VISIGENIC

Programmer's Guide

Version 2.0

VisiBroker for C++

Release, 10/15/96

The *VisiBroker for C++ Programmer's Guide* provides information on developing distributed object-based applications. The Preface lists the contents of the *VisiBroker for C++ Programmer's Guide*, describes typographic and syntax conventions used throughout the manual and provides references for more information about CORBA.

ORGANIZATION OF THIS MANUAL

This manual includes the following sections:

- Chapter 1,"VisiBroker Basics", introduces CORBA concepts and describes the software development process using VisiBroker for C++.
- Chapter 2, "Getting Started", provides in-depth descriptions of application development using an example application.
- Chapter 3, "Naming and Binding to Objects", describes how objects are identified and located by client applications.
- Chapter 4, "Object and Implementation Activation", discusses how objects are implemented and made available for use by client applications.
- Chapter 5, "The ORB Smart Agent", describes the directory service agent and its features.
- Chapter 6, "Error Handling", provides detailed information on handling error with C++ exceptions and Environments.
- Chapter 7, "Handling Events", describes the VisiBroker event handling mechanism.
- Chapter 8, "Advanced Programming Topics", discusses multi-threaded programming and how to integrate event processing with other event-based services.
- Chapter 9, "Dynamic Interfaces", describes the how to dynamically obtain object interfaces and build requests.
- Chapter 10, "The IDL Compiler", describes the VisiBroker IDL compiler for C++.
- Chapter 11, "IDL to C++ Language Mapping", describes the language mapping for C++.
- Chapter 12, "Parameter Passing Rules", describes the conventions for passing parameters.
- Appendix A provides information about platforms without C++ exception support.

Typographic Conventions

This manual uses the following conventions:

CONVENTION	USED FOR
boldface	Bold type indicates that syntax should be typed exactly as shown. For UNIX, used to indicate database names, filenames, and similar terms.
italics	Italics indicates information that the user or application provides, such as variables in syntax diagrams. It is also used to introduce new terms.
`computer`	Computer typeface is used for sample command lines and code.
UPPER CASE	Uppercase letters indicate Windows file names.
[]	Brackets indicate optional items.
...	An ellipsis indicates that the previous argument can be repeated.
|	A vertical bar separates two mutually exclusive choices.
. . .	A column of three dots indicates the continuation of previous lines of code.

Platform Conventions

This manual uses the following symbols to indicate that information is platform-specific:

All Windows platforms including Windows 3.1, Windows NT, and Windows 95

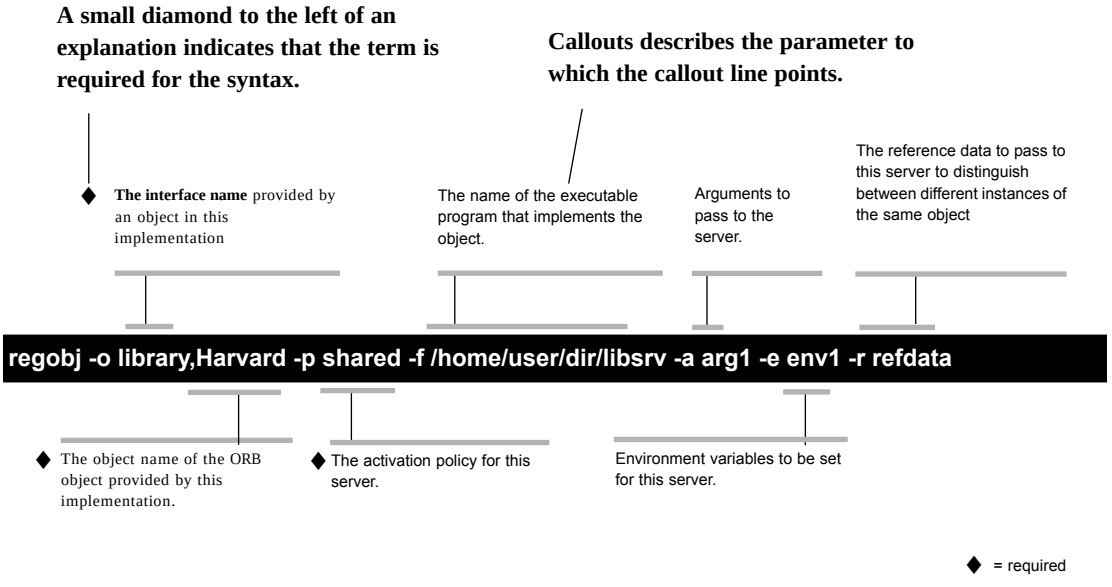
Windows NT only

Windows 95 only

All UNIX platforms

Syntax Conventions

This manual also uