interface InterfaceDef is a special case because there is an extra description structure associated with it called FullInterfaceDescription. This structure is provided in the light of the special importance of InterfaceDef objects. It enables a full description of the interface plus all its contents to be obtained in one step. The description is given as the return value of the special operation InterfaceDef::describe_interface().

Type Interfaces in the Interface Repository

A number of repository interfaces are used to represent definitions of types in the Interface Repository. These are the following interfaces:

- StructDef
- UnionDef
- EnumDef
- AliasDef
- InterfaceDef
- PrimitiveDef
- StringDef
- SequenceDef
- ArrayDef

This property is independent of and overlaps with the properties of containment. It is useful to represent this property by having those objects inherit from an abstract base interface which is called IDLType and is defined as follows:

```
// IDL
// In module CORBA.
interface IDLType : IRObject {
    readonly attribute TypeCode type;
};
```

This base interface defines just a single attribute that gives the TypeCode of the defined type. It is also useful for referring to the type interfaces collectively.

The type interfaces can be classified as either named or unnamed types.

Named Types

The named type interfaces are as follows:

- StructDef
- UnionDef
- EnumDef
- AliasDef
- InterfaceDef.

For example, consider the following IDL definition:

```
// IDL
enum UD {UP, DOWN};
```

This effectively defines a new type, UD, which may be used wherever an ordinary type might appear. It is represented by an EnumDef object. More obviously, the following IDL definition gives rise to the new type AccountName:

```
typedef string AccountName;
```

These two interfaces are examples of named types. That is, the definitions give rise to a new type identifier, such as "UD" or "AccountName", which may be reused throughout the IDL file.

A further distinction is made between InterfaceDef and the other named types. The named types StructDef, UnionDef, EnumDef, and AliasDef are grouped together by deriving from the abstract base interface TypedefDef. It is important to note that interface TypedefDef does not directly represent an IDL typedef. The interface AliasDef (which derives from TypedefDef) is the interface representing an IDL typedef. The abstract interface TypedefDef is defined as follows:

```
// IDL
// In module CORBA.
interface TypedefDef : Contained, IDLType {
};
```

The definition of TypedefDef is trivial and causes the four named interfaces to derive from Contained in addition to IDLType. The interfaces inherit the attribute Contained::name. This gives the name of the type, and the operation Contained::describe().

For example, the definition of enum UD gives rise to an EnumDef object which has an EnumDef::name of "UD". Calling EnumDef::describe() gives access to a description of type TypeDescription. The type member of the TypeDescription gives the TypeCode of the enum. The TypedefDef interfaces all share the same description structure, TypeDescription.

The interface InterfaceDef is also a named type but it is a special case. Its inheritance is given as follows:

```
// IDL
// In module CORBA.
interface InterfaceDef : Contained, Container, IDLType
{
    ...
};
```

Interface InterfaceDef has three base interfaces. Since IDL object references can be used in just the same way as any ordinary type, IntefaceDef inherits from IDLType. For example, the definition interface Account {...}; gives rise to an InterfaceDef object. This object has an InterfaceDef::name which is "Account" and this name may be reused as a type.

Unnamed Types

The unnamed type interfaces are as follows:

- PrimitiveDef
- StringDef
- SequenceDef
- ArrayDef

These interfaces are not strictly necessary but offer an approach to querying the types in the repository that operates in parallel to the use of TypeCodes.

Therefore there are two independent approaches to querying types in the repository. The traditional approach is to provide TypeCode attributes whenever necessary so that all the types defined in the repository can be determined. However, the Interface Repository also provides a complete object-oriented approach for querying the types.

Consider the following example where the return type of getLongAddress needs to be determined:

```
// IDL
interface Mailer {
    sequence<string> getLongAddress();
};
```

The definition of getLongAddress() maps to an object of type OperationDef in the repository. One way of querying the return type is to call OperationDef::result_def() that returns an object reference of type IDLType. The type of object returned by result_def() can be determined by getting the attribute OperationDef::def_kind that is inherited from IRObject.

In this example the object reference is of type SequenceDef corresponding to the sequence<string> return type. The returned SequenceDef object may be further queried by getting the attribute SequenceDef::element_type_def. This returns an IDLType which is a PrimitiveDef object. This PrimitiveDef object, in turn, has an attribute PrimitiveDef::kind that has a value of pk_string. At this stage the return type has been fully determined to be a sequence<string>.

The alternative approach is to obtain the TypeCode at the outset. This retrieves the complete type information in a single step. For example, the OperationDef object associated with getLongAddress has an attribute OperationDef::result, which gives the TypeCode of sequence<string>.

Retrieving Information about IDL Definitions

There are three ways to retrieve information from the Interface Repository:

1. Given an object reference, its corresponding InterfaceDef object can be found. From this, all of the details of the object's interface definition can be determined.
2. Obtain an object reference to a Repository, after which the full contents can then be navigated.
3. Given a RepositoryId, a reference to the corresponding object in the Interface Repository can be obtained and interrogated.

These are explained in more detail in the following three subsections.

CORBA::Object::_get_interface()

Given an object reference to any CORBA object, say objVar, an object reference to an InterfaceDef object can be acquired as follows:

```
// C++
// Must include <ifr.hh>
CORBA::InterfaceDef_var ifVar =
            objVar->_get_interface();
```

The member function _get_interface() returns a reference to an object within the Interface Repository.

Browsing or Listing a Repository

Once a reference to a Repository object is obtained, the contents of that repository can be browsed or listed. There are two ways to obtain such an object reference.

Firstly, the resolve_initial_references() operation can be called on the ORB (of type CORBA::ORB), passing the string "InterfaceRepository" as a parameter. This returns an object reference of type CORBA::Object, which can then be narrowed to a CORBA::Repository reference.

Alternatively, the Orbix _bind() function can be used, as follows:

```
// C++
Repository_var repVar =
    Repository::_bind(
        "IDL\\:abigbank.com/Repository:IFR", "host");
```

The operations that enable browsing of the Repository are provided by the abstract interface Container. There are four provided, as follows:

- contents()
- describe_contents()

- lookup()
- lookup_name()

The last two are particularly useful because they provide a facility for searching the Repository.

The IDL for the search operations is as follows:

```
// IDL
// In module CORBA.
interface Container : IRObject {
    ...
    Contained lookup(in ScopedName search_name);
    ...
    ContainedSeq lookup_name(
        in Identifier search_name,
        in long levels_to_search,
        in DefinitionKind limit_type,
        in boolean exclude_inherited);
    ...
};
```

Container::lookup()

The operation lookup() provides a simple search facility based on a ScopedName. For example, consider the case where Container is a ModuleDef object representing Finance. Passing the string "Account::balance" to ModuleDef::lookup() then retrieves a reference to an AttributeDef object representing balance. This is an example of using a relative ScopedName. However, lookup() is not restricted to just searching a specific Container. By passing an absolute ScopedName as an argument it is possible to search the whole repository given any Container as a starting point. For example, given the InterfaceDef for Account it is possible to pass the string "::Finance::Bank::newAccount" to InterfaceDef::lookup to find the newAccount operation lying within the scope of the interface Bank.

Container::lookup_name()

The operation lookup_name() provides a different approach to searching a Container. Instead of the ScopedName it specifies only a simple name to search for within the Container. Because more than one match is possible with a given simple name, the lookup() operation can return a sequence of Contained objects.

The parameters to lookup_name() are as follows:

search_name	Specifies the simple name of the object to search for. The Orbix implementation also allows the use of "*" which matches any simple name.
levels_to_search	Specifies the number of levels of nesting to be included in the search. If set to 1, the search is restricted to the current object. If set to -1, the search is unrestricted.
limit_type	Limits the objects that are returned. If it is set to dk_all, all objects are returned. If set to the DefinitionKind for a particular Interface Repository kind, only objects of that kind are returned. For example, if operations are of interest, limit_type can be set to dk_operation.

exclude_inherited	If set to TRUE, inherited objects are not returned. If set to FALSE, all objects, including those inherited, are returned.

Note: You cannot use the `lookup_name()` operation to search outside of the given `Container`.

Finding an Object Using its Repository ID

A Repository ID (of type `CORBA::RepositoryId`) can be passed as a parameter to the `lookup_id()` operation of an object reference for a repository (of type `CORBA::Repository`). This returns a reference to an object of type `Contained`, and this can be narrowed to the correct object reference type.

Example of Using the Interface Repository

This section presents some sample code that uses the Interface Repository.The following code prints the list of operation names and attribute names defined on the interface of a given object:

```
// C++
// The following two lines must appear near
// the top of the file:
//
#include <ifr.hh>
#include <IT_ifr.hh>

int i;
Repository_var rVar;
Contained_var cVar;
InterfaceDef_var interfaceVar;
InterfaceDef::FullInterfaceDescription_var full;

try {
    // Bind to the IFR server:
    rVar = Repository::_bind("IDL\\:abigbank.com/
                  Repository:IFR", "host");

    // Get the interface definition:
    cVar = lookup("grid");
    interfaceVar = InterfaceDef::_narrow(cVar);

    // Get a full interface description:
    full = interfaceVar->describe_interface();
    // Now print out the operation names:
    cout << "The operation names are:" << endl;
    for (i=0; i < full->operations.length(); i++)
        cout << full->operations[i].name << endl;
    // Now print out the attribute names:
    cout << "The attribute names are:" << endl;
    for (i=0; i < full->attributes.length(); i++)
        cout << full->attributes[i].name << endl;
}
catch (...) {
    ...
}
```

All applications that use the Interface Repository must include the file `ifr.hh`. This file is available in the `include` directory of your Orbix installation. In addition, you must link these applications against the Orbix library. This is available in the `lib` directory of your Orbix installation.

The example can be extended by finding the `OperationDef` object for an operation called `doit()`. The `Container::lookup_name()` can be used as follows:

```
// C++
ContainedSeq_var opSeq;
OperationDef_var doitOpVar;

try {
    cout << "Looking up operation doit()"
           << endl;
    opSeq = interfaceVar->lookup_name(
                  "doit", 1, dk_Operation, 0);
    if (opSeq->length() != 1) {
        cout << "Incorrect result for lookup_name()";
        exit(1);
    } else {
        // Narrow the result to be an OperationDef.
        doitOpVar =
               OperationDef::_narrow(opSeq[0]))
    }
    ...
}
catch (...) {
    ...
}
```

Repository IDs

Each Interface Repository object that describes an IDL definition has a Repository ID. A Repository ID globally identifies an IDL module, interface, constant, typedef, exception, attribute, or operation definition. A Repository ID is simply a string that identifies the IDL definition.

Three formats for Repository IDs are defined by CORBA. However, Repository IDs are not, in general, required to be in one of these formats. The formats defined by CORBA are described next.

OMG IDL Format

This format is derived from the IDL definition's scoped name. It contains three components which are separated by colons (:) as follows:

```
IDL:<identifier/identifier/
       identifier/...>:<version number>
```

- The first component identifies the Repository ID format as the OMG IDL format.
- The second component consists of a list of identifiers. These identifiers are derived from the scoped name by substituting "/" instead of "::".

- The third component contains a version number with the following format:

  ```
  <major>.<minor>
  ```

Consider the following IDL definitions:

```
// IDL
interface Account {
    attribute float balance;
    void deposit(in float amount);
};
```

The following is an IDL format Repository ID for the attribute `Account::balance` based on these definitions:

```
IDL:Account/balance:1.0
```

This is the format of the Repository ID that is used by default in Orbix.

DCE UUID Format

The DCE UUID format is the following:

```
DCE:<UUID>:<minor version number>
```

LOCAL Format

Local format IDs are for local use within an Interface Repository and are not intended to be known outside that repository. They have the following format:

```
LOCAL:<ID>
```

Local format Repository IDs may be useful in a development environment as a way to avoid conflicts with Repository IDs using other formats.

Pragma Directives

You can control Repository IDs using pragma directives in an IDL source file. These pragmas enable you to control the format of a Repository ID for IDL definitions. At present Orbix supports the use of all three pragma directives: ID, prefix and version.

ID Pragma

An ID pragma directive takes the format:

```
#pragma ID <name> "<id>"
```

The `<name>` can be a fully scoped name or an identifier whose scope is interpreted relative to the scope in which the pragma directive is included. The `<id>` is the repository ID string which is to be associated with the `<name>`.

Prefix Pragma

A Prefix pragma directive takes the format:

```
#pragma prefix "<string>"
```

The `<string>` sets the current prefix used in generating repository IDs. The specified prefix applies to repository IDs generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered.

Version Pragma

You can specify a version number for an IDL definition's Repository ID (IDL format) by using a version pragma. The version pragma directive takes the format:

```
#pragma version <name> <major>.<minor>
```

The `<name>` can be a fully scoped name or an identifier whose scope is interpreted relative to the scope in which the pragma directive is included. Where no version pragma is specified for an IDL definition, the version number for the definition defaults to 1.0.

For example, consider the following:

```
// IDL
module Engineering {
        interface component {
        };
        #pragma ID component
                    "IDL:abigbank.com/component:1.0"
};

#pragma prefix "FirstTrust"
module Finance {
        module Banking {
                #pragma prefix "CorporateBanking"
                    interface Account {
                    };
        };
        module Stockmarket {
                interface invest {
                };
        };
        #pragma version Banking::Account 2.7
};
```

These definitions yield the following Repository IDs:

```
::Engineering::component IDL:abigbank.com/component:1.0
::Finance::Banking::Account IDL:CorporateBanking/Account:2.7
::Finance::Stockmarket::invest IDL:FirstTrust/Finance/Stockmarket/invest:1.0
```

It is important to realize that pragma directives do not only affect Repository IDs. If pragma directives are used to set the version of an interface, the version number also becomes embedded in the string format of an object reference. A client must bind to a server object whose interface has a matching version number. If the IDL interface on the server side has no version, `_bind()` does not require matching versions. In the present implementation of the Interface Repository, you should use only one version number per Interface Repository.

Part IV

Advanced Orbix C++ Programming

In this part

This part contains the following:

Filtering Operation Calls

Orbix allows you to specify that additional code is to be executed before or after the normal operation or attribute code. This support is provided by allowing applications to create filters that can perform security checks, provide debugging traps or information, maintain an audit trail, and so on. Filters are an Orbix-specific feature.

There are two forms of filters:

- *per-process filters*

  Per-process filters see all operation and attribute calls leaving or entering a client's or server's address space, irrespective of the target object.
- *per-object filters*

  Per-object filters apply to individual objects.

Both of these filter types are illustrated in Figure 22. The sections "Introduction to Per-process Filters" and "Introduction to Per-Object Filters", respectively, give a brief introduction to each. The remainder of the chapter then describes each in detail.

Use of the Dynamic Invocation Interface does not by-pass the filtering mechanism. Calls made using the DII result in the use of all appropriate outgoing and incoming filters.

Note: Because of the Orbix-specific nature of filters, you can only use filters with Orbix.

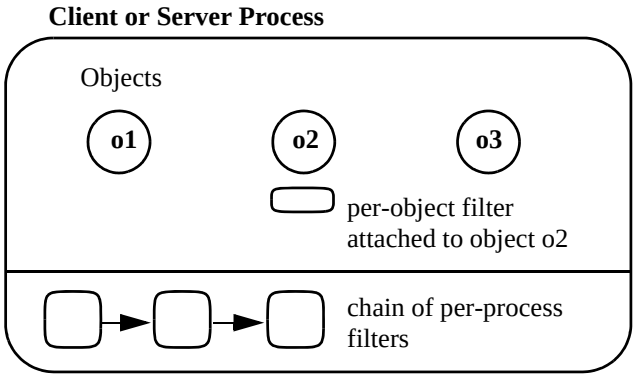


Figure 22: *Per-process and Per-object Filtering*

Introduction to Per-process Filters

Per-process filters monitor all incoming and outgoing operation and attribute requests to and from an address space. Each process can have a chain of such filters, with each element of the chain performing its own actions. You can add a new element to the chain by carrying out the following two steps:

- Define a class that inherits from class `CORBA::Filter`.
- Create a single instance of the new class.

Each filter of the chain can monitor *ten* individual points during the transmission and reception of an operation or attribute request. Refer to Figure 23 on page 236.

Pre-marshalling Filter Points

The four most commonly used filter points are as follows:

- *out request pre-marshal* (in the caller's address space)

  This filter monitors the point before an operation or attribute request is transmitted from the filter's address space to any object in another address space. Specifically, it monitors the point before the operation's parameters have been added to the request packet.
- *in request pre-marshal* (in the target object's address space)

  This filter monitors the point where an operation or attribute request has arrived at the filter's address space, but before it has been processed. Specifically, it monitors the point before the operation has been sent to the target object and before the operation's parameters have been removed from the request packet.
- *out reply pre-marshal* (in the target object's address space)

  This filter monitors the point after the operation or attribute request has been processed by the target object, but before the result has been transmitted to the caller's address space. Specifically, it monitors the point before an operation's out parameters and return value have been added to the reply packet.
- *in reply pre-marshal* (in the caller's address space)

  This filter monitors the point after the result of an operation or attribute request has arrived at the filter's address space, but before the result has been processed. Specifically, it monitors the point before an operation's return parameters and return value have been removed from the reply packet.

Post-marshalling Filter Points

There are four similar post-marshalling monitor points:

- *out request post-marshal* (in the caller's address space)

  This filter operates the same way as 'out request pre-marshal' but after the operation's parameters have been added to the request packet.
- *in request post-marshal* (in the target object's address space)

  This filter operates the same way as 'in request pre-marshal' but after the operation's parameters have been removed from the request packet.
- *out reply post-marshal* (in the target object's address space)

  This filter operates the same way as 'out reply pre-marshal' but after the operation's out parameters and return value have been added to the reply packet.
- *in reply post-marshal* (in the caller's address space)

  This operates the same as 'in reply pre-marshal' but after the operation's out parameters and return value have been removed from the reply packet.

Failure Points

Two additional monitor points deal with exceptional conditions:

- *out reply failure* (in the target object's address space)

  This is called if the target object raises an exception or if any preceding filter point ('in request' or 'out reply') raises an exception or uses its return value to indicate that the call should not be processed any further.
- *in reply failure* (in the caller's address space)

  This is called if the target object raises an exception or if any preceding filter point ('out request', 'in request', 'out reply' or 'in reply') raises an exception or uses its return value to indicate that the call should not be processed any further.

Once an exception is raised or a filter point uses its return value to indicate that the call should not be processed further, no further monitor points (other than the two failure monitor points) are called. If this occurs in the caller's address space, in reply failure is called. If it occurs in the target object's address space, out reply failure and in reply failure are both called (in the target object's and the caller's address spaces respectively).

All monitor points (eight marshalling points and two failure points) are shown in Figure 23.

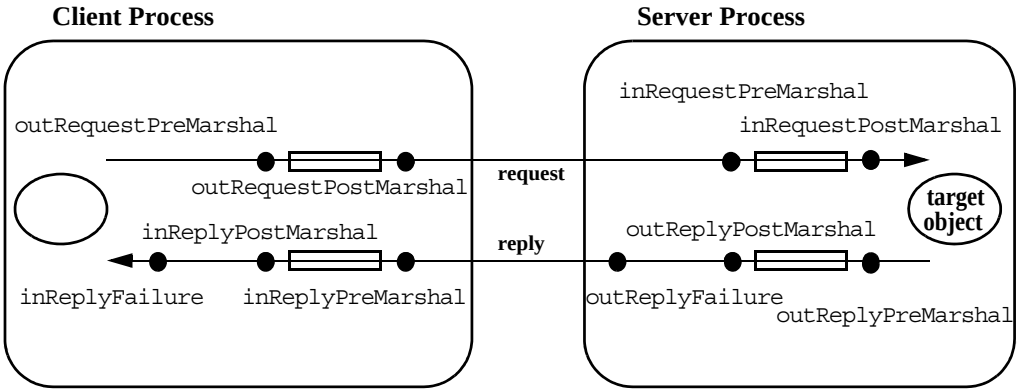


Figure 23: *Per-process Monitor Points*

A particular filter on the per-process filter chain may perform actions for any number of these filter points, although it is common to handle four filter points, for example:

- out request pre-marshal
- out reply pre-marshal
- in request pre-marshal
- in reply pre-marshal

In addition to monitoring incoming and outgoing requests, a filter on the client side and a filter on the server side can co-operate to pass data between them, in addition to the normal parameters of an operation (or attribute) call. For example, the 'out' filter points of a filter in the client can be used to insert extra data into the request package (for example, in 'out request pre-marshal'); and the 'in' filter points of a filter in the server can be used to extract this data (for example, in 'in request pre-marshal').

Each filter point must indicate how the handling of the request should be continued once the filter point itself has completed. In particular, a filter point can determine whether or not Orbix is to continue to process the request or to return an exception to the caller.

Because per-process filters are applied only when an invocation leaves or arrives at an address space, they are not informed of invocations between collocated objects.

Example Usages of Per-Process Filter

In addition, there are two special forms of per-process filters, each with its own special use:

Authentication filter	A filter that passes authentication information from a client to a server. The ability to verify the identify of the caller is a fundamental requirement for security. Authentication filters are discussed in "Defining an Authentication Filter" on page 244.
Thread filter	A filter that (optionally) creates lightweight threads when an operation invocation arrives at a server. The filter point `inRequestPreMarshal()` actually creates the thread. These filters are available in Orbix-MT only, refer to "Using Threads with Orbix" for details.

Introduction to Per-Object Filters

Per-object filters are associated with a *particular* object, rather than with *all* objects in an address space as in per-process filtering. Unlike per-process filters, per-object filters apply even for intra-process operation requests.

The following filtering points are supported:

- *per-object pre*

  This filter applies to operation invocations on a particular object—before they are passed to the target object.

- *per-object post*

  This filter applies to operation invocations on a particular object—after they have been processed by the target object.

A per-object pre-filter can indicate, by raising an exception, that the actual operation call should not be passed to the target object.

Per-object filters are created by carrying out the following three steps:

1. Define a new class that implements all of the IDL operations and attributes for the target object.
2. Create an instance of this new class. This instance behaves as a per-object filter when it is installed.
3. Install this filter object as either a pre-filter or as a post-filter to a particular target object.

It is important to realize that a per-object filter is either a pre-filter or a post-filter. In contrast, a single per-process filter can perform actions for any or all of its eight monitor points.

An object can have a chain of pre-filters and/or a chain of post-filters. For example, a chain of pre-filters can be constructed by attaching a pre-filter to the object, then attaching a pre-filter to that filter, and so on.

Note that per-object filtering can only be used if:

- The TIE approach has been used to associate the target object's class with its IDL C++ class.
- Per-object filtering was enabled when the corresponding IDL interface was compiled by the IDL compiler (see "IDL Compiler Switch to Enable Object Filtering" on page 246).

The parameters to an IDL operation request are readily available for both pre and post per-object filters. Any `in` and `inout` parameters are valid for *pre* filters; `in`, `out` and `inout` parameters and return values are valid for *post* filters. In contrast, for per-process filters, parameters to the operation request are not in general available.

The per-process 'in request' (both pre and post-marshal) filters are applied before any per-object pre-filter. The per-object post-filters are applied before any per-process 'out reply' (both pre and post-marshal) filters.

Using Per-Process Filters

A per-process filter is installed by defining a derived class of class `CORBA::Filter`, and re-defining one or more of its member functions:

`outRequestPreMarshal()`	Operates in the caller's filter before outgoing requests (before marshalling).
`outRequestPostMarshal()`	Operates in the caller's filter before outgoing requests (after marshalling).
`inRequestPreMarshal()`	Operates in the receiver's filter before incoming requests (before marshalling).
`inRequestPostMarshal()`	Operates in the receiver's filter before incoming requests (after marshalling).
`outReplyPreMarshal()`	Operates in the receiver's filter before outgoing replies (before marshalling).
`outReplyPostMarshal()`	Operates in the receiver's filter before outgoing replies (after marshalling).
`inReplyPreMarshal()`	Operates in the caller's filter before incoming replies (before marshalling).
`inReplyPostMarshal()`	Operates in the caller's filter before incoming replies (after marshalling).
`outReplyFailure()`	Operates in the receiver's filter if a preceding filter point raises an exception or indicates that the call should not be processed further or if the target object raises an exception.
`inReplyFailure()`	Operates in the caller's address space if the target object raises an exception or a preceding filter point raises an exception or indicates that the call should not be processed further.

Each of these member functions takes two parameters; the marshalling member functions (the functions not concerned with failure) return a `CORBA::Boolean` value to indicate whether or not Orbix should continue to make the request. (`inRequestPreMarshal()` returns an int value.)

For example:

```
CORBA::Boolean
    outRequestPreMarshal(CORBA::Request& r,
                         CORBA::Environment&);
```

The failure functions, `outReplyFailure()` and `inReplyFailure()`, have a void return type.

You can obtain the details of the request being made by calling member functions on the `CORBA::Request` parameter. Examples of this are shown in "An Example Per-Process Filter" on page 240. The `CORBA::Environment` variable can be used to raise an exception if the C++ compiler does not support native exceptions. Refer to the ***Orbix Programmer's Reference C++ Edition*** for full details of these functions.

The constructor of class `Filter` adds the newly created filter object into the per-process filter chain. Direct instances of `Filter` cannot be created (the constructor is `protected` to enforce this). Derived classes of `Filter` normally have `public` constructors.

Note:

Each function (except the two failure functions) returns a value to indicate how the call should continue. Redefinitions of these functions in a derived class should retain the same semantics for the return value as specified in the relevant entries in the ***Orbix Programmer's Reference C++ Edition***.

You should define derived classes of `Filter` and redefine some subset of the member functions to carry out the required filtering. If any of the non-failure monitoring functions is not redefined in a derived class of `CORBA::Filter`, then the following implementation is inherited in all cases:

```
// C++
{ return 1; } // Continue the call.
```

The failure filter functions inherit the following implementation:

```
// C++
{ return; }
```

Note that the two 'out reply' marshalling filter points are used only if the operation request is issued to the target object. The two 'in reply' marshalling filter points are used only if the operation request is sent out of the caller's address space.

An Example Per-Process Filter

Consider the following simple example of a per-process filter:

```
// C++
#include <CORBA.h>
#include <iostream.h>

class ProcessFilter :
    public virtual CORBA::Filter {
public:
    CORBA::Boolean
    outRequestPreMarshal(CORBA::Request& r,
              CORBA::Environment&) {
        CORBA::String_var s;
        s = (r.target())->_object_to_string();
        cout << endl << "Request out-going to "
             << s << " with operation name "
             << r.operation() << endl;
        return 1; // Continue the call.
    }
    int inRequestPreMarshal(CORBA::Request& r,
            CORBA::Environment&) {
        CORBA::String_var s;
        s = (r.target())->_object_to_string();
        cout << endl << "Request incoming to "
             << s << " with operation name "
             << r.operation() << endl;
        return 1; // Continue the call.
    }

    CORBA::Boolean outReplyPreMarshal(
            CORBA::Request& r,
            CORBA::Environment&) {
        cout << " Incoming operation "
             << r.operation()
             << " finished. " << endl << endl;
        return 1; // Continue the call.
    }

    CORBA::Boolean inReplyPreMarshal(
            CORBA::Request& r,
            CORBA::Environment&) {
        cout << "Outgoing " << r.operation()
             << "finished." << endl << endl;
        return 1; // Continue the call.
    }

    void outReplyFailure(
            CORBA::Request& r,
            CORBA::Environment&) {
        cout << "Operation" << r.operation()
             << "raised exception." << endl << endl;
        return;
    }

    void inReplyFailure(
            CORBA::Request& r,
            CORBA::Environment&) {
        cout << "Operation" << r.operation()
```

```
            << "raised exception." << endl << endl;
        return;
    }
};
```

Filter classes can have any name; but they must inherit from `CORBA::Filter`. `CORBA::Filter` has a protected default constructor; `ProcessFilter` is given a default (no parameter) `public` constructor by C++.

The function `target()` can be applied to a `Request` to find the object reference of the target object; and the function `_object_to_string()` can be applied to an object reference to get a string form of an object reference. The function `operation()` can be applied to a `Request` to find the name of the operation being called.

Installing a Per-Process Filter

To install this per-process filter, you need only create an instance of it, usually at the file level:

```
// C++
ProcessFilter myFilter;
```

This object automatically adds itself to the per-process filter chain.

Raising an Exception in a Filter

Any of the per-process filter points can raise an exception in the normal manner. For example, the `inRequestPostMarshal()` filter point can be changed to raise a `NO_PERMISSION` system exception:

```
// C++
CORBA::Boolean
ProcessFilter::inRequestPostMarshal(
        CORBA::Request& r,
        CORBA::Environment& env) {
    if (.....) {
        throw CORBA::NO_PERMISSION(
                CORBA::INVOKE_DENIED,
                CORBA::COMPLETED_NO);
        // The NO_PERMISSION system exception
        // has been  raised here, with a minor
        // code of INVOKE_DENIED, and a
        // completion status of COMPLETED_NO.
    }
    ...
}
```

Rules for Raising an Exception

The following rules apply when a filter point raises an exception:

- Per-process filters can raise only system exceptions. Any such exception is propagated by Orbix back to the caller. However, raising an exception in an `inReplyPostMarshal()` filter point does not cause the exception to be propagated: at that stage, the invocation is essentially already completed and it is too late to raise an exception.

- If any filter point raises an exception, then no further filter points are processed for that invocation, except for one or both of the failure filter points, `outReplyFailure()` and `inReplyFailure()`.
- If one of the following filter points:

  ```
  outRequestPreMarshal()
  outRequestPostMarshal()
  inRequestPreMarshal()
  inRequestPostMarshal()
  ```

  raises an exception then the actual function call is not forwarded to the target application object.
- If the operation implementation raises a *user exception* and one of the filter points

  ```
  outReplyFailure()
  inReplyFailure()
  ```

  raises a system exception, the system exception is raised in the calling client (that is, the user exception is overwritten). You may wish to test whether an exception has already been raised before raising one in the filter. You can do this by testing the environment formal variable, for example `env`, as follows:

  ```
  // C++
  if (env.exception()) {
      // Have exception already.
  }
  ```
- If the operation implementation raises a *system exception*, no further filter points, except one or both of `outReplyFailure()` and `inReplyFailure()` are called for this invocation.

Piggybacking Extra Data to the Request Buffer

One of the `outRequest` filter points in a client can add extra piggybacked data to an outgoing request buffer—and this data is then made available to the corresponding `inRequest` filter point on the server side. In addition, one of the 'out reply' marshalling filter points on a server can add data to an outgoing reply. This data is then made available to the corresponding `inReply` filter point on the client-side.

At each of the four 'out' marshalling monitor points, you can add data by using `operator<<()` on the `Request` parameter, for example:

```
// C++
CORBA::Long l = 27L;
// . . .
r << l;
```

This is the same `operator<<()` that is used in the DII. Refer to the chapter "Dynamic Invocation Interface" on page 185 for details.

At each of the 'in' marshalling monitor points, data can be extracted using `operator>>()`, for example,

```
// C++
CORBA::Long j;
// . . .
r >> j;
```

This is a fundamental difference from the normal use of `operator>>()` for the Dynamic Invocation Interface, as described in "Dynamic Invocation Interface". In the dynamic invocation interface, `operator>>()` is *only* used to determine the return value of an invocation. In particular, `inout` and `out` parameters are not obtained using `operator>>()`, but their values are instead established using the `outMode` and `inoutMode` stream manipulators. In contrast here, `operator>>()` can be used to extract piggybacked data from an incoming request (or reply).

Matching Insertion and Extraction Points

You must ensure that the insertion and extraction points match correctly, as follows:

Insertion Point	Extraction Point
`outRequestPreMarshal()`	`inRequestPreMarshal()`
`outReplyPreMarshal()`	`inReplyPreMarshal()`
`outRequestPostMarshal()`	`inRequestPostMarshal()`
`outReplyPostMarshal()`	`inReplyPostMarshal()`

For example, a value inserted by `outRequestPreMarshal()` must be extracted by `inRequestPreMarshal()`. Unmatched insertions and extractions corrupt the request buffer and potentially cause a program crash.

When only one filter is being used, its `outRequestPostMarshal()` function can add piggybacked data that the corresponding `inRequestPostMarshal()` function, on the called side, does not remove. However, this would cause problems if more than one filter is used.

Ensuring that Unexpected Extra Data is not Passed

When coding a filter that adds extra data to the request, care should be taken not to include this data when communicating with a server that does not expect it. Frequently, a filter should add extra data only if the target object is in one of an expected set of servers.

For example, it is necessary to include the following code in `outRequestPreMarshal()` and `outRequestPostMarshal()` (assuming the `Request` parameter is `r`):

```
// C++
// First find the server name.
CORBA::ImplementationDef_ptr impl;
impl = (r.target())->get_implementation();

if (strcmp(impl, "some_server") == 0) {
    // Can add extra data.
}
else {
    // Do not add any extra data.
}
```

The function `CORBA::Object::_get_implementation()` returns the server name of an object reference (in this case, of the target object).

You should be particularly careful not to add data when communicating with the Orbix daemon, `IT_daemon`. The Orbix library communicates with the daemon process, and you should ensure that you do not pass extra data to the daemon.

Defining an Authentication Filter

Verification of the identity of the caller of an operation is a fundamental component of a protection system. Orbix supports this by installing an authentication filter in every process's filter chain. This default implementation transmits the name of the principal (user name) to the server when the channel between the client and the server is first established and adds it to all requests at the server side. A server object can obtain the user name of the caller by calling the function:

```
// C++
// In class CORBA::BOA.
char* get_principal(Object_ptr,
    CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
```

on the `CORBA::Orbix` object. The first parameter, of type `Object_ptr`, is not used.

The default authentication filter can be overridden by declaring a derived class of `CORBA::AuthenticationFilter` and creating an instance of this class. For example, an alternative authentication filter may use a ticket-based authentication system rather than passing the caller's user name.

Using Per-Object Filters

You can attach a pre and/or a post per-object filter to an individual object of a given IDL C++ class. Consider the following IDL interface:

```
// IDL
interface Inc {
    unsigned long increment(in unsigned long vin);
};
```

This maps to the following C++ class:

```
// C++
class Inc : public virtual CORBA::Object {
    virtual CORBA::ULong increment(
        CORBA::ULong vin,
        CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
};
```

You can implement interface `Inc` as follows:

```
// C++
class IncImpl {
    virtual CORBA::ULong increment(
        CORBA::ULong vin,
        CORBA::Environment&)
    { return (vin+1); }
};

DEF_TIE_Inc(IncImpl);
```

Note: To facilitate object-level filtering, you must use the TIE approach.

If you have two objects of this type created, as follows:

```
// C++
Inc_ptr i1 = new TIE_Inc(IncImpl) (new IncImpl());
Inc_ptr i2 = new TIE_Inc(IncImpl) (new IncImpl());
```

you may wish to have pre and/or post-filtering on; for example, the specific object referenced by i1. To achieve this, you must define one or more additional classes and additional TIE classes.

To perform pre-filtering, you could define a class (for example, FilterPre) to have the same functions and parameters as a normal implementation class of the corresponding IDL C++ class:

```
// C++
class FilterPre {
public:
    virtual CORBA::ULong increment(
        CORBA::ULong vin,
        CORBA::Environment&) {
            cout << "*** PRE call with parameter "
                 << vin << endl;
            return 0; // Here any value will do.
    }
};
```

Similarly, to perform post-filtering, you could define a class (for example, FilterPost) as follows:

```
// C++
class FilterPost {
public:
    virtual CORBA::ULong increment(
            CORBA::ULong vin,
            CORBA::Environment&) {
                cout << "*** POST call with parameter "
                     << vin << endl;
                return 0; // Any value will do.
        }
};
```

In the examples shown here, a per-object filter cannot access the object it is filtering. A filter can however do this by having a member variable that points to the object it is filtering. You can set up this member using a constructor parameter for the filter.

You need to create TIE classes for these filters:

```
// C++
DEF_TIE_Inc(FilterPre)
DEF_TIE_Inc(FilterPost)
```

To apply filters to a specific object, do the following:

```
// C++
// Create two filter objects.
Inc_ptr serverPre = new TIE_Inc(FilterPre)
           (new FilterPre());
Inc_ptr serverPost = new TIE_Inc(FilterPost)
           (new FilterPost());

// Attach the two filter objects to
// the target object pointed to by i1.
i1->_attachPre(serverPre);
i1->_attachPost(serverPost);
```

It is not always necessary to attach both a pre and a post-filter to an object.

Attaching a pre-filter to an object that already has a pre-filter causes the old filter to be removed and the new one to be attached. The same applies to a post filter.

The functions `_attachPre()` and `_attachPost()` return, respectively, the previous pre-filter and post-filter, if any, attached to the object. The functions `_getPre()` and `_getPost()` return a pointer to an object's pre-filter and post-filter, respectively.

To attach a chain of per-object pre-filters to an object, `_attachPre()` can be used to attach the first pre-filter, and then it can be used again to attach a pre-filter to the first pre-filter, and so on. The same applies to post-filters.

If a per-object pre-filter raises an exception in the normal way, the actual operation call is not made. Normally, this exception is returned to the client to indicate the outcome of the invocation. However, if the pre-filter raises the exception `CORBA::FILTER_SUPPRESS`, no exception is returned to the caller—the caller cannot tell that the operation call has not been processed as normal.

You can raise a `FILTER_SUPPRESS` exception as follows:

```
// C++
throw CORBA::FILTER_SUPPRESS(
    CORBA::FILTER_SUPPRESS_IND,
    CORBA::COMPLETED_NO);
```

In the preceding example, the same filter objects (those pointed to by `serverPre` and `serverPost`) could be used to filter invocations to many objects. Other filters, for example a filter holding a pointer to the object it is filtering, can only be used to filter one object.

IDL Compiler Switch to Enable Object Filtering

Per-object filtering can be applied to an IDL interface only if it has been compiled with the `-F` switch to the IDL compiler. By default, `-F` is not set, so object level filtering is not enabled.

Using Smart Proxy Classes

Smart Proxies are an Orbix-specific feature that allow you to implement proxy classes manually, thereby enabling optimization of client interaction with remote services. This chapter describes how proxy objects are generated, and the general steps involved in implementing smart proxy support for a given interface. It also describes how to build a simple smart proxy, using an example that builds on the BankSimple example from the chapters "Developing Applications with Orbix" and "Using and Implementing IDL Interfaces".

It is sometimes beneficial to be able to implement proxy classes manually. Although it is not expected that you do this if you are a client programmer calling a remote interface; it is a useful option if you are implementing an interface. You can provide *smart proxy* code, for example, to optimize how your clients use the services provided.

A typical example would be to use a smart proxy to examine client requests bound for server objects. The smart proxy forwards requests only if certain criteria are met. The example used in this chapter uses a smart proxy to check if there is sufficient funds in the account before forwarding the request to the server.

This involves constructing a *smart proxy* for the `Account` interface. You can do this by manually programming a class derived from the IDL C++ class `Account` (generated by the IDL compiler). This inheritance is best considered in terms of inheriting from the default proxy class generated by the IDL compiler.

In fact, the IDL C++ class and the default proxy class are the same class, created for every application that you write. The member functions of class `Account` package requests for the target object; the member functions of a derived class can provide optimized application specific coding.

You can then link client programs with this smart proxy code, but you do not have to change them in any other way. When a proxy is created in a client's address space, a smart proxy is created rather than a default one.

Note: Use of the Dynamic Invocation Interface currently bypasses the smart proxy mechanism. Calls made using the DII do not result in invocations on smart proxies.

This chapter first considers the details of how proxy objects are actually generated, and the general steps needed to implement smart proxy support for a given interface. It then describes how to build a simple smart proxy using an example.

The source code for the example described in this chapter is available in the `demos\banksmartproxy` directory of your Orbix installation.

Management of Proxies by Proxy Factories

This section begins describing how Orbix manages proxies. The Orbix IDL compiler generates the following classes for each IDL interface:

- An IDL C++ class—this is also the *default proxy class*.
- A *default proxy factory class* for that class.

The Default Proxy Class

The default proxy class gives the code for standard proxies for that IDL interface. This proxy transmits requests to its real object and returns the results it receives to the caller.

The Default Proxy Factory Class

The proxy factory class produced by the IDL compiler creates these standard proxies for its class, and there is a single global instance of this class linked into the client code. This instance constructs a new standard proxy for its IDL interface when requested by Orbix. The proxy factory class is termed the *default proxy factory class.*

For example, the IDL compiler generates the following classes for IDL interface `BankSimple::Account`:

- `BankSimple::Account`

  This is the IDL C++ class—it also acts as the default proxy class.
- `BankSimple::AccountProxyFactoryClass`

  This is the default proxy factory class for interface `Account`.
- `BankSimple::AccountProxyFactory`

  This is the single global instance of `AccountProxyFactoryClass`.

Generating Smart Proxies

To provide smart proxies for an IDL interface you must:

1. Define the smart proxy class.

   This must inherit from its IDL C++ class.
2. Define a proxy factory class.

   This creates instances of the smart proxy class on request using
   its `New()` member function.
3. Create a single instance of the proxy factory class.

Client programs must be linked with the smart proxy class and the proxy factory class, and must create the instance of the proxy factory class. You should normally provide a header file and a corresponding object file to carry out all of these steps. This involves very minimal changes for clients—their normal operation invocation code remains unchanged.

When these steps are carried out, Orbix communicates with the factory whenever it needs to create a proxy of that interface as follows:

- When a reference to an object of that interface is passed back as an `out` or `inout` parameter or a return value, or when a reference to a remote object enters an address space via an `in` parameter.
- When the interface's `_bind()` function is called.
- When `CORBA::Orbix.string_to_object()` is called with a stringified object reference for a proxy of that interface.

Creating a Smart Proxy

The following steps describe in more detail the steps you must perform to create a smart proxy:

1. Declare and implement the smart proxy class, derived from its IDL C++ class. The constructor of this class is used by the smart proxy factory, in step 2.
2. Declare and implement a new proxy factory class, derived from the default proxy factory class.

   Orbix calls a proxy factory's `New()` member function when it wishes to create a proxy for a particular interface. The a new proxy factory class should redefine the `New()` function to create new smart proxy objects from the class in step 1. Alternatively, it should return zero to indicate that it is not willing to create a smart proxy.
3. Declare a global object of this new class.

   The constructor of the base class automatically registers this new proxy factory object with the factory manager in Orbix.

Smart Proxy Factory Chains

You can define more than one smart proxy class (and associated smart proxy factory class) for a given IDL interface. Orbix maintains a linked list of all of the proxy factories for a given IDL interface.

A chain of smart proxy factories is allowed for an IDL interface because the same IDL interface might be provided by a number of different servers in the system. It may be useful to have different smart proxy code to handle each server, or set of servers.

Each factory in turn can examine the marker and server name of the target object for which the proxy is to be created, and decide whether to create a smart proxy for it or to defer the request to the next proxy factory in the chain. Initially, there is a single entry in this list—the default proxy factory class.

When a new proxy is required, Orbix calls all of the registered proxy factories for the class until one of them successfully builds a new proxy. The only guarantee on the order of use of smart proxy factories is that the factory manager ensures that an interface's default proxy factory object is the last factory on the chain. Thus if no other proxy factory is willing to manufacture a new proxy, a standard proxy is constructed.

The factory manager requests each proxy factory to manufacture a new proxy via its `New()` member function. The first parameter to this function is the full object reference of the target object:

```
// C++
// Returns a pointer to the new smart proxy:
void* New(const char*, CORBA::Environment&)
```

The code for this function may need to extract the target object's marker. One way to extract the target object's marker and server name is by constructing a direct occurrence of `CORBA::Object`, passing the full object reference string as a constructor parameter, and then calling `_marker()` and `_implementation()` on that temporary object.

The `New()` function can raise an exception, in the normal way. If the function returns zero, but does not raise an exception, Orbix tries the next smart proxy factory in the chain.

A Simple Smart Proxy Example

This section describes a simple smart proxy class for interface `Account`, based on the `BankSimple` example.

The Account IDL Interface

The `BankSimple` IDL interface for `Account` is as follows:

```
// IDL
// In file banksimple.idl

...
module BankSimple {
    typedef float CashAmount;
    ...
    interface Account {
        readonly attribute string name;
        readonly attribute CashAmount balance;
        void deposit(in float f);
        void withdraw(in float f);
    }
};
```

Defining a New Proxy Class

This section defines a smart proxy class, named `SmartAccount`, for class `Account`. `SmartAccount` objects check if the client has sufficient funds before the `withdraw()` operation reaches the server:

```
    // C++
    // In file banksimple_smartaccount.h

    #include "banksimple.hh"

1   class SmartAccount : public virtual
      BankSimple::Account {

    public:
        // The required constructor:
2           SmartAccount(const char*);

        // Functions for IDL operations and attributes.
        // List only those which require a different
        // implementation in the smart proxy class:
3           virtual void withdraw(
                BankSimple::CashAmountamount,
                CORBA::Environment& IT_env =
                CORBA::IT_chooseDefaultEnv());
    };
```

This code is described as follows:

1. Class `SmartAccount` inherits from the default proxy code (the IDL C++ class) generated by the IDL compiler. It therefore inherits all of the code required to make a remote invocation. Each `SmartAccount` function can call its base class function to make a remote call. Virtual inheritance is not strictly necessary in the previous code sample. It is used in case C++ multiple inheritance is required later.
2. The constructor for the smart proxy class takes a full object reference string as a parameter. It must pass this to the constructor of its default proxy class.
3. The `withdraw()` function is overridden because of the extra check performed by the smart proxy to see if there are sufficient funds in the account.

The corresponding implementation of the `withdraw()` function is as follows:

```
    // C++
    // In file banksimple_smartaccount.cxx

    #include "banksimple_smartaccount.h"
    #include <iostream.h>

    // Constructor.
    SmartAccount::SmartAccount(const char* OR)
        : BankSimple::Account(OR){}

    // Implementation of the withdraw() function.
    void SmartAccount::withdraw(
        BankSimple::CashAmount amount, CORBA::Environment&)
    {
        float balance;
        ...
```

```
        // Check the account balance.
        balance = BankSimple::Account::balance();
        ...
        if ((balance-amount) < 0) {
            cout <<"Smart Proxy detected insufficient funds
                      for withdraw operation."<<endl;
            cout <<"Withdraw operation not called"<<endl;
            return;
        }
        ...
        BankSimple::Account::withdraw(amount);
        ...
    }
```

The `SmartAccount()` constructor calls the constructor of the IDL C++ class for which it implements proxies (`Account`), passing it the string form of the object reference for the remote object. This call is necessary because the constructor of `Account` in turn calls the constructor of its base class `CORBA::Object`, registering the proxy in the object table. The object table registers all objects in an address space. Refer to `CORBA::ORB::resizeObjectTable()` in the ***Orbix Programmer's Reference C++ Edition*** for more details.

The functions inherited from `Account` are used to make remote calls.

Defining a New Proxy Factory Class

The next step is to define a new proxy factory to generate smart proxies when required. The default proxy factory produced by the IDL compiler for interface `Account` is `AccountProxyFactoryClass`—you should derive from this class, as the following code shows:

```
// C++
// In file banksimple_smartaccount.h

#include "banksimple.hh"
...
class SmartAccountFactoryClass : public virtual
    BankSimple::AccountProxyFactoryClass {

public:
    // Constructor:
    SmartAccountFactoryClass() : CORBA::ProxyFactory(
        BankSimple_Account_IR) {}

    // The New() member function is called when a
    // proxy is required.
    virtual void* New (
        const char* OR,
        CORBA::Environment&) {
        // Create and return a new smart proxy.
        return(BankSimple::Account_ptr)
            new SmartAccount(OR);
    }
};
```

The member initialization list of the constructor of class `SmartAccountFactoryClass` makes a call to the constructor of class `ProxyFactory` (the base class of `AccountProxyFactoryClass`). The parameter passed is `BankSimple_Account_IR`, an automatically-defined macro.

Each default proxy factory class has a default constructor without any parameters. The constructor of `SmartAccountFactoryClass` therefore does not need to be concerned with calling the constructor of `AccountProxyFactoryClass`; however, it must call the `ProxyFactory` constructor. Figure 24 shows the various class hierarchies involved.

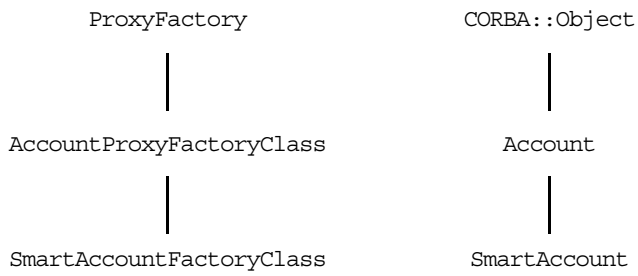


Figure 24: *Class Hierarchy for Smart Proxy Class*

The `SmartAccountFactoryClass::New()` function is called by Orbix to signal that a smart proxy is to be created. Orbix passes it the object reference of the object for which the proxy is required. The `New()` function dynamically constructs the smart proxy, passing it the object reference. Orbix also passes any other constructor parameters agreed on by the smart proxy and the smart proxy factory.

Note:

A member variable, `m_next`, is defined in the default proxy factory class for each interface. This is automatically maintained by Orbix and should not be modified by any factory.

If an `Account` smart proxy factory needs to test whether or not it should create a smart proxy, its `New()` function should do the following:

```
// C++
if (...) // Test condition omitted here.
    // The target object is one that you should
    // create a smart proxy for.
    return (BankSimple::Account_ptr)
        new SmartAccount(OR);
else
    // Pass on the object reference parameter to the
    // next factory in the factory chain.
    return 0;
```

The factory can use the stringified object reference parameter to determine whether it should create a smart proxy: it might determine this from the server name of the object reference, or perhaps by communicating with the object's server. If the request is propagated as far as the default factory, it will create a standard proxy.

In the code for `SmartProxyFactoryClass` above, *all* account proxies are built as smart proxies. We could re-implement `SmartAccountFactoryClass::New()` to build smart proxies only for particular remote servers. To do this, `SmartAccountFactoryClass::New()` should find the server name of the target object to decide whether it should create a smart proxy,

or pass the request to the next factory in the linked list (and finally to the default proxy factory class that constructs a standard proxy).

Declaring a New Proxy Factory Class Instance

Finally, you need to declare a single instance of this new class:

```
// C++
// In file banksimple_smartaccount.cxx
SmartAccountFactoryClass  SmartAccountFactory;
```

The constructor of the base class then registers this new factory—entering it into the linked list of factories for interface `Account`.

Callbacks from Servers to Clients

Orbix clients usually invoke operations on objects in Orbix servers. However, Orbix clients can also implement some of the functionality associated with servers, and all servers can act as clients. This flexibility increases the range of client-server architectures that you can implement with Orbix. This chapter describes a common approach to implementing callbacks in an Orbix application, illustrated by a stock-trading example.

A callback is an operation invocation made from a server to an object implemented in a client. Callbacks allow servers to send information to clients without forcing clients to explicitly request the information.

Implementing Callbacks in Orbix

This chapter introduces a simple model for implementing callbacks in a distributed system. It describes the following steps:

1. Defining the IDL interfaces for the system.
2. Implementing the IDL Interfaces.
3. Writing the client.
4. Writing the server.

Defining the IDL Interfaces

In the stock-trading example, the client invokes operations on the server and the server invokes operations on the client. You must therefore define IDL interfaces that allow each application to access the other. In the simplest case, this involves two interfaces, for example:

```
// IDL
// In file stock.idl

// Implemented by the client.
interface StockInfoCB {
    ...
};

// Implemented by the server.
interface RegStockInfo {
    ...
};
```

In this example, the client application supplies an implementation of type `StockInfoCB`, while the server implements `RegStockInfo`.

The server in this example cannot bind to the client implementation object, because clients are not registered in the Orbix Implementation Repository. Instead, the IDL definition provides a `Register()` operation that allows the client to explicitly pass an implementation object reference to the server.

The full IDL for the stock-trading example is as follows:

```
// IDL
// In file stock.idl

1       interface StockInfoCB {
          oneway void NotifyPriceChange
              (in string stockname, in float newprice);
        };

2           interface RegStockInfo {
            void Register (in StockInfoCB obj);
            void Deregister (in StockInfoCB obj);
3               void Notify (in float newprice);
        };
```

This IDL is described as follows:

1. `StockInfoCB` is the callback interface implemented by the client. Its `NotifyPriceChange()` operation is invoked by the server when a stock price changes.

   `NotifyPriceChange()` is a `oneway` operation. This means that the server calling this operation does not block while the client object processes the call. Orbix does not guarantee that a oneway operation call will succeed; if a oneway operation fails, the client may not know. Refer to "Preventing Deadlock in a Callback Model" for more details.
2. `RegStockInfo` is the register interface implemented by the server. Its `Register()` and `Deregister()` operations enable clients that wish to receive stock price callbacks to register or deregister for a given stock.
3. The `Notify()` operation is used by the server to notify clients of a stock price change. `Notify()` calls the `NotifyPriceChange()` operation.

The source code for the example described in this chapter is available in the Orbix `demos\callback` directory.

Implementing the IDL Interfaces

You can use the BOAImpl or TIE approach to implementing IDL interfaces. Using the BOAImpl approach, the implementation class for type `RegStockInfo` is as follows:

```
// C++
// In file stock_impl.h

#include <it_demo_streams.h>
#include "stock.hh"

// Implementation class
class RegStockInfoImpl : public RegStockInfoBOAImpl {
public:
    // C++ operations
    RegStockInfoImpl(char* initialname);
    ~RegStockInfoImpl();

    // IDL operations
    void Register (
        StockInfoCB_ptr obj,
```

```
            CORBA::Environment& env)
            throw (CORBA::SystemException);
        void Deregister (
            StockInfoCB_ptr obj,
            CORBA::Environment& env)
            throw (CORBA::SystemException);

        void Notify (
            CORBA::Float newprice,
            CORBA::Environment& env)
            throw (CORBA::SystemException);

    protected:
        // A list of all the objects to callback.
        StockInfoCB_ptr clientlist[20];

        CORBA::Long number_clients;
        int max_number_clients;
        char* stockname;
    };
```

The implementation of the `Register()` function for the `RegStockInfo` interface requires special attention:

```
    // C++
    // In file stock_impl.cxx

    // Called by a client wishing to receive
    // stock price callbacks.
    void RegStockInfoImpl::Register (StockInfoCB_ptr obj,
        CORBA::Environment& ) throw
    (CORBA::SystemException) {
        if(number_clients > max_number_clients) {
            cout << "All server connections used for
            callback purposes"
            <<endl;
            return;
        }
        cout << "Registering client for stockname "
                << stockname << endl;
        clientlist[number_clients] =

    StockInfoCB::_duplicate(obj);
        number_clients++;
    }
```

This `Register()` function receives an object reference from the client. When this object reference enters the server address space, a proxy for the client's `StockInfoCB` object is created, as shown in Figure 25 The server uses this proxy to call back to the client. The implementation of `Register()` stores the reference to the proxy for later use.

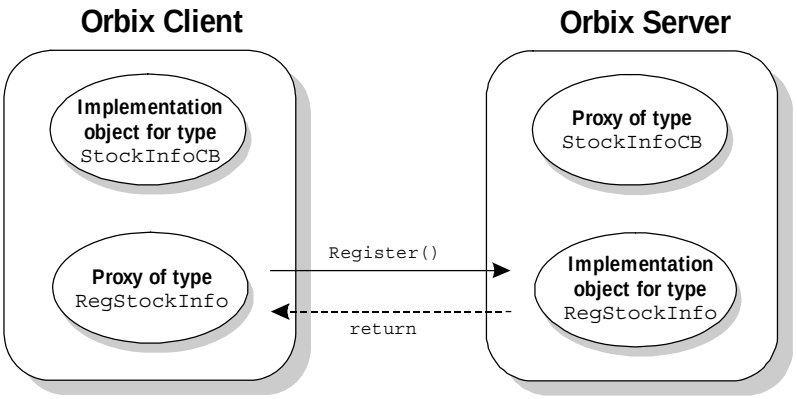


Figure 25: *Client Passes Implementation Object Reference to Server*

Once the server creates the proxy in its address space, it can invoke `NotifyPriceChange()` (using its `Notify()` operation) to respond to a change in a stock price.

The implementation of the `Notify()` function that calls `NotifyPriceChange()` is as follows:

```
// C++
// In file stock_impl.cxx

// Called by the server when sending stock price
// updates (callbacks) to the client.
void RegStockInfoImpl::Notify(
    CORBA::Float newprice,
    CORBA::Environment& )
    throw (CORBA::SystemException) {
    if(number_clients>0) {
        for (int i=0; i < number_clients; i++) {
            try {
                clientlist[i]>NotifyPriceChange(
                                        "IONAY",newprice);
            }
            catch (const CORBA::Exception& e) {
                cerr << "Unexpected exception: "
                        << e << endl;
            }
        }
    }
}
```

The `NotifyPriceChange()` invocation on the `StockInfoCB` proxy is routed to the client implementation object as shown in Figure 26 on page 259.

The transmission of requests from server to client is possible because Orbix maintains an open communications channel between client and server while both processes are alive. The server can send the callback invocation directly to the client and does not need to route it through an Orbix daemon. Therefore, the client can process the callback event without being registered in the Orbix Implementation Repository and without being given a server name.

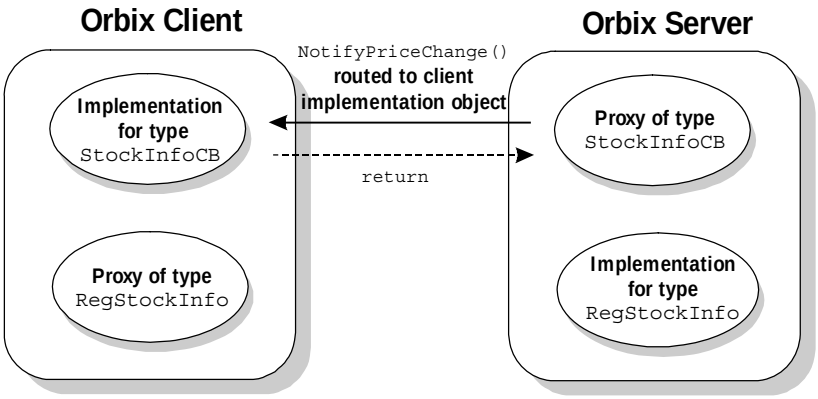


Figure 26: *Server Invoking Operation on Client Callback Object*

Writing the Client

The code for the client `main()` function is as follows:

```
// C++
// In file client.cxx.
...
#include "stock.hh"
#include "callback.h"

int main (int argc, char* argv[]) {
   try {
      // Basic Setup. Process command-line arguments.
      ...
      // ORB Setup - initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "Orbix");
      CORBA::BOA_var boa = orb->BOA_init(argc, argv, "Orbix_BOA");

      // Set the diagnostic level from the options
      orb->setDiagnostics(clientopt.diagnostics());
      // Naming Service Setup
      // Resolve an object using a Naming Service Wrapper (NSW).
      // See demolib/it_demo_nsw.* for details.
      ...

      // Create a Naming Service Wrapper object and define a name
      // prefix used for subsequent operations.
      IT_Demo_NSW ns_wrapper;
      ns_wrapper.setNamePrefix(clientopt.context());

      // Get CORBA object.
      // Provide the object's name in the Naming Service.
      const char* object_name = "CallBack.CallBack";

      // Use the NSW to obtain a reference to the required object.
      CORBA::Object_var obj = ns_wrapper.resolveName(object_name);

      // Narrow the object reference.
      RegStockInfo_var stock = RegStockInfo::_narrow(obj);
      if (CORBA::is_nil(obj)) {
```

```
            cerr << "Object in naming service is not of expected
            type"<< endl;
        }

        // Perform demo-specific operations on the CORBA object.
        StockInfoCB_var cbobj = new StockInfoCBImpl();
        cout << "Sending request for IONAY stock prices"<<endl;
        stock->Register(cbobj);

        // Client is coded to receive ten callbacks.
        for (int i=0; i < 10; i++) {
            boa->processNextEvent();
        }
        cout << "Sending request deregister from IONAY stock price
        callbacks "<<endl;
        stock->Deregister(cbobj);
    }
    catch (const CORBA::Exception& e) {
        cerr << "Unexpected exception" << e << endl;
        cerr << "Demo failed" << endl;
        exit(1);
    }
    cout << "Demo finished" << endl;
    return 0;
}
```

This client creates an implementation object of type `StockInfoCBImpl`. It then invokes the `Register()` function and connects to an object of type `RegStockInfo` in the server. At this point, the client holds an implementation object of type `StockInfoCB` and a proxy for an object of type `RegStockInfo`, as shown in Figure 27.

To allow the server to invoke operations on the `StockInfoCB` implementation object, the client must pass this object reference to the server. Consequently, the client then calls the operation `Register()` on the `RegStockInfo` proxy object, as shown in Figure 27.

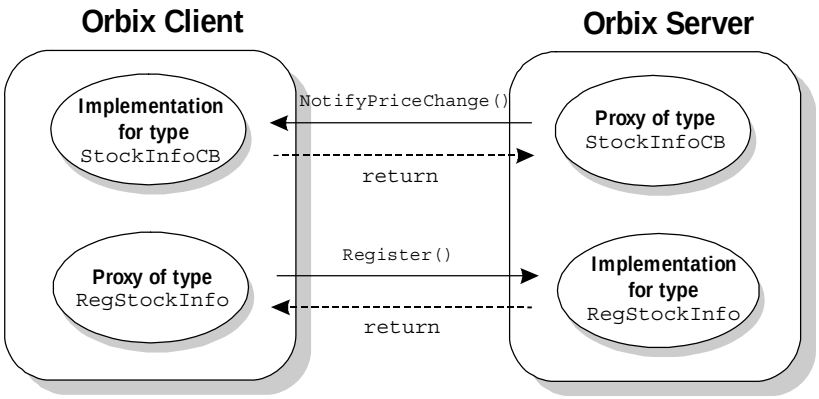


Figure 27: *Client-Server Callback Interaction*

Writing the Server

The code for the server `main()` function is as follows:

```
// C++
// In file server.cxx
...
#include "stock_impl.h"

int main(int argc, char*  argv[]) {
    try {
        ...

        // ORB and BOA setup.
        // Initialize the ORB and BOA
        CORBA::ORB_var orb = CORBA::ORB_init
                                (argc, argv, "Orbix");
        CORBA::BOA_var boa = orb->BOA_init
                                (argc, argv, "Orbix_BOA");

        // Set diagnostics and server name.
        ...

        // Server should not quit while clients
        // are connected.
        boa->setNoHangup(1);

        boa->impl_is_ready
            ((char*)serveropt.server_name(), 0);

        // Create an implementation object.
        RegStockInfo_var symbol
                          = new RegStockInfoImpl("IONAY");

        // Naming Service setup as normal.
        ...

        // ORB/BOA event processing
        // Server completed its initialization,
        // and waiting for incoming requests.
        boa->impl_is_ready
                ((char*)serveropt.server_name(), 0);
        // Set share price.
        float shareprice = 18.5;

        if (!serveropt.bindns()) {
            for (int i=0; i<100;i++) {
                if (boa->isEventPending()) {
                    boa->processNextEvent();
                }
                // Calculate a new stock price
                symbol->Notify(shareprice);

                // Share price increases by 25 cents
                // on each iteration.
                shareprice += .25;
                ...
                // Sleep for 3 seconds.
                Sleep(3000);
            }
```

```
        }
        cout << "server exiting" << endl;
    }
    catch (const CORBA::Exception& e) {
        cerr << "Unexpected exception: " << e << endl;
        return 1;
    }
    return 0;
}
```

This server creates an implementation object of type `RegStockInfo`, and registers this object in the Naming Service using a Naming Service Wrapper. It then sets the share price and notifies the client of share price changes using the `Notify()` operation. This calls the `NotifyPriceChange()` operation, which calls back to the client.

The call to `processNextEvent()` is made in case a client wishes to register or deregister. This call has a zero timeout value. This means that the server is not blocked; the call returns immediatelly if there is no pending event.

The server main thread must either sleep or do other processing to avoid exiting.

Note: You should only invoke `setServerName()` from the server. If a client invokes `setServerName()`, server operations on its callback object will fail to connect.

Preventing Deadlock in a Callback Model

When an application invokes an IDL operation on an Orbix object, by default, the caller is blocked until the operation returns. Deadlock can occur in a single-threaded system where applications can both invoke and implement operations.

For example, in the stock trading application, a simple deadlock can occur if the server attempts to call back to the client in the implementation of the function `Register()`. In this case, the client is blocked on the call to `Register()` when the server invokes `NotifyPriceChange()`. The `NotifyPriceChange()` call blocks the server until the client reaches an event processing call and handles the server request. Each application is blocked, pending the return of the other, as shown in Figure 28 on page 263.

Unfortunately, it is not always possible to design a callback architecture where simultaneous invocations between groups of processes are guaranteed never to occur. However, there are alternative approaches to preventing deadlock in an Orbix system.

The two primary approaches to preventing deadlock are as follows:

- Using non-blocking operation invocations.
- Using a multi-threaded event processing model.

These approaches are discussed in the two subsections that follow.

Using Non-Blocking Operation Invocations

There are two ways to invoke an IDL operation in an Orbix application without blocking the caller:

- Declaring an IDL operation as `oneway`.
- Invoking the operation using the *deferred synchronous* approach supported by the Dynamic Invocation Interface (DII).

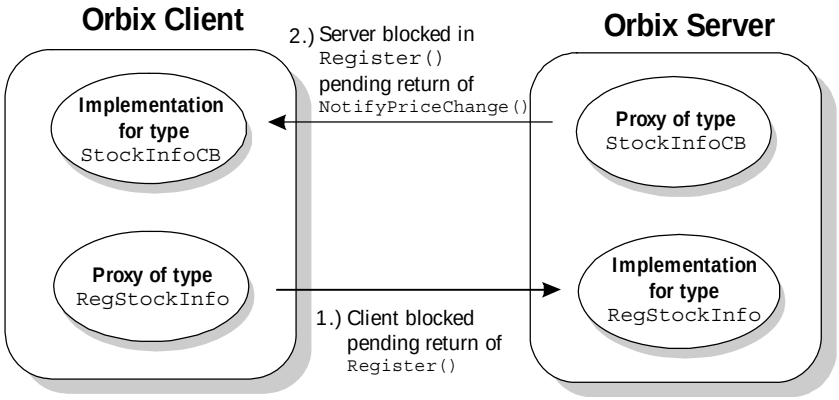


Figure 28: *Deadlock in a Callback Model*

Declaring an IDL Operation as Oneway

You can declare an IDL operation `oneway` only if it has no `out`, or `inout` parameters and no return value. A `oneway` operation can only raise an exception if a local error occurs before a call is transmitted. Consequently, the delivery semantics for a `oneway` request are "best-effort" only. This means that a caller can invoke a `oneway` request and continue processing immediately, but is not guaranteed that the request arrives at the server.

You can avoid deadlock, as shown in Figure 28, by declaring either `Register()` or `NotifyPriceChange()` as a `oneway` operation. The IDL for the stock trading example is as follows:

```
// IDL
interface StockInfoCB {
    oneway void NotifyPriceChange (in String message);
};

interface RegStockInfo {
    void Register (in StockInfoCB objRef);
};
```

In this case, the client's call to `NotifyPriceChange()` returns immediately, without waiting for the server's implementation function call to return. This allows the client to enter the Orbix event processing call. At this point, the callback invocation from the server is processed and routed to the client's implementation of `Register()`. When this function call returns, the server no longer blocks and both applications wait again for incoming events.

Note: Using `oneway` operations does not prevent deadlock in SSL-enabled callback applications because establishing SSL connections requires a response from the client. This problem does not occur for multi-threaded SSL-enabled application. Refer to the ***OrbixSSL C++ Programmer's and Administrator's Guide*** for details.

Using the Deferred Synchronous Approach

You can achieve a similar functionality by using the Orbix DII deferred synchronous approach to invoking operations. As described in the chapter "Dynamic Invocation Interface", the DII allows an application to dynamically construct a function invocation at runtime, by creating a `CORBA::Request` object. You can then send the invocation to the target object using one of a set of functions supported by the DII.

The chapter "Dynamic Skeleton Interface" describes how to call the following functions on an orb to invoke an operation without blocking the caller. If any of the following functions are used, the caller can continue to process in parallel with the target implementation function:

```
CORBA::Request::send_deferred()
CORBA::Request::send_oneway()
CORBA::ORB::send_multiple_requests_deferred()
CORBA::ORB::send_multiple_requests_oneway()
```

Operation results can be retrieved at a later point in the caller's processing, and avoid deadlock as if the operation call is a `oneway` invocation.

Using Multiple Threads of Execution

An Orbix application can create multiple threads of execution. To prevent deadlock, it can be useful to create a separate thread dedicated to handling Orbix events. Refer to "Using Threads with Orbix" for details of how to create threads using Orbix.

If another thread in this application becomes blocked while invoking an operation on a remote object, the event processing continues in parallel. The remote operation can then safely call back to the multi-threaded application without causing deadlock.

Event Processing Functions

Orbix applications can use event processing functions that do not implicitly initialize the application server name. The client can safely call either the function `processEvents()` or the function `processNextEvent()` on the `ORB` object.

These event processing functions are defined on Orbix class `BOA`. If the client is to receive callbacks, the client's `ORB` object must be initialized as type `BOA`. The client call, for example, to, `processEvents()` blocks while waiting for incoming Orbix events. If the server invokes an operation on the `StockInfoCB` object reference forwarded by the client, this call is processed by `processEvents()` and routed to the correct function in the client's implementation object.

Callbacks and Bidirectional Connections

If you use the Orbix protocol, the server sends its callbacks on the same connection that the client initiated and used to make requests on the server. This means that the client does not need to accept an incoming connection.

Standard IIOP, on the other hand, requires that the client accept a connection from the server to allow the callbacks to be sent. Orbix introduces an optional extension to IIOP to allow the protocol to use bidirectional connections. Bidirectional connections allow clients to receive requests from servers on the connection that the client originated to the server.

To configure your client to use bidirectional connections, call `CORBA::ORB::supportBidirectionalIIOP()` with the `on` parameter set to `true`. This is defined as follows:

```
CORBA::Boolean supportBidirectionalIIOP(
    CORBA::Boolean on,
    CORBA::Environment& IT_env =
                CORBA::IT_chooseDefaultEnv());
```

By default, bidirectional IIOP is disabled. Refer to the ***Orbix Progammer's Reference C++*** for more details on `supportBidirectionalIIOP()`.

Loading Objects at Runtime

If a client invokes an IDL operation on an object that does not exist in a server, Orbix returns an exception to the client. However, Orbix also allows server programmers to create loaders that are responsible for instantiating objects in response to client requests. This chapter explains how to use loaders in Orbix, using an example named BankPersistent. This example is based on the BankExceptions example and builds on concepts already introduced.

When an operation invocation arrives at a process, Orbix searches for the target object in the object table for that process. By default, if the object is not found, Orbix returns an exception to the caller. However, if one or more *loader* objects are installed in the process, these are informed about the *object fault* and allowed to load the target object and resume the invocation transparently to the caller. The loaders are C++ objects maintained in a chain, and are tried in turn until one can load the object. If no loader can load the object, an exception is returned to the caller.

Loaders are not just called when a "missing object" is the target of a request; they are also called when an object reference enters an address space. This can arise in the following ways:

- When a call to either `_bind()` or `CORBA::Orbix.string_to_object()` is made from within a process.
- For a server, as an `in` parameter.
- For a client (or a server making a function call), as an `out` or `inout` parameter, or a return value.

The loaders are given an opportunity to respond to such object faults by loading the target object of the reference into the process address space. If no loader can load the referenced object, Orbix constructs a proxy for the object.

Loaders can provide support for persistent objects. These are long-lived objects stored on disk in the file system or in a database.

Overview of Creating a Loader

To create a loader, first define a derived class of `CORBA::LoaderClass`. You can then install a loader by creating a (dynamic) instance of that new class.

`CORBA::LoaderClass` provides the following functions:

- `load()`

  Orbix uses this function to inform a loader of an object fault. The loader is given the marker of the missing object so that it can identify the object to load.

- `save()`

  When a process terminates, its loader(s) can save the objects in its address space. To enable this, Orbix makes an individual call to `save()` for each object managed by that loader. The `save()` function is also called when an object is destroyed.

  You can also explicitly call `save()` using the `CORBA::Object::_save()` function, defined on all Orbix objects.

- `record()` and `rename()`

  These functions are used to enable naming of objects when using loaders. Refer to "Loaders and Object Naming" on page 270 for more details.

For full details of class `CORBA::LoaderClass`, refer to the ***Orbix Programmer's Reference C++ Edition***.

Installing a Loader

You should remember three important points when creating a loader object:

- A loader *must* be created dynamically using `new()` and should *not* be deleted explicitly by application code. Otherwise an error occurs when Orbix tries to delete the loader as the process terminates.

  Static creation of loaders is not supported because of the possibility that C++ may destroy a loader before Orbix calls that loader's `save()` function. This would affect each of the objects it controls.

- If a loader's constructor uses either of the functions `CORBA::ORB::isBaseInterfaceOf()` or `CORBA::ORB::baseInterfacesOf()`, that loader must *not* be created before the first line of the main function. This means that the loader cannot be created directly at the file level or in the constructor of an object created at the file level. Attempts to break this rule could lead to calls on these functions before their underlying data structures are initialized. This depends on the C++ compiler used.

- The constructor of `CORBA::LoaderClass` (the base class of all loaders) takes a `CORBA::Boolean` parameter that must be non-zero if the new loader's `load()` function is to be called by Orbix. The default value of this parameter is `false`.

Specifying a Loader for an Object

Each object is associated with a loader object that is informed when the object is named or renamed and when the object is to be saved. If no loader is explicitly specified for an object, it is associated with a default loader, implemented by Orbix. This loader does not support persistence.

An object's loader can be specified as the object is being created, using either the TIE or the BOAImpl approach.

Using the BOAImpl Approach

In the BOAImpl approach, you can specify a loader for an object by declaring the implementation class constructor to take a pointer to the loader as a parameter. You should then call this constructor, passing on this pointer as a parameter to set the loader for the object. For example:

```
// C++
// In file bankexceptions_bankimpl.h

class BankExceptions_BankImpl : public virtual
BankExceptions::BankBOAImpl {
public:
    // Constructor.
    BankExceptions_BankImpl(
        CORBA::LoaderClass* loader) throw();
    ...
}

// In file bankexceptions_bankimpl.cxx

// BankExceptions_BankImpl constructor call
BankExceptions_BankImpl::BankExceptions_BankImpl(CORBA:
:LoaderClass* loader) throw() :
    m_accounts(
    new BankExceptions::Account_var[MAX_ACCOUNTS]),
    account_loader(loader)
...
```

You can obtain a pointer to an object's loader by calling:

```
// C++
// In class CORBA::Object.
CORBA::LoaderClass* _loader(
        CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
```

Using the TIE Approach

In the TIE approach, you can specify a loader for an object by passing a pointer to the loader as a parameter to the TIE constructor. For example,

```
// C++
// myLoader is a pointer to a loader object.
bank= new TIE_BankExceptions_Bank(
        BankExceptions_BankImpl) (
        new BankExceptions_BankImpl(),// Object pointer
        myLoader);                    // Loader pointer
```

Loaders and Object Naming

When supporting persistent objects, it is often important to control the markers that are assigned to them. For example, you often need to use an object's marker as a key to search for its persistent data. The format of these keys depend on how the loader implements the persistence. Therefore, it is common for loaders to choose object markers, or at least to be allowed to accept or reject markers chosen by application level code.

Naming Objects

The two main ways to name objects when using loaders are as follows:

- Using the constructor.
- Using `_marker()`.

Naming Objects Using the Constructor

Using the BOAImpl approach, you can pass the marker name to the BOAImpl constructor, for example:

```
// C++
// In file bankexceptions_accountimpl.cxx

// BankExceptions_AccountImpl constructor
BankExceptions_AccountImpl::BankExceptions_AccountImpl(
    const char* name,
    BankExceptions::CashAmount balance,
    const char* marker,
    CORBA::LoaderClass* loader) throw() :
        BankExceptions::AccountBOAImpl(marker, loader),
        m_balance(balance), m_name(name) {}
    ...
```

This constructor sets the marker as the account name and sets the loader for the account object.

Naming Objects Using _marker()

`CORBA::Object::_marker(const char*)` sets the target object's marker name; for example:

```
bank->_marker("mybank");
```

Refer to the chapter "Making Objects Available in Orbix" for more details on naming objects in Orbix.

record() and rename()

Regardless of whether `_marker()` or the constructor is used, Orbix calls the object's loader to confirm the chosen name, thereby allowing the loader to override the choice. When using the constructor, Orbix calls `record()`. When using `_marker()` Orbix calls `rename()` because the object already exists.

Orbix executes the following algorithm when an object is created, or an object's existing marker is changed:

- If the specified marker (`char*` pointer) is not null, Orbix checks whether the name is already in use within the process. If it is not in use, the name is suggested to the loader (by calling `record()` or `rename()`). The loader can accept the name by not changing it. Alternatively, the loader can reject it by changing it to a new name. If the loader changes the name, Orbix again checks that the new name is not already in use within the current process; if it is already in use, the object will not be correctly registered.
- If no name is specified or if the specified name is already in use within the current process, Orbix passes a nil `char*` pointer to the loader (by calling `record()` or `rename()`) which must then choose a name. Orbix then checks the chosen name; the object will not be correctly registered if this chosen name is already in use.

If necessary, both `record()` and `rename()` can raise an exception. The implementations of `rename()` and `record()` in `CORBA::LoaderClass` both return without changing the suggested name. The implementation of `load()` and `save()` perform no actions.

The Default Loader

The default loader is associated with all objects that are not explicitly associated with a loader. This is an instance of `CORBA::NullLoaderClass`, a derived class of `CORBA::LoaderClass`. This class inherits `load()`, `save()` and `rename()` from `CORBA::LoaderClass`. It implements `record()` so that if no marker name is suggested it chooses one that is a string of decimal digits, different to any already generated in the current process. The default loader does not support persistence.

Loading Objects

When an object fault occurs, the `load()` function is called on each loader in the chain until one successfully returns the address of the object, or until they all return zero. Orbix cannot call the correct loader directly, because the object does not yet exist in the address space.

The `load()` function performs the following tasks:

- It determines if the required object is to be loaded by the current loader.
- If the required object is to be loaded, it recreates the object and assigns it the correct marker.

The `load()` function is passed the following information:

- The interface name.
- The target object's marker.

- A `CORBA::Boolean` value, set as follows depending on why the object fault occurred:

`1`	Because of a call to `_bind()` or `CORBA::Orbix.string_to_object()` by the process that contains the loader.
`0`	Because of an object fault on the target object of an incoming operation invocation, or on an `in`, `out` or `inout` parameter or return value.

- An `Environment` parameter.

The interface name of the missing object is determined as follows:

- If an object fault occurs because of the following call:

```
p = I1::_bind( <parameters>);
```

  the interface name in `load()` is "`I1`".

  If the first parameter to the `_bind()` is a full object reference string, Orbix returns an exception if the reference's Interface field is not `I1` or a derived interface of `I1`.

  Refer to the entry for `CORBA::ORB::object_to_string()` in the ***Orbix Programmer's Reference C++ Edition*** for details on the string format of Orbix object references.

- If an object fault occurs during the following call:

```
p = CORBA::Orbix.string_to_object
        ( <full object reference string>);
```

  the interface name in `load()` is extracted from the full object reference string.

- If a loader is called because of a reference entering an address space (as an `in`, `out` or `inout` parameter, a return value, or as the target object of an operation call), the interface name in `load()` is the interface name extracted from the object reference.

Saving Objects

When a process terminates, Orbix iterates through all the objects in its object table and calls the `save()` function on the loader associated with each object. A loader may save the object to persistent storage (either by calling a function on the object, or by accessing the object's data and writing this data itself).

The `save()` function is also called on the loader associated with an object when that object is destroyed. You can also explicitly call an object's `_save()` function. The `_save()` function simply calls the `save()` function on the object's loader. You must call `_save()` in the same address space as the target object—calling it in a client process on a proxy has no effect.

The `reason` parameter to `save()` indicates why this function has been called. Its possible values are as follows:

`processTermination`	The process is about to exit.
`explicitCall`	The object's `_save()` function has been called.
`objectDeletion`	`CORBA::release()` has been called on the object, which previously had a reference count of `1`.

If the reason is `objectDeletion`, you would normally code a loader's `save()` function to delete the persistent representation of the object, as follows:

```
// C++
if (reason == CORBA::objectDeletion)
    // Delete the persistent representation
```

On process termination, Orbix does not delete the objects themselves as it iterates through its object table. Instead Orbix calls `save()` on each object's loader. It does, however, destroy the loader objects after they have been used.

Writing a Loader

If you are writing a loader for a specific interface, you would typically perform the following actions:

- Redefine the `load()` function to do the main work of the loader—to load the object on demand. The object's marker is normally used to find the object in the persistent store.
- Redefine the `save()` function so that it saves its objects on process termination, and also if `_save()` is called. This normally deletes an object's persistent storage if the save reason is `objectDeletion`.
- Redefine the `record()` and `rename()` functions. Often, `record()` chooses the marker for a new object; and `rename()` is written to prevent an object's marker being changed. However, `record()` and `rename()` are sometimes not redefined in a simple application, where the code that chooses markers at the application level can be trusted to choose correct values.

Example Loader

This section presents a simple loader example named BankPersistent. This example builds on the BankExceptions example introduced in the chapter "Exception Handling in Orbix". The code used in this example is available in the `demos\bankpersistent` directory of your Orbix installation.

The IDL Interface

This example uses the BankExceptions IDL, as follows:

```
// IDL
// In file bankexceptions.idl

module BankExceptions {
    typedef float CashAmount;
    interface Account;

    interface Bank {
        // User-defined exceptions.
        exception CannotCreate { string reason; };
        exception NoSuchAccount { string name; };

        Account create_account (in string name)
            raises (CannotCreate);
        Account find_account (in string name)
            raises (NoSuchAccount);
    };

    interface Account {
        // User-defined exception.
        exception InsufficientFunds { };

            readonly attribute string name;
        readonly attribute CashAmount balance;

        void deposit (in CashAmount amount);
        void withdraw (in CashAmount amount)
            raises (InsufficientFunds);
    };
};
```

Implementing the IDL

This example uses the BOAImpl approach. Interfaces `Account` and `Bank` are implemented by classes `BankExceptions_AccountImpl` and `BankExceptions_BankImpl`, respectively.

Instances of class `BankExceptions_AccountImpl` are made persistent using a class named `Loader` inheriting from `CORBA::LoaderClass`. A simple persistence mechanism is used, with one file per account object. This section shows the implementation of classes `AccountImpl` and `BankImpl`. The implementation of class `Loader` is shown in "Coding the Loader" on page 278.

Class AccountImpl

Class `AccountImpl` is implemented as follows:

```
// C++
// In file bankexceptions_accountimpl.h
...
#include "bankexceptions.hh"

1   class BankExceptions_AccountImpl : public virtual
              BankExceptions::AccountBOAImpl
    {
    public:
        // IDL operations
        virtual void deposit(BankExceptions::CashAmount amount,
                             CORBA::Environment&) throw();

        virtual void withdraw(BankExceptions::CashAmount amount,
                             CORBA::Environment&)
        throw (BankExceptions::Account::InsufficientFunds);

        // IDL attributes
        virtual char* name(CORBA::Environment&) throw();

        virtual void
        name(const char*  _new_value, CORBA::Environment&)
        throw();

        virtual BankExceptions::CashAmount
        balance(CORBA::Environment&) throw();

        // C++ operations
2        BankExceptions_AccountImpl (
            const char* name,
            BankExceptions::CashAmount balance,
            const char* marker,
            CORBA::LoaderClass* loader) throw ();
        virtual ~BankExceptions_AccountImpl() throw();

3        static BankExceptions::Account_ptr LoadObject(
            const char* marker, CORBA::LoaderClass*)
            throw();

4        virtual void SaveObject(char* file_name) throw();

5        virtual void* _deref() { return this;}

    protected:
        CORBA::String_var m_name;
        BankExceptions::CashAmount m_balance;

        // The following are not implemented:
        BankExceptions_AccountImpl(const
                BankExceptions_AccountImpl&);
        BankExceptions_AccountImpl& operator=(const
                BankExceptions_AccountImpl&);
    };
```

This code is explained as follows:

1. `BankExceptions_AccountImpl` is the application implementation class, inheriting from the IDL-generated BOAImpl class.
2. The `BankExceptions_AccountImpl` constructor sets the marker as the account name and sets the loader for the account object.
3. The `LoadObject()` function is called from the `load()` function of the loader. This is passed the name of the file to load the account from.
4. The `SaveObject()` function writes the member variables of an account to a specified file.
5. The `_deref()` function casts from the `Account` interface class to the implementation class. A reference to the implementation class is required because you cannot call non-IDL operations (in this case, `SaveObject()`) from an interface class. Refer to "Casting from Interface to Implementation Class" on page 155 for more details.

LoadObject()

The `LoadObject()` function is called from the `load()` function of the loader. This loads an `Account` object from a specified file. `LoadObject()` can be coded as follows:

```
    // C++
    // In file bankexceptions_accountimpl.cxx

1     BankExceptions::Account_ptr
              BankExceptions_AccountImpl::LoadObject(
        const char* marker, CORBA::LoaderClass* loader
    ) throw() {
        char file_name[260];

2   char* envvar = getenv("IT_DEMO_ACCOUNTS_DIR");
        if (envvar == NULL) {
            envvar = "";
        }
        strcpy(file_name, envvar);
        strcat(file_name, marker);
        strcat(file_name, ".ser");

        ifstream account_file(file_name);
        if (account_file) {
            char loaded_account_name[100];
            float loaded_account_balance;
            account_file >> loaded_account_name
                         >> loaded_account_balance;

            // Now recreate the object
            BankExceptions::Account_var loaded_account
                = new BankExceptions_AccountImpl(
                                    loaded_account_name,
                                    loaded_account_balance,
                                        marker, loader);
            return
            BankExceptions::Account::_duplicate(loaded_account);
        }
```

```
        else {
            cerr << "Error loading file "<< file_name
                    << endl;
            return 0;
        }
    }
```

This code is described as follows:

1. `Loader::load()` must call `LoadObject()` because `load()` does not have access to the `Account` private data members.
2. Save the file to `<IT_DEMO_ACCOUNTS_DIR><marker>.ser`. By default during `make register`, the makefile sets the `IT_DEMO_ACCOUNTS_DIR` environment variable to the `bankpersistent` demonstration directory.

SaveObject()

The `SaveObject()` function is called from the `save()` function of the loader. This saves an `Account` object to a given file name when the server exits. `SaveObject()` can be coded as follows:

```
// C++
// In file bankexceptions_bankimpl.cxx

void BankExceptions_AccountImpl::SaveObject(
    char* file_name) throw()
{
    ofstream account_file(file_name);
    if (!account_file) {
        cerr << "Cannot open file " << file_name
                << "for writing"<< endl;
        cerr << "Object not saved" <<endl;
    }

    account_file << m_name << endl << m_balance
        << endl;
    if (!account_file) {
        cerr << "Cannot write to file " << file_name
        << endl;
        cerr << "Object not saved" <<endl;
    }
}
```

`Loader::save()` must call `SaveObject()` because `save()` does not have access to the `Account` private data members

Class BankImpl

Class `BankImpl` is implemented as follows:

```
// C++
#include "bankexceptions.hh"
#include "it_demo_nsw.h"

class BankExceptions_BankImpl : public virtual
BankExceptions::BankBOAImpl
{
public:
    // IDL operations
    virtual BankExceptions::Account_ptr
    create_account(const char*  name,
                          CORBA::Environment&)
    throw(BankExceptions::Bank::CannotCreate);
```

```
    virtual BankExceptions::Account_ptr
    find_account( const char*  name,
                      CORBA::Environment&)
    throw(BankExceptions::Bank::NoSuchAccount);

    // C++ operations
    BankExceptions_BankImpl(CORBA::LoaderClass* loader)
    throw();
    virtual ~BankExceptions_BankImpl() throw();

protected:
    // Size of m_accounts array storing accounts.
    static const int MAX_ACCOUNTS;
    // An array of Account_var
    BankExceptions::Account_var* m_accounts;
    CORBA::LoaderClass* account_loader;

    // The following are not implemented
    BankExceptions_BankImpl(
        const BankExceptions_BankImpl&);
    BankExceptions_BankImpl& operator=(const
        BankExceptions_BankImpl&);
};
```

Coding the Loader

A single loader object, of class `Loader`, is created in the server `main()` function, and each `Account` object created is assigned this loader. Each `BankExceptions_BankImpl` object holds a pointer to the loader object to associate with each `Account` object when it is created. Accounts are assigned an account name that acts as a marker for the object. The ability to choose markers is an important feature for persistence.

Bank objects are not associated with an application level loader. These are implicitly associated with the Orbix default loader. The server mainline creates a loader and a bank as follows:

```
// C++
// In file server.cxx.

// Loaders must be created dynamically.
Loader* accountloader = new Loader();
BankExceptions::Bank_var my_bank
    = new BankExceptions_BankImpl(accountloader);
```

Class Loader

Class `Loader` is the loader class for `Account` objects. This inherits from `CORBA::LoaderClass`. You can implement class `Loader` as follows:

```
// C++
// In file bankpersistent_loader.h

class Loader : public CORBA::LoaderClass {

public:
        Loader();
        virtual ~Loader();

    // Load object with given interface and marker.
        virtual CORBA::Object_ptr load (
                const char* object_interface,
                const char* marker,
                CORBA::Boolean isBind,
                CORBA::Environment&);
        // Save object.
        virtual void save (
                CORBA::Object*,
                CORBA::saveReason reason,
                CORBA::Environment&);
};
```

Class `Loader` redefines the `load()` and `save()` functions and inherits `rename()` and `record()` from `CORBA::LoaderClass`.

The `Loader` member functions can be implemented as follows:

```
// C++
// In file bankpersistent_loader.cxx

// The Loader constructor registers the loader
// object as a loader.
1   Loader::Loader ()
       : CORBA::LoaderClass(1)
    {}

    // Loader destructor
    Loader::~Loader() {
    }

2  // Loader::load()
    CORBA::Object_ptr Loader::load (
        const char* object_interface,
        const char* marker,
        CORBA::Boolean, CORBA::Environment&)
    {
        cout << "Loading object ... " <<endl
               << "interface: " << object_interface
               << endl << "marker: " << marker <<endl ;

        return BankExceptions_AccountImpl::LoadObject(
            marker, this);
    }
```

```
3               // Loader::save()
                void Loader::save (
                    CORBA::Object_ptr obj,
                    CORBA::saveReason reason,
                    CORBA::Environment&)
                {
                    if (reason == CORBA::explicitCall) {
                        char* file_name = new char[260];

                        cout<< "Saving object ... " <<endl
                               << "marker/filename: "
                               << obj->_marker () <<endl ;

                        BankExceptions::Account_var accountvar =
                               BankExceptions::Account::_narrow(obj);

4                       BankExceptions_AccountImpl* account_to_save =
                             (BankExceptions_AccountImpl*)
                              accountvar->_deref();

5                       strcpy(file_name, getenv("IT_DEMO_ACCOUNTS_DIR"));
                        strcat(file_name, obj->_marker ());
                        strcat(file_name, ".ser");
                        account_to_save->SaveObject (file_name);

6                       IT_Demo_NSW ns_wrapper;

                        // Set up object_name of the form
                        // BankPersistent.<accountname>
                        char object_name [100];
                        strcpy(object_name , "IT_Demo.BankPersistent.");
                        strcat(object_name, obj->_marker ());

                        ns_wrapper.setBehaviourOption(
                            IT_Demo_NSW::createMissingContexts);
                        ns_wrapper.setBehaviourOption(
                            IT_Demo_NSW::overwriteExistingObject);

                        try{
                            ns_wrapper.registerObject(
                                    object_name, accountvar);
                        }
                        catch (const CORBA::Exception& e) {
                        cout << "Unexpected exception" << e << endl;
                            throw;
                        }

                        CORBA::release(obj);
                        delete [] file_name;
                        return;
                    }
                }
```

This code is explained as follows:

1. The `CORBA::LoaderClass()` constructor takes a parameter indicating whether the loader being created should be included in the list of loaders tried when an object fault occurs. By default, this value is false. The `Loader()` constructor passes a value of `1` to the constructor. This indicates that instances of `Loader` should be added to this list.

2. Orbix calls the `Loader::load()` function when an object fault occurs on an account object associated with this loader. This in turn calls `BankExceptions_AccountImpl::LoadObject()`.

   The `AccountImpl::LoadObject()` function assigns the correct marker to the newly-created object. If it fails to do this, subsequent calls on the same object result in further object faults and calls to the `Loader::load()` function.

   You could use the `Loader::load()` function to read the data itself, rather than calling the static function `AccountImpl::LoadObject()`. However, to construct the object, `load()` would be dependent on there being a constructor on class `AccountImpl` that takes all of an account's state as parameters. Because this is not the case for all classes, it is safer to introduce a function such as `LoadObject()`.

   Equally, `Loader::save()` could access the account's data and write it out, rather than calling `AccountImpl::SaveObject()`. However, it would then be dependent on `AccountImpl` providing access to all of its state.

   In addition, defining `LoadObject()` and `SaveObject()` within class `AccountImpl` provides a useful split of functionality between the application level class, `AccountImpl`, and the loader class.
3. Orbix calls the `save()` function after the `_save()` is called on the `Account` object by the `bankexceptions_bankimpl` destructor. The `Account` object is passed in the first parameter. This example only handles the explicit call `saveReason`.
4. You must convert from `CORBA::Object_ptr` to an implementation object because the `SaveObject()` function is not an IDL operation.
5. Save the file to `<IT_DEMO_ACCOUNTS_DIR> <marker>.ser`. The environment variable `IT_DEMO_ACCOUNTS_DIR` is set by the makefile when `make register` is executed.
6. Bind the object in the Naming Service. The `Account` object bound is not used by the client. Instead, it is used by future server executions to check if an account exists in persistent storage.

Loaders are Transparent to Clients

When using loaders, clients can make invocations in the normal way. For example, a client that wishes to create a specific account can execute the following:

```
// C++
// In file bankmenu.cxx

// Call create_account() and run an account menu
void BankMenu::do_create()
    throw(CORBA::SystemException)
{
    cout << "Enter account name: " << flush;
    CORBA::String_var name =
                IT_Demo_Menu::get_string();

    try
    {
```

```
            BankExceptions::Account_var account =
                            m_bank->create_account(name);
            // Start a sub-menu with the returned account
            // reference
            AccountMenu sub_menu(account);
            sub_menu.start();
        }
        catch (const BankExceptions::Bank::CannotCreate&
                                    cant_create) {
            cout << "Cannot create an account, reason: "
                   << cant_create.reason << endl;
        }
    }
```

The `load()` function of the loader object is called if the target account is not already present in the server. If the loader recognizes the object, it handles the object fault by recreating the object from the saved data. If the load request cannot be handled by that loader, the default loader is tried next and this always indicates that it cannot load the object. This finally results in a `CORBA::INV_OBJREF` exception being returned to the caller.

Using Opaque Types in IDL

Orbix provides an extension to IDL that allows you to define opaque data types. You can pass opaque data types by value through an IDL definition without any interference from Orbix. This chapter describes how to use these Orbix-specific data types.

In accordance with the CORBA standard, Orbix objects are passed to and from IDL operations *by reference*. All such objects are described by an interface defined in IDL. Objects supporting an IDL interface are created in a server and object references rather than actual copies of the objects are passed to clients.

This model is appropriate to the majority of applications that use an ORB. However, in some circumstances, you may wish to pass objects across a CORBA IDL interface *by value* rather than by reference. Passing an object by value means that the internal state of the object is included in an operation parameter or return value and a copy of the object is constructed in the receiving process.

In addition, there has been demand for a mechanism that allows existing C++ objects to be passed across an IDL interface without the necessity to retrospectively define IDL interfaces for these objects. Such a mechanism allows the integration of IDL types with non-IDL data types within a CORBA environment.

The *opaque types* mechanism described in this chapter addresses both of these issues. A data type may be identified in IDL as *opaque* by the introduction of a new keyword, `opaque`. This means that nothing (except that it is a valid IDL type) is known at the IDL level. A type defined to be opaque behaves like an interface type. It can therefore be passed as a parameter or return value to an IDL operation, or used as an attribute type or as a member of a struct or exception. An opaque type is always passed to and from IDL operations by value, and you must supply a C++ class which implements the type. You must also provide *marshalling* functions that define how the object's state is packaged for transmission across the network and *unmarshalling* functions that define how the object's state can be extracted by the receiving process.

Note: Because of the Orbix-specific nature of opaque data types, you can only use opaque data types with Orbix.

Possible Alternative Solutions

Software's approach to passing objects between client and server processes by value is to introduce a new type constructor at the IDL level.

You can achieve similar results without extending the IDL language. One solution to transmitting an object by value is to define its state in an IDL `struct` definition. This solution is unsatisfactory for two reasons: first, you must separate state information from interface information; second, in the IDL definitions, you should make explicit information that properly belongs to the implementation.

A second solution is to pass an object's state information in binary form as a `sequence<octet>`. This mechanism does not make explicit the type of the information transmitted, so it does not violate the object's privacy. However, no marshalling or unmarshalling is performed on a `sequence<octet>`, so byte-swapping and other data-conversion becomes your responsibility. Furthermore, in stripping the interface of type information, the ORB assumes the role of an RPC package.

Using Opaque Types

This section demonstrates how to use the opaque mechanism to pass a user-defined type by value in IDL operations.

IDL Definitions

The example used defines an IDL interface `Calendar` that makes use of the opaque type `Date`. The IDL definitions are as follows:

```
// IDL
// In, for example, file calendar.idl.
opaque Date;

interface Calendar {
    // Today's date:
    readonly attribute Date today;

    // How long from the given date until today ?
    unsigned long daysSince(in Date d);
};
```

The opaque data type is introduced by the keyword `opaque`. An opaque type can be defined at file-level scope or within a module, at the same level as an interface definition. Like a `typedef` definition, opaque introduces a new IDL type. In the example, the new `Date` type is used as an attribute type and as an `in` parameter.

You can define the IDL definitions as follows:

```
idl -K calendar.idl
```

`opaque` is not a keyword in CORBA IDL, so the `-K` IDL compiler switch is used to indicate that support for opaque types is required.

Mapping of Opaque Types to C++

An opaque type declaration maps to an `include` directive in the C++ header file generated by the IDL compiler. For example, the declaration:

```
// IDL
// In calender.idl.
opaque Date;
```

maps to:

```
// C++
// In calender.hh.
#include <calenderO.h>;
```

In addition, the IDL compiler generates three operator prototypes for the opaque data type as follows:

```
// C++
// In calender.hh.
CORBA::Request& operator<<(
                    CORBA::Request&, const Date*);
CORBA::Request& operator<<(
                    CORBA::Request&, Date*&);
CORBA::Request& operator>>(
                    CORBA::Request&, Date*&);
```

To use the opaque type `Date`, you must define a C++ class `Date` in file `calenderO.h` and implement these operators. The operator implementations specify how to marshal and unmarshal the opaque type. These specify how to stream the opaque object's state into and out of a `CORBA::Request` object so that an object defined to be opaque can be transmitted over a network. Thus, the mapping to C++ for the IDL definitions described in "IDL Definitions" on page 284 is as follows:

```
// C++
// In calender.hh.
#include <calenderO.h>;

// This operator is now deprecated refer to page 288.
CORBA::Request& operator<<(
                CORBA::Request&, const Date*);
CORBA::Request& operator<<(
                CORBA::Request&, Date*&);
CORBA::Request& operator>>(
                CORBA::Request&, Date*&);
...
class Calendar: public virtual CORBA::Object {
public:
    ...
    // Details omitted.
    virtual Date* today(
        CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
    virtual CORBA::ULong daysSince(
        const Date* d,
        CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
};
```

Mapping for Operation Parameters

The mapping for opaque types used as operation parameters and return values is shown in the following table:

IDL	in	out	inout	return
T	T*	T*&	T*&	T*

Memory Management Rules

The memory management rules for opaque types follow a strict pattern of their own to allow as flexible use of opaques as possible. These rules are outlined as follows:

in parameters

Client side	You need to allocate storage and provide an appropriate value. You should not pass an uninitialized pointer. You must free the storage when it is no longer required, using the C++ `delete` operator.
Server side	Orbix makes the parameter available for the duration of the operation call. You must copy the parameter if it is to be retained beyond the lifetime of the operation call.

inout parameters

Client side	You must allocate storage and provide an appropriate value. You should not pass an uninitialized pointer. You must free the storage when it is no longer required, using the C++ `delete` operator.
Server side	Orbix makes the parameter available for the duration of the operation call. You can change the value passed in. If the value passed in is changed, the old value must be freed. The value is *not* deallocated automatically by Orbix when the operation completes.

out parameters and return values

Client side	In line with the rules for CORBA types, you cannot modify the value passed back in an `out` parameter. A copy of the value passed back can, of course, be modified. You must free the storage associated with an `out` parameter, when it is no longer required, using the C++ `delete` operator.
Server side	You must allocate storage and perform initialization. The value is *not* deallocated automatically by Orbix when the operation completes. You should not return an uninitialized pointer.

Implementing an Opaque Type

You must provide an implementation class for the opaque type. This class must be defined in the file included in the generated `.hh` file.

A simple class definition for the `Date` class is as follows:

```
// C++
// In file calenderO.h.
#include <iostream.h>
...

class Date {
    friend CORBA::Request& operator<<(
                    CORBA::Request&, const Date*);
    friend CORBA::Request& operator<<(
                    CORBA::Request&, Date*&);
    friend CORBA::Request& operator>>(
                    CORBA::Request&, Date*&);
protected:
    short day, month, year;
public:
    Date();
    Date(short d, short m, short y);
    void print();
};
```

This class could be implemented as follows:

```
// C++
// In, for example, date.cc.
#include <iostream.h>
#include "calendar.hh"
#include "date.h"
...

Date::Date() {
    // Construct an object containing today's date
    // (code not shown).
}

Date::Date(short d, short m, short yr) {
    day = d;
    month = m;
    year = yr;
}

void Date::print() {
    cout << day << "/" << month << "/" << year;
}
```

To complete the implementation, you must implement the marhsalling operators as follows:

- The insertion operator, `operator<<()`, marshals the opaque object's state into a `CORBA::Request` for transmission to a remote process.

  (Class `CORBA::Request` is used to package an operation request and to return out and inout parameters and results. For more details, see the chapter "Dynamic Invocation Interface" in this

guide, and the entry for CORBA::Request in the ***Orbix Programmer's Reference C++ Edition***.)

On the client side, the `const` version of this operator is used to marshal `in` parameters and the non-`const` version is used to marshal `inout` parameters. On the server side, the (non-`const` version of the) operator is used to marshal `inout` and `out` parameters and operation results.

- The extraction operator, `operator>>()`, unmarshals an opaque object that is received from a remote process in a `CORBA::Request`.

  On the client side, this operator is used to unmarshal `inout` and `out` parameters and results. On the server side, it is used to unmarshal `in` and `inout` parameters.

Note: Recent revisions in the operator prototypes for opaques have deprecated this operator. These revisions add more flexibility in terms of memory management. However, this operator is still used here for backwards compatibility.

The following is an implementation of these operators for the `Date` class:

```
// C++
// Marshalling operator for (client side)
// in parameters.
CORBA::Request& operator<<(
                    CORBA::Request& r, const Date* d) {
    if (d) {
        r << d->day;
        r << d->month;
        r << d->year;
    } else {
        r << 0;
        r << 0;
        r << 0;
    }
    return r;
}

// Marshalling operator for (client and
// server side) inout parameters and (server
// side) out parameters.
CORBA::Request& operator<<(
                    CORBA::Request& r, Date*& d) {
    if (d) {
        r << d->day;
        r << d->month;
        r << d->year;
    } else {
        r << 0;
        r << 0;
        r << 0;
    }
    // To avoid memory leak of inout and out
    // parameters:
    delete d;

    return r;
}
```

```
    // Unmarshalling operator for (client side) inout
    // and out parameters and results and for (server
    // side) in and inout parameters.
    CORBA::Request& operator>>(
                        CORBA::Request& r, Date*& d) {
        d = new Date;
        r >> d->day;
        r >> d->month;
        r >> d->year;
        return r;
}
```

The order in which class `Date`'s members are inserted into the `CORBA::Request` is irrelevant. However, the unmarshalling operator must extract the members in the *same* order as the order in which they are inserted.

Because a nil (zero) pointer might be passed in a parameter expecting an opaque type, the insertion operators should ensure that appropriate zero values for each member are inserted into the `CORBA::Request`. If required to handle marshalling errors, the insertion and extraction operators for an opaque type may raise a `CORBA::MARSHAL` system exception.

Note: The non-`const` version of the marshalling operator, `operator<<()`, should free the memory allocated to the opaque object (allocated in `operator>>()`) in order to avoid a memory leak for `inout` and `out` parameters on the server side.

Implementing an Interface that uses an Opaque Type

The implementation of the `Calendar` interface is straightforward; the code is shown below.

```
// C++
#include "calendar.hh"
#include "date.h"

class CalendarImpl : public CalendarBOAImpl {
protected:
    Date* day;
public:
    CalendarImpl();
    virtual Date* today(
        CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
    virtual CORBA::ULong daysSince(
        const Date* d,
        CORBA::Environment& IT_env =
            CORBA::IT_chooseDefaultEnv());
};

CalendarImpl::CalendarImpl() {
    day = 0;
}

Date* CalendarImpl::today(
        CORBA::Environment&) {
    Date* d = new Date;
    return d;
}

CORBA::ULong CalendarImpl::daysSince(
        const Date* d, CORBA::Environment&){
    // Calculate number of days between this
    // date and date in d (code not shown).
}
```

Transforming Requests

This chapter describes how you can use transformers to modify data buffers containing Orbix operation call information, immediately before and after transmission across the network. Transformers are an Orbix-specific feature.

In Orbix, an operation invocation or an operation reply is transmitted between a client and a server in a `CORBA::Request` object. Using the Dynamic Invocation Interface, a `CORBA::Request` is explicitly created while a static invocation results in the implicit creation of a `CORBA::Request` object.

This chapter describes how you can modify a `CORBA::Request`'s data buffer, allowing a client or server process to specify what modifications to the buffer should occur when requests or replies are transmitted to other processes. The ability to modify this data directly preceding its transmission, or directly subsequent to its reception means that you can add additional information to the data stream; for example, information identifying the participants in the communication. The data stream may be encrypted for security purposes and so on. The process of modifying a `CORBA::Request`'s data buffer is known as *transforming* the data buffer.

The functionality provided by `Request` transformation is at a lower level than that provided by filters, as described in the chapter "Filtering Operation Calls" on page 233. Transforming requests allows access to the actual data buffer transmitted in a `Request`.

Note: Because of the Orbix-specific nature of transformers, you can only use transformers with Orbix.

Transforming Request Data

Transformation of a `CORBA::Request`'s data buffer is performed by a *transformer object*. To obtain a new transformer object do the following:

1. Define a class that inherits from the class `CORBA::IT_reqTransformer`.
2. Create an instance of this class.
3. Register this instance with Orbix.

   You can register the transformer object so that it performs transformations on all communications to and from the process that contains the transformer object. Alternatively, you can register the transformer so that transformations are performed only on communications to and from a particular server on a particular host that contains the transformer.

Because transformations are applied when an operation invocation leaves or arrives at an address space, no transformations are applied when the caller and invoked object are collocated.

The IT_reqTransformer Class

The CORBA::IT_reqTransformer class defines the interface to transformer objects. This class is defined as follows:

```
// C++
// In class CORBA.
class IT_reqTransformer {
protected:
    const char* m_remote_host;
public:
    virtual CORBA::Boolean transform(
        CORBA::Octet*& data,
        CORBA::ULong& actual_sz,
        CORBA::ULong& allocd_sz,
        CORBA::Boolean send,
        CORBA::Boolean is_first);

    virtual void free_buf(
        unsigned char* data,
        CORBA::ULong actual_sz,
        CORBA::ULong allocd_sz);
    virtual const char* transform_error();

    void setRemoteHost(const char* c);
};
```

A class derived from IT_reqTransformer can access a CORBA::Request's data and can therefore manipulate or transform the data as required. The derived class must, at least, override the transform() function. Refer to the ***Orbix Programmer's Reference C++ Edition*** for full details on the IT_reqTransformer class.

The transform() function is called by Orbix immediately before transmitting the data in a Request from an address space and immediately subsequent to receiving a Request from another address space. The derived class can allocate new storage to handle any alteration in the data size caused by the transformation. If the derived class alters the method by which the data is stored in buffers, you may also need to override the default free_buf() operation to handle the release of this data. Before calling the transform() function, Orbix records the name of the host that initiates a request in the member m_remote_host.

The transform() function may indicate that a TRANSFORM_ERR system exception should be raised by Orbix by returning 0 (false) from transform().

A derived class may implement the transform_error() function to return a string containing suitable error text. The string returned by this function forms part of the error string output by the operator:

```
// C++
friend ostream& operator<<(
    ostream&, CORBA::SystemException*);
```

when the TRANSFORM_ERR exception is caught. You must free the string returned by transform_error(), using CORBA::string_free().

Registering a Transformer

Orbix provides two functions to register a transformer object (an instance of `CORBA::IT_reqTransformer`). You can call both on the `CORBA::Orbix` object.

1. The function:

   ```
   // C++
   // In class CORBA::ORB.
   CORBA::IT_reqTransformer* setMyReqTransformer(
       CORBA::IT_reqTransformer* transformer,
       CORBA::Environment& IT_env =
           CORBA::IT_chooseDefaultEnv())
   ```

   registers a transformer object as the default transformer for all `Request`s entering and leaving an address space.

2. The function:

   ```
   // C++
   // In class CORBA::ORB.
   void setReqTransformer(
       CORBA::IT_reqTransformer* transformer,
       const char* server,
       const char* host = 0,
       CORBA::Environment& IT_env =
           CORBA::IT_chooseDefaultEnv())
   ```

   registers a transformer object for all `Request`s destined for a specific server and host and for all `Request`s received from a specific server and host. This function can be called more than once to register different server/host pairs.

A transformer registered using `setReqTransformer()` overrides any default transformer registered with `setMyReqTransformer()`.

Note: At most one transformation is applied to any `Request`—the default transformation registered with `setMyReqTransformer()` or overriding specific transformation registered with `setReqTransformer()`.

An Example Transformer

This section presents a simple example of a transformer that adds the name of the sending host to a `Request`'s buffer when sending a `Request` out of a process. It also removes the host name from a `Request`'s buffer when the process receives a `Request` object that contains a operation reply.

The transformer is implemented by class `Transformer` defined as follows:

```
// C++
#include <CORBA.h>
#include <iostream.h>

#define ERR_STR "Transformer: bad transformation"

class Transformer :
                public CORBA::IT_reqTransformer {
public:
    virtual CORBA::Boolean transform(
        CORBA::Octet*& data,
        CORBA::ULong& actual_sz,
        CORBA::ULong& allocd_sz,
        CORBA::Boolean send,
        CORBA::Boolean first);

    virtual const char* transform_error();
};

extern Transformer* transformer;
```

The `Transform` class overrides the functions `transform()` and `transform_error()` only. These are implemented as follows.

```
// C++
...
unsigned char Transformer::transform(
        CORBA::Octet*& data,
        CORBA::ULong& actual_sz,
        CORBA::ULong& allocd_sz,
        CORBA::Boolean send,
        CORBA::Boolean first) {

    if (!first)
        return 1;

    unsigned long i;
    // m_remote_host is set by Orbix prior
    // to invoking transform().
    if (send) { // Sending.
        unsigned long shift = strlen(m_remote_host);
        unsigned char*  old_data = data;

        if ((shift + actual_sz) > allocd_sz) {
            data = new unsigned char [shift + actual_sz];
        }

        for (i = shift + actual_sz -1; (i >= shift); i--)
            data [i] = old_data [i - shift];
```

```
            for (i = 0; i < shift; i++)
                data [i] = m_remote_host [i];

            actual_sz += shift;

            if (data != old_data)
                delete[] old_data;
        }
        else {
            unsigned long shift =
                            strlen(CORBA::Orbix.myHost());
            char* this_host = new char[shift];
            this_host = strdup(CORBA::Orbix.myHost());
            for (i = 0; i < shift; i++)
                if (data [i] != this_host [i])
                    return 0;

            for (i = 0; i < actual_sz - shift ; i++)
                data [i] = data [i + shift];

            actual_sz -= shift;
            delete this_host;
        }

        cout << "---USER Transform returning"
                << " actual_sz: " << actual_sz
                << " allocd_sz: " << allocd_sz
                << " send: " << (int)send << endl;
        return 1;

        const char* Transformer::transform_error() {
            return ERR_STR;
    }

    // Create a Transformer:
    Transformer* transformer = new Transformer;
```

The `first` parameter to the function `transform()` indicates whether the buffer in `data` is the first in a sequence of buffers. In Orbix, a `Request` object being sent from an address space can contain more than one data buffer while a `Request` object received into an address space always contains just one buffer. In this example, the first buffer is the only one modified by `transform()`. The `send` parameter indicates whether the `Request` is incoming or outgoing. The `transform()` function uses the `send` parameter to determine whether to add or remove the host name to the `Request`'s buffer.

Registering the Transformer

Calling the following on the ORB registers this transformer as the default transformer for a client or server process:

```
setMyReqTransformer(transformer);
```

To register a transformer that acts on `Request`s going to or received from a specific server on a specific host, make the following call on the ORB:

```
setReqTransformer(
    transformer, "myServer", "alpha");
```

Using Threads with Orbix

This chapter presents details and explains the benefits of multi-threaded clients and servers, and the mechanisms available for multi-threaded programming.

Normally, Orbix client and server programs contain one thread that starts executing at the beginning of the program and continues until the program terminates. Many modern operating systems allow a process to create lightweight threads, with each thread having its own set of CPU registers and its own stack. Each thread is independently scheduled by the operating system, so it can run in parallel with the other threads in its process. The mechanisms for creating and controlling threads differ between operating systems, but the underlying concepts are common. The POSIX standard is supported by most UNIX and OpenVMS systems.

The programming steps required to create threads in Orbix are straightforward. In addition, you can program many different models of thread support.

The example code in this chapter uses POSIX-compliant threads to illustrate these concepts. The `Orbix3.0\demo` directory of your Orbix installation provides analogous examples for the threads package available on your operating system.

Benefits of Multi-threaded Clients and Servers

Both clients and servers can benefit from multi-threading. However, the advantages of multi-threading are more apparent for servers than for clients.

Multi-threaded Servers

For some servers, it is satisfactory to accept one request at a time and to process each request to completion before accepting the next. Where parallelism is not required by an application, there is little point in making such a server multi-threaded. However, some servers would offer a better service to their clients if they processed a number of requests in parallel. Parallelism of such requests may be possible because a set of clients can concurrently use different objects in the same server, or because some of the objects in the server can be used concurrently by a number of clients.

Some operations can take a significant amount of time to execute, because they are compute-bound, because they perform a large number of I/O operations, or because they make invocations on remote objects. If a server can execute only one such operation at a time, clients suffer because of long latencies before their requests can be started. The main benefits of multi-threading are that the latency of requests can be reduced, and the number of

requests that a server can handle over a given period of time can be higher. Multi-threading also enables you to take advantage of multi-processor machines.

The simplest threading model is where a thread is created *automatically* for each incoming operation/attribute request. Each thread executes the code for the operation/attribute being called, executes the low level code that sends the reply to the caller, and then terminates. Any number of such threads can run concurrently in a server, and they can use normal concurrency control techniques to prevent corruption of the server's data. You must program this protection at two levels: the underlying ORB library must be thread-safe so that concurrent threads do not corrupt internal variables and tables; and the application level must be made thread- safe by the programmer.

Threads are not without their costs, however. Firstly, it may be more efficient to avoid creating a thread to execute a very simple operation. The overhead of creating a thread may be greater than the potential benefit of parallelism. Secondly, you must ensure that the application code is thread-safe.

Specifically, Solaris, Windows NT and POSIX threads are pre-emptive. This means that they can be interrupted at any time and delayed while other threads execute. Nevertheless, the benefits frequently outweigh the costs and multi-threaded servers are considered essential for many applications.

A benefit of using Orbix is that the actual creation of threads in a server is very simple, and therefore adds little or no cost for the programmer.

You can also explicitly create threads in servers, using the threading facilities of the underlying operating system. You can do this so that a remote call can be made without blocking the server. You can also do this within the code that implements an operation or attribute, so that some complex algorithm can be parallelized and performed by a number of threads. These threads can be in addition to those created implicitly to handle each request.

Multi-threaded Clients

Multi-threaded clients can also be useful. A client can create a thread that makes a remote operation call, rather than making that remote call directly. The result is that the thread that makes the call blocks until the operation call has completed, but the rest of the client can continue in parallel. "Comparison with Non-Blocking Calls" on page 299 compares this approach with the use of non-blocking calls made by single-threaded clients. Another advantage of a multi-threaded client is that it can receive incoming operation requests to its objects without having to poll for communication events; for example, it can receive callbacks from a server.

Clients must create threads explicitly, using the threading facilities of the underlying operating system; this is not difficult to perform. Naturally, you must code multi-threaded clients to ensure they are thread-safe, using some concurrency control mechanism. For

servers, the difficulty of doing this depends on the complexity of the data, the complexity of the concurrency control rules, and the form of concurrency control mechanism being used.

Comparison with Non-Blocking Calls

You can gain some of the benefits of using multiple threads by making operation calls that do not block the caller. IDL `oneway` calls do not block their caller, and you can make normal calls without blocking by using the DII and the `send_deferred()` function on a `Request`. Non-blocking calls can be made within a client or a server.

However, there is little to recommend in using non-blocking calls. Using threads is easier and more powerful than using non-blocking calls:

- *Easier*

  Threads provide an easier means of gaining concurrency. Consider a client that wishes to carry out a number of actions, each requiring a number of two-way operation requests. One way to do this is to make the first two-way operation call associated with each action without blocking, and to process the results in whatever order they arrive. In this way, at any time, there is one outstanding (non-blocking) operation call for each action. Once a reply arrives for the current operation call for an action, the next call for that action can be made. The difficulty here is that the client must loop to accept each reply, and it must maintain a table to indicate the next request to make for each action. This is complex and error-prone.

  In contrast, the equivalent coding using threads is very simple. A thread can be created for each action, and that action can make normal blocking calls for each request that is to be made in turn.

- *More powerful*

  The real benefit of multi-threaded servers is the ability to handle calls from a number of clients concurrently. This cannot be gained using non-blocking calls.

  Consider an attempt to do so. A single-threaded server can accept an incoming operation request, and during the processing of this request it can use a non-blocking call to make a request on a remote object. Naturally, the server does not block while the remote object is processing the call, but it cannot accept another incoming operation request from the same or another client.

  The only way that it can accept another operation call is to complete the first call, the call on whose behalf it has made the non-blocking remote call (do this by exiting the C++ member function that implements the operation). The server cannot accept another call until it has completed the current one.

Nevertheless, non-blocking calls can sometimes be useful. Firstly, some operating systems do not support threads; and secondly, although threads may be available, it might not be possible to use

them because an application is using a library that is not thread-safe. Finally, for very simple uses in clients, the complexity of using non-blocking calls is no greater than that of using threads. Nevertheless, the real benefits of multi-threaded servers is the ability to handle calls from different clients concurrently. This cannot be gained using non-blocking calls.

Thread Programming in Orbix

Orbix provides a thread-safe version of the Orbix libraries for use with the underlying operating system's threads package. At appropriate points within the Orbix libraries, locking code has been added to ensure that the Orbix internal data structures are correctly managed in a pre-emptive threading environment. The Orbix libraries are thread-safe.

In addition to the locking code, the client and server library both create and use threads internally. These threads are not exposed to application programmers, and execute code within the library only.

Note: All existing application code written for the non-threaded Orbix libraries continues to execute correctly if linked with the threaded Orbix libraries. In addition, an Orbix programmer can choose to ignore threads.

Although the threaded Orbix libraries create some threads internally, by default there is only one thread to handle incoming requests: for example, a server only handles one call at a time. To create a thread per incoming request, you must install a filter that creates these threads. Refer to the chapter "Filtering Operation Calls" for details. This code is supplied, and can be used without modification. It should be viewed as code that extends the ORB, rather than as application level code.

You can use application level threads within a client application, or within a server application, or within both. A non-threaded client can interact with a threaded server, and vice-versa. Naturally, applications written using the standard (non thread-safe) Orbix product can also interact with threaded applications. Server applications can choose when to create threads, including in response to incoming operation requests.

Compiling Orbix Applications

This section describes the compilation switches required when building Orbix applications on Windows and UNIX platforms.

Windows Platforms

The Orbix libraries are built with the `/MD` switch, which links them with the MSVCRT multi-threaded runtime libraries. You should also build your applications using the `/MD` switch.

UNIX Platforms

To build an application using the thread-safe version of the Orbix libraries, it is important to compile with `-D_REENTRANT`. In fact, this is true for most threaded applications. It ensures that the C++ compiler generates re-entrant code correctly, and also selects the correct header file options:

```
% CC -D_REENTRANT foo.cc
```

Your link line should link with the `mt` form of the Orbix libraries, and with the appropriate library for the threads package used. The details vary depending on the particular platform, so you should consult a `Makefile` provided in the `Orbix3.0/demo` directory for exact details.

UNIX examples are as follows:

Solaris

```
% CC -D_REENTRANT -o foo foo.cc \
           -lorbixmt -mt -lnsl -lsocket
```

POSIX-Compliant

```
% CC -D_REENTRANT -o foo foo.cc \
           -lorbixmt -threads -lrt
```

Operating System Support for Creating Threads

Before discussing the filter code that creates threads, this section shows the code that is required on some operating systems to create a thread.

Windows

```
// C++
#include <process.h>
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpsa,
    DWORD cbStack,
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpbThreadParm,
    DWORD fdwCreate,
    LPDWORD lpIDThread);
```

Solaris

```
// C++
#include <thread.h>
thr_create (void* stack_base, size_t stack_size,
    void*(start_routine)(void*), void* arg,
    long flags, thread_t* new_thread)
```

POSIX-Compliant

```
// C++
#include <pthread.h>
int pthread_create(pthread_t* tid,
    pthread_attr_t*,
    void*(start_routine)(void*), void* arg);
```

If a client or server creates a thread to make a remote request, it can wait for that thread to terminate using one of the following calls:

Windows	`WaitForSingleObject()`
Solaris	`thr_join()`
POSIX	`pthread_join()`

Creating a Thread to Handle a Request

As explained in the chapter "Filtering Operation Calls", a per-process filter's `inRequestPreMarshal()` function can create a thread to handle an incoming request. The `inRequestPreMarshal()` function should use an underlying threads package—for example, the Solaris threads package—to create a thread, and the thread should then handle the request, usually by instructing Orbix to send the invocation to the target object.

The `inRequestPreMarshal()` function should return `-1` to Orbix to indicate that it has created a thread that handles the call. Unlike the other filter points, `inRequestPreMarshal()` has a return type `int`. This allows it to return `1` to indicate that the request is accepted and should be processed as normal; return `0` to indicate that the request should be rejected; or return `-1` to indicate that the call is being handled by a separate thread.

The new filter class should inherit from `CORBA::ThreadFilter`, which in turn derives from `CORBA::Filter`. The code below is the example thread filter that creates a thread per request. The version shown uses the Solaris threading facility.

```
// C++
class CreatesThread : public CORBA::ThreadFilter {
public:
    // Only consider one monitor point here.
    virtual int inRequestPreMarshal
                    (CORBA::Request&,
                     CORBA::Environment&);
};

// Create the required single instance.
CreatesThread threadDispatcher;


// Define start function for new thread
static void* startThread(void* vp) {
    // Tell Orbix to resume processing a request.
    CORBA::Orbix.continueThreadDispatch
                            (*(CORBA::Request*)vp);
    return 0;
}

// Implementation of inRequestPreMarshal().
int CreatesThread::inRequestPreMarshal
                                (CORBA::Request& r,
                                    CORBA::Environment&) {
    // Create a thread using the threads-package:
    // The thread entry point is 'startThread'
    thread_t tid;
    thr_create(NULL, 0, startThread, (void*)&r,
                                        THR_DETATCHED,&
tid);
    // Indicate to Orbix that a thread was created.
    return -1;
}
```

`CreatesThread::inRequestPreMarshal()` is the first part of this code to execute. It uses the Solaris function `thr_create()` to create the new thread, specifying that the new thread is to execute the function `startThread()`. The value `-1` is returned to inform Orbix that a new thread has been created.

The role of `startThread()` is to instruct Orbix to continue to process the operation or attribute request within the new thread. It does this by calling the low-level Orbix function `continueThreadDispatch()`, passing it the `Request` variable that represents the request being made. The request is passed to `startThread()` as parameter `vp`, which although declared to be of type `void*`, is actually of type `CORBA::Request*`. The rules of the Solaris threading package dictate that the function that a thread is to execute (`startThread()` in this case) must take a `void*` parameter—passed as the fourth parameter to `thr_create()`.

Concurrency Control

Although Orbix contains sufficient locks to ensure the thread-safety of its internal variables and tables, and the low-level variables associated with each Orbix object, you must add appropriate synchronization code to the shared data structures and objects created in an application. Refer to the appropriate system programmer's manual to understand how to do this for a particular threads package.

Note: Orbix does not synchronize access to application level objects and application data structures.

Thus, for example, if a server programmer creates a thread filter as described in "Creating a Thread to Handle a Request" on page 302, it is possible that several application level threads may try to access the same application object in the server. In particular, if clients simultaneously request the server to invoke IDL operations on the *same* target object within the server, that object will be subject to concurrent access. You must thus take care that access to the state of the target application object is synchronized as appropriate, by using locking code built using the underlying threads package. For example:

```
// C++
class Foo {
    short m_counter; // Some state.
    mutex_t* m_lock; // Mutex lock for state.
public:
    Foo() {
        m_counter = 0;
        mutex_init(&m_lock, USYNC_THREAD, NULL);
    }
    void increment() {
        mutex_lock(&m_lock);
        m_counter++;
        mutex_unlock(&m_lock);
    }
};
```

Each `CORBA::Object` in Orbix includes an internal read/write lock used by Orbix to synchronize concurrent access to the Orbix-specific state of that object.

A read lock is acquired, for example, if a thread calls the `CORBA::Object::_refCount()` member function. Similarly, a write lock is acquired for the duration of the `_duplicate()` static member function on each IDL C++ class. However this read/write lock is not acquired when any application specific state of that object is accessed. For example, if an implementation class derives from a BOAImpl class that in turn derives indirectly from `CORBA::Object` adds member variables, or if a smart proxy does likewise, this additional state is not protected by the internal `CORBA::Object` read/write lock.

In principle, the internal `CORBA::Object` read/write lock could be made available to derived BOAImpl classes. In practice, however, there is a possibility that deadlock situations might occur because of interactions between the internal use of this lock in Orbix, and the use made by a programmer in a derived class. For this reason, access to the internal lock is discouraged.

Models of Thread Support

In addition to the *thread per request* model described in "Creating a Thread to Handle a Request" on page 302, a derived class of `ThreadFilter` can be used to program other models, such as the following:

Pool of Threads

In this model, a pool of threads is created to handle incoming requests. The size of the pool puts some limit on the server's use of resources, and in some cases that is better than the unbounded nature of the *thread per request* model. Each thread waits for an incoming request, and handles it before looping to repeat this sequence.This can provide the best balance between concurrency and resource usage.

Thread Per Client

In this model, a thread is created for each client process that is currently connected to a server. Each thread handles the requests from one client process, and ignores other requests. This may be useful if thread creation is too expensive to have a thread created for each request; but of course it does give the potential of having idle threads corresponding to clients that are currently not making requests to objects in the server. One particularly important use of this model is for DBMS integration, where in some cases it is important to run all of a client's requests in the same thread. This is normally because it is necessary to run consecutive requests from the same client in the same transaction.

Thread Per Object

In this model, a thread is created for each object (actually, for a subset of the objects in the server). Each of these threads accept requests for one object only, and ignores all others. This can be an important model in real-time processing, where the threads associated with some objects need to be given higher priorities that those associated with others.

Implementing Models of Thread Support

This section gives a brief outline of how you can implement these models.

Implementing Pool of Threads

To implement this model, you should create a pool of threads, and each thread should wait on a shared semaphore. When a request arrives, the `inRequestPreMarshal()` function of the `ThreadFilter` should place a pointer to the `Request` in an agreed variable, and signal the semaphore. Alternatively, a queue can be used. One of the threads will awaken, and should call `continueThreadDispatch()` before looping to repeat the sequence.

Implementing Thread Per Client

There are two variations on how this should be implemented, depending on whether or not a single client can make concurrent calls on objects in a server. If a client can only make one call at a time, then the `inRequestPreMarshal()` function should determine the identity of the caller, perhaps finding the file descriptor on the server that the call was made through. It should then use this to locate the corresponding thread. Specifically, a synchronization variable (a mutex or semaphore) is located; and this is signaled so that the thread will awaken. The `inRequestPreMarshal()` function should pass (a pointer to) the `Request` object to the thread, so that it can call `continueThreadDispatch()`.

If a client can make concurrent calls to the objects in the server, `inRequestPreMarshal()` should use a queue to communicate with the chosen thread. It should add the `Request` to the correct thread's queue, and signal a semaphore to mark the fact that there is one more entry in the queue. There should be one semaphore and one queue per thread, and each thread should wait on its own semaphore.

Implementing Thread Per Object

To implement this model, you should create a thread for each (or a subset of) the objects in the server. Each thread should have it's own semaphore and queue of requests, and it should wait on its own semaphore.

The `inRequestPreMarshal()` function should add the `Request` to the correct queue of requests, and signal the correct semaphore. When the thread awakens, it should call `continueThreadDispatch()` to process the topmost request, and then loop to await the next one.

Changing Internal Orbix Thread Creation

When you run an Orbix application, Orbix starts a number of internal threads in a thread pool. These threads work together to listen for incoming connection attempts from clients and read requests from the network. For a request event, ultimately an application thread is used to process the request using a thread model written by the user, such as the models described above.

The internal network threads use a leader-follower design. This means that one thread in the pool is doing the low-level `TCP/IP select()`, and when activity occurs, this thread processes it.

Simultaneously another thread is dispatched from the pool to perform the low-level `TCP/IP select()`. When a thread is done with whatever activity it needs to perform, it is returned to the pool.

The size of the internal network thread pool is controlled by the configuration parameter `IT_DEF_NUM_NW_THREADS`. The value defaults to 1. The user can change this default if a larger initial internal network thread pool is needed, and this can be an effective tuning parameter. Additionally, the user can use the method `CORBA::ORB::add_nw_threads()` to increase the number of threads in the pool at any time.

The function is defined as follows:

```
// C++
// In class CORBA::ORB
CORBA::Boolean add_nw_threads(CORBA::ULong num_threads)
```

The installed default number of network threads (1) is the best setting for most applications. The network threads are responsible for a small amount of work to read the network buffers and deposit the message on an event queue, which is processed by the application model. Some larger applications that have many clients may want to increase the number of network threads in the server processes.

Note:

As with any threading issue, the benefits of the additional threads depends greatly on the underlying hardware/OS architecture.

The network threads are also responsible for recomposing a full IIOP message from any IIOP fragments. However, IIOP fragment messages are rarely used in most CORBA systems (Orbix 3.x can receive fragments, but does not generate fragments). For tuning, the number of network threads should only be increased if the TCP/IP buffers are over filling. Having two or more network threads allows the TCP/IP buffers to be processed while messages are being processed in application threads.

Service Contexts in Orbix

This chapter introduces service contexts in Orbix applications. Service contexts are a CORBA-defined way of implicitly passing service- specific information in IIOP requests and replies. This chapter describes the Orbix APIs that enable you to supply and consume context information.

Service contexts provide a mechanism for passing service-specific information as *hidden parameters* in Internet Inter-ORB Protocol (IIOP) message headers. The CORBA interoperability specification defines service contexts as a sequence of octets with an associated identity number. For example:

```
module IIOP {
    typedef unsigned long ServiceId;

    struct ServiceContext {
1       ServiceId context_id;
2       sequence<octet> context_data;
    };
typedef sequence <ServiceContext> ServiceContextList;

        const ServiceId TransactionService = 0;
        const ServiceId CodeSets = 1;
    };
```

The code is explained as follows:

1. The `context_id` is the means by which a particular service context is recognized.
2. The `context_data` or octet sequence is the data part of the context.

According to the General Inter-ORB Protocol (GIOP) specification, service contexts are transmitted between clients and servers in GIOP `RequestHeader`s and `ReplyHeader`s.

The `RequestHeader_1_1` struct is defined in IDL as follows:

```
module GIOP {

    // GIOP 1.1
    struct RequestHeader_1_1 {
        IOP::ServiceContextList         service_context;
                unsigned long           request_id;
                 boolean                response_expected;
                    octet               reserved[3];
             sequence<octet>            object_key;
             string                     operation;
             Principal              requesting_principal;
};
```

The Orbix Service Context API

The CORBA-compliant API for service contexts in Orbix comprises the following external interfaces:

- The `ServiceContextHandler` class.
- The ORB interfaces.
- The `ServiceContextList`.

ServiceContextHandler Class

The `ServiceContextHandler` class is the base class used to define handlers for a particular `ServiceContext` ID you want to deal with. There is a handler registered on the client and the server for each `ServiceContext` you wish to handle. The handlers are recognized by their ID, which corresponds to the ID of the `ServiceContext` they are handling.

The `ServiceContextHandler` class is defined as follows:

```
class ServiceContextHandler {
   public:
    CORBA(Ulong) m_serviceContextId;

    ServiceContextHandler
      (CORBA::Ulong SrvCntxtId, CORBA::Environment& env));
    ~ServiceContextHandler();

    CORBA::Boolean incomingRequest
      (CORBA::Request& req, CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
    CORBA::Boolean outboundRequest
      (CORBA::Request& req, CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
    CORBA::Boolean incomingReply
      (CORBA::Request& req, CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
    CORBA::Boolean outboundReplyHandler
      (CORBA::Request& req, CORBA::Environment& IT_env =
        CORBA::IT_chooseDefaultEnv());
   };
```

ORB Interfaces

ORB APIs are provided to allow services to supply and consume context information at appropriate points in the process of sending and receiving requests and replies.

Examples of ORB APIs are as follows:

```
CORBA::ORB::registerPerRequestServiceContext
    (ServiceContextHandler* CtxHandler, CORBA::Environment&
                IT_env = CORBA::IT_chooseDefaultEnv());
CORBA::ORB::unregisterPerRequestServiceContext
    (CORBA::Ulong& CtxHandlerId, CORBA::Environment&
                IT_env = CORBA::IT_chooseDefaultEnv());
CORBA::ORB::registerPerObjectServiceContext
    (ServiceContextHandler* CtxHandler,
             CORBA::Object* handledObject, CORBA::Environment&
                IT_env = CORBA::IT_chooseDefaultEnv());
CORBA::ORB::unregisterPerObjectServiceContext
    (CORBA::Ulong& CtxHandlerId,
             CORBA::Object* HandledObject, CORBA::Environment&
                IT_env = CORBA::IT_chooseDefaultEnv());
```

ServiceContextList

The `ServiceContextList` is a field in an IIOP message header containing all the service context data associated with a particular request or reply. The `ServiceContextList` is implemented as a sequence of `ServiceContext`s.

```
class ServiceContextList {
    public:
    // Includes all of the normal sequence operators.
    ...
    friend PerObjectServiceContextHandler;
    friend PerRequestServiceContextHandler;
};
```

Using Service Contexts in Orbix Applications

The API for service contexts in Orbix is based on two usage models:

- `ServiceContext` per request.

  This is where service contexts are handled on all requests and replies entering and leaving a process.

- `ServiceContext` per object.

  This is where only service context information is handled for requests and replies going to or coming from a particular object.

The mechanism whereby a particular service context per request can be implemented is discussed in detail here. An overview of the implementation of a particular service context per object is also given.

ServiceContext Per-Request Model

Consider the following overview of implementing service contexts per request in Orbix applications.

Client Side

To add service context information to all requests leaving a client, do the following:

1. Call the `useServiceContext()` method to switch on `ServiceContexts`.
2. In the user code, derive some classes from the base class `ServiceContextHandler`—for example, `myHandler`.
3. Create an instance of this class within the client passing it `ServiceContext_id`.
4. Register this handler instance with the ORB using `CORBA::ORB::registerPerRequestServiceContextHandler (myHandler,env)`.
5. This registration means that if any out-going requests now leave the client, the method `ServiceContextHandler::outboundRequest()` is called. As a parameter, this method is passed a reference to the request that caused the invocation.
6. Depending on what the application wants to do, the request is interrogated by the user handler class. For example, the user-handler class may indicate that the operation name is `foobar` and trigger another process to be performed.
7. In the user code of `myHandler::outboundRequest()`, create a new instance of `ServiceContext`. Populate the `context_data` part of the `ServiceContext` with information and add it to the `ServiceContextList`.

   This `ServiceContextList` is marshaled with the request message and is passed across the wire to the server. Once the handler method has completed, the ORB possesses a copy of this newly-allocated memory. This copy is deleted after the request has been marshaled.

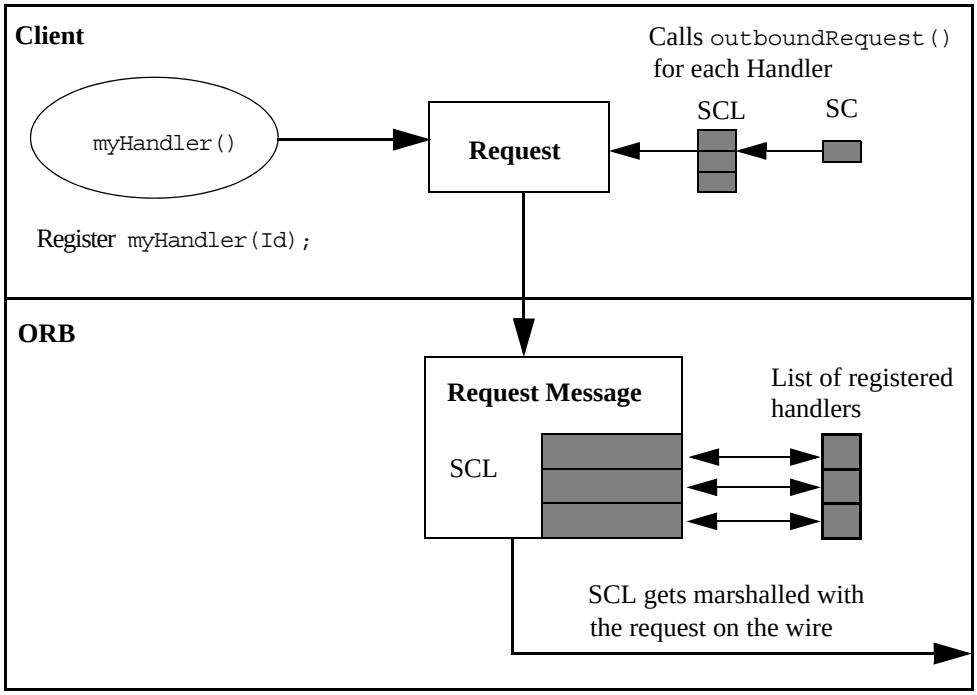


Figure 29: *Service Contexts Per Request: Client Side*

Figure 29 illustrates the operation of the service context per-request model on the client side.

The design is similar on the server side in that it creates and registers handlers, and re-implements the methods from the `ServiceContextHandler` class.

Server Side

To receive service context information from all requests entering a server, do the following:

1. Call the `useServiceContext()` method to switch on service contexts.
2. In the user code, derive some classes from the base class `ServiceContextHandler`—for example, `myHandler()`.
3. Create an instance of this class within the server, passing it the `ServiceContext_id`. You can use the same code on both the server and client sides.
4. Register this handler instance with the ORB using `CORBA::ORB::registerPerRequestServiceContextHandler (myHandler env)`.
5. This registration means that when a request comes into the server address space, the `ServiceContextList` in the request's header is unmarshalled and the incoming request methods are called on the relevant handlers.
6. Using the `incomingRequest()` method, take a copy of the `ServiceContext` required, extract whatever information is needed from it, and call whatever code is necessary.
7. After the handler has returned, and all other `ServiceContext` handlers have completed, the request continues as normal.

Note: Replies are treated the same as requests. They activate the `outboundReply()` and `incomingReply()` handlers in the same manner.

Figure 30 illustrates the operation of the service context per-request model on the server side.

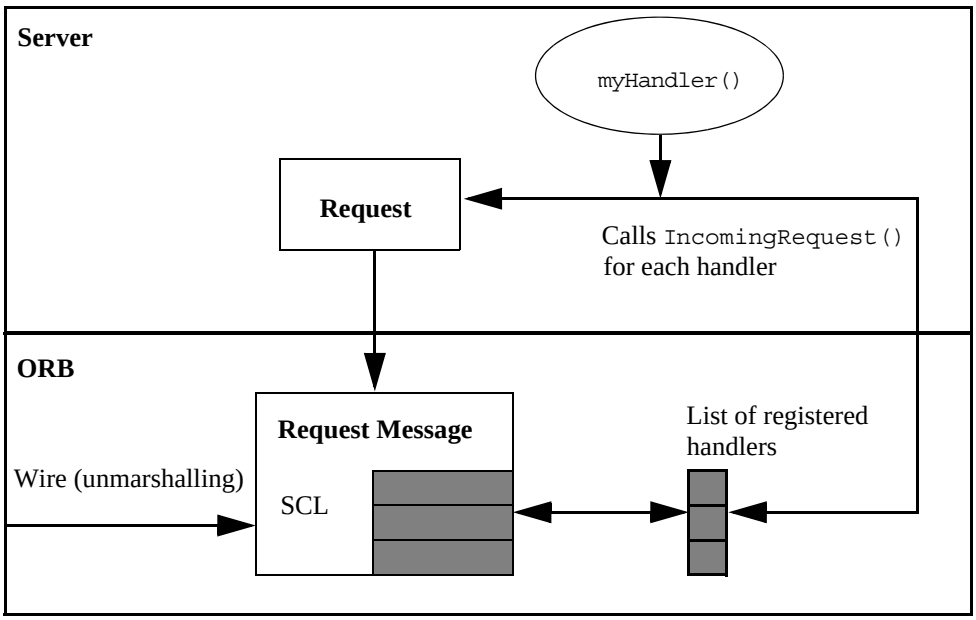


Figure 30: *Service Contexts Per Request: Server Side*

ServiceContext Per-Object Model

Consider the following overview of implementing service contexts per object in Orbix applications.

Client Side

Adding `ServiceContexts` to requests leaving the client for a particular object involves creating and registering handlers. In particular, this involves the following:

1. Calling the `registerPerObjectServiceContextHandler()` method. This method passes over the handler and object reference.
2. Converting the object reference to a stringified object reference. After a hashing algorithm is performed on it, it is inserted into a hash table.
3. Each entry in the hash table is made up of a key (stringified object reference) and a value (list of handlers).
4. Calling the `outboundRequest()` method for each object reference where any service context ID corresponds to a registered handler.
5. Each `ServiceContext` in the `ServiceContextList` has the same ID as one of the handlers registered for that object.
6. With each request, only one `ServiceContextList` gets marshaled and sent across on the wire.

Figure 31 illustrates the operation of the service context per-object model on the client side.

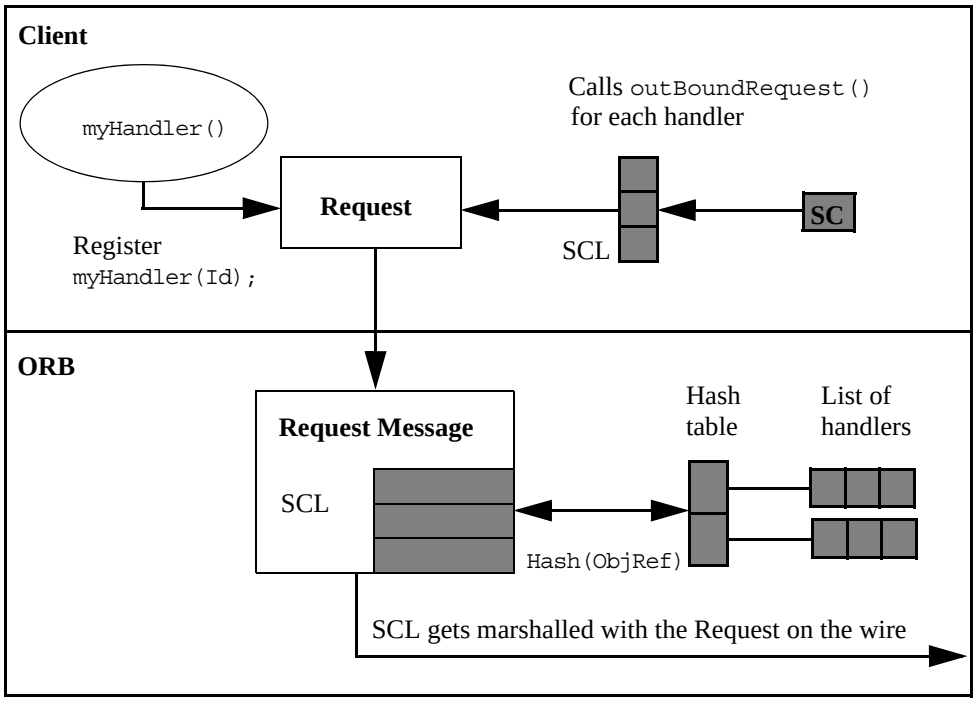


Figure 31: *Service Contexts Per-Object: Client Side*

Server Side

Receiving `ServiceContexts` from requests entering the server for a particular object also involves creating and registering handlers. In particular, this involves the following:

1. Obtaining an object reference and converting it into a stringified object reference.
2. Performing a hashing algorithm on the stringified object reference and searching for it in a populated list of handlers.
3. Calling the `incomingRequest()` method for any service context ID that corresponds to a registered handler.

Figure 32 illustrates the operation of the service context per-object model on the server side.

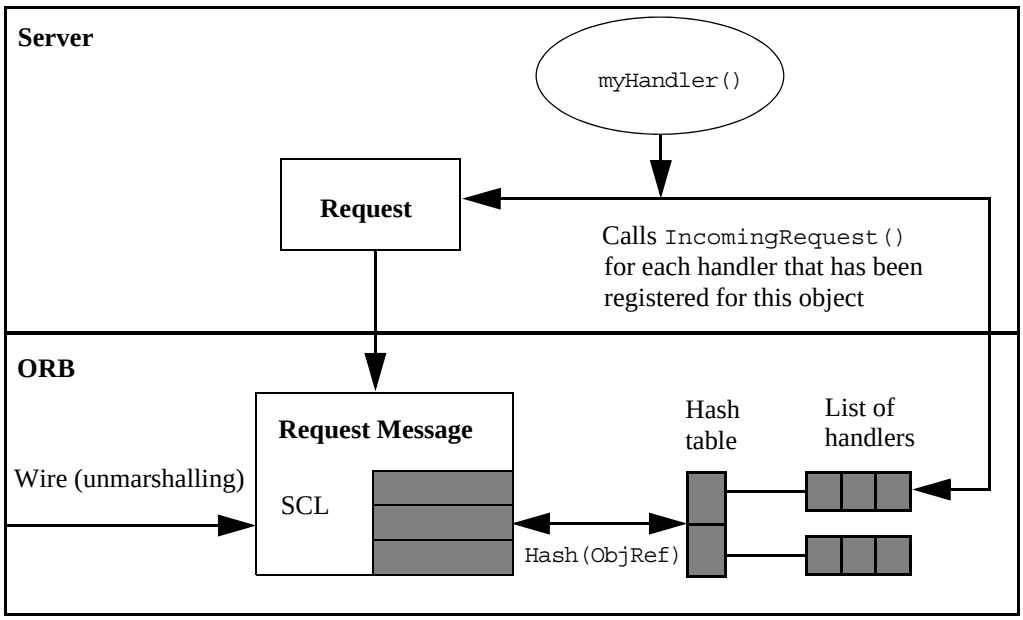


Figure 32: *Service Contexts Per Object: Server Side*

Main Components of the Service Context Model

The `ServiceContext` per-request and `ServiceContext` per-object models comprise a number of components. This section defines each of the components and explains how they fit together.

ServiceContextHandler

This base class is for users to define their own handlers for a particular `ServiceContext` ID that they want to deal with. For each `ServiceContext` you wish to handle, there is a handler registered on the client and the server. The handlers are recognized by their ID which corresponds to the ID of the `ServiceContext` they are handling.

PerRequestServiceContextHandler

This is a `ServiceContextHandler` that has been registered as a handler for all requests on the client or server side. The user derives from the base class, registers the handler, and the handler is recognized by its ID—which corresponds to the ID of the `ServiceContext` that it handles.

PerObjectServiceContextHandler

This is a `ServiceContextHandler` that has been registered as a handler for all requests to a particular object on the client or server side. The user derives from the base class, registers the handler, and the handler is recognized by its ID—which corresponds to the ID of the `ServiceContext` that it handles.

Note: The code in the handler describes what you would do with the service context data in the service context.

PerRequestServiceContextHandlerList

This is a list of service context handlers. For all requests or replies leaving an address space, all the outbound methods in all handlers are called. This is because you do not know which `ServiceContext` to add to each request.

PerObjectServiceContextHandlerList

This works the same way as `PerRequestServiceContextHandlerList` except that only requests and replies pertaining to a particular object are tagged and their `ServiceContext` information investigated. This is actually a list indexed by both the context ID and the `CORBA::Object` that it references.

Service Context Handlers and Filter points

Service context handlers interact with Orbix filter points. In Orbix, there are `10` filter points including the in reply and out reply failure filter points. Refer to "Filtering Operation Calls" for more details.

The service context mechanism provides four more points for interaction with requests and replies in a typical invocation. Figure 33 on page 315 shows the location of the `ServiceContextHandlers` in an invocation, the subsequent reply, and the order in which they are called.

You should note the following:

- If an exception is thrown in any of the `outRequest()` pre or post- marshall filter points on the client side, the `incomingReplyHandler()` is not called.
- One-way calls do not return anything. Thus they do not call the client side `inboundReplyHandler`.

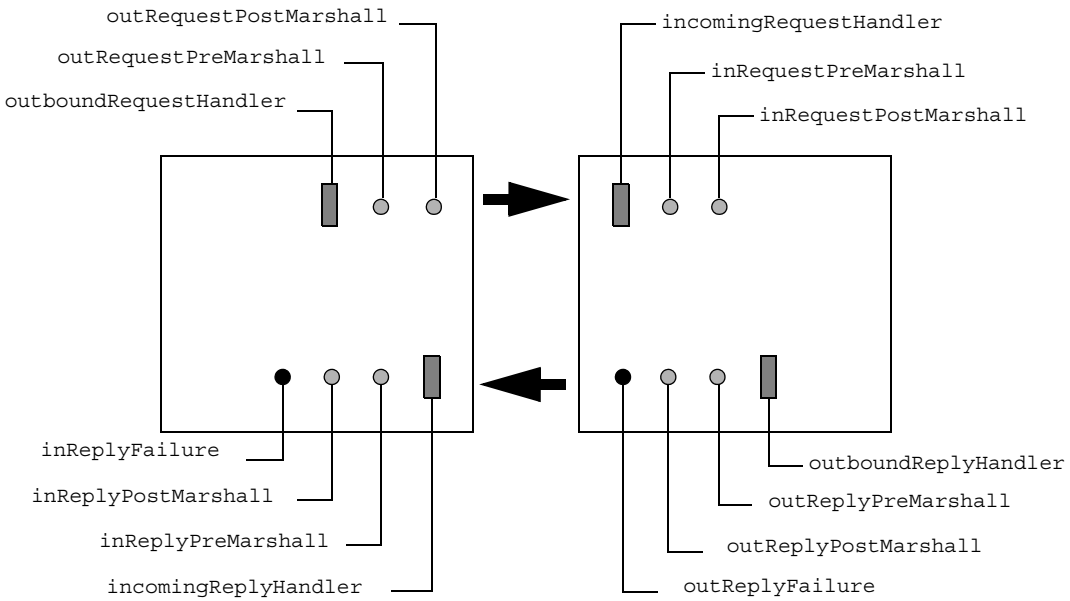


Figure 33: *Service Context Handlers and Filter Points*

For an example of using service contexts, refer to the `demos\servicecontext` directory of your Orbix installation.

Part V

Appendix

In this part

This part contains the following:

Orbix IDL Compiler Options

This appendix describes the command-line switches to the Orbix IDL compiler. The IDL compiler command is `idl`. This command accepts the following switches:

IDL Compiler Switch	Description
`-A`	Required if the IDL file contains the definition of a `struct`, `union`, `sequence`, or object reference, an instance of which can be contained directly in an `any`—that is, returned by `CORBA::Any::value()`. The structs defined by the Interface Repository can be passed as a component of a parameter of type `any` without specifying the `-A` switch.
`-B`	Required if you use the BOAImpl approach to implement the interfaces in the IDL file. The `-B` switch requests the generation of BOAImpl classes for each interface. You can use the TIE approach to implement any of the interfaces in the IDL file whether or not the `-B` switch has been specified. Clients are not affected by whether or not `-B` is specified.
`-Bonly`	The `-Bonly` switch has the same effect as the `-B` flag, but it also represses the generation of TIE code.
`-bound_seq_check`	Generate additional index-checking code to identify the location of an out-of-bounds error in a bounded sequence. By default this is not enabled.
`-c <extension>`	Specifies the file extension to be used when generating the client source file from the IDL file. The default is `C.C`, `C.cc` or `C.cpp` depending on the target C++ compiler.
`-D <name>`	Predefine <`name`> as a macro with definition.
`-E`	Only run the Orbix IDL pre-processor. Do not pass the output of the pre-processor to the Orbix IDL compiler, but output the pre-processed file to the standard output. By default, the output of the Orbix IDL pre-processor is sent to the Orbix IDL compiler.

IDL Compiler Switch	Description
-f	Do not suppress code generation for sequence<octet> and sequence<string> types. These are normally suppressed because their code is included in <CORBA.h>, and generation would lead to duplicate definitions. This switch is rarely needed. You should also refer to the -i switch.
-F	Generate per-object filtering code.
-flags	Display option information.
-h <extension>	Specifies the file extension to be used when generating the header file for the IDL file. This is .hh by default.
-i	Force insertion of sequence<octet>/sequence<string> types into the IDL parse tree. This switch is normally be used in conjunction with the -f switch. However, it is rarely needed because CORBA.h provides such support.
-I	Specify #include search path.
-K	Required if the IDL file uses the opaque type specifier.
-k23	Do not allow use of CORBA 2.3 IDL keywords as identifiers.
-l	Language mapping = C++, Java, Ada.
-L	Legacy orbix_release_no = 121
-M <filename>	Required if more than one IDL file in an application uses IDL sequences of the same type. The function definitions for sequences are then output to the specified file. This file must be compiled and then linked into the client and server. Each run of the IDL compiler *appends* to the end of the specified file, so this file should be deleted when the directory is cleaned up. (This switch is useful where sequences of the same type are generated from multiple idl sources.)
-N	Specifies that the IDL compiler is to compile and produce code for included files (files included using the #include directive). Without the -N switch, included files are compiled but no code is output.
-O	Generates a makefile rule, describing the dependencies.
-oc <path>	Specify the target directory for the client stubs. This flag overrides the -out switch.

IDL Compiler Switch	Description
-os <path>	Specify the target directory for the server stubs. This flag overrides the -out switch.
-out <path>	Specify the target directory for the client and server stubs. The -oc and -os flags override the -out flag.
-P <filename>	Allows the statements specified in <filename> to be executed *before* the standard proxy code. Its effect is to insert a #include directive at the appropriate place in the proxy code. <filename> can be any string acceptable to the #include directive. A filename enclosed in angle brackets (for example, <file.h>) denotes a standard include file; while a filename enclosed in quotes (for example, "../../file.h") denotes a file elsewhere. In some cases, this can be an alternative to writing smart proxies.
-pragma	Display fully scoped IDL type identifiers with their repository IDs.
-Q <dbIntegration>	Provides support for integration with database management systems. Valid values for <dbIntegration> depend on the database management system as specified in the relevant documentation provided with the database adapter.
-S	Specifies that the compiler is to produce files with the initial coding of the implementation classes for the IDL interfaces in the file. Two output files are produced as follows: <filenamePrefix>.ih This gives the definition of the class and the declaration of its member functions. The name of the class is the name of the IDL interface with "==" appended. This name must be changed before the file compiles. <filenamePrefix>.ic This gives the definition of the member functions—each with a blank body. Once again, the name of the class needs to be changed. The .ih file #includes the normal header file (by default .hh) produced by the compiler. The .ic file includes its corresponding .ih header file.
-s <extension>	Specify the file extension to be used when generating the server source file from the IDL file. The default is S.C, S.cc or S.cpp depending on the target C++ compiler.
-silent	Stop output of the IDL compiler filename.

IDL Compiler Switch	Description
-two_arg_def_tie	Generate two argument version of the DEF_TIE_XXX macros. This flag is required to associate scoped classes with your TIE classes.
-typeCode	Used with the -A switch to indicate that legacy TypeCodes should be generated.
-U <name>	Undefine the macro name. If -U is specified for a macro name, that macro name is not defined even if -D is used to define it.
-v	Output version information for the IDL compiler and the version number of the C++ compiler supported.
-z	Display parse tree only.

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W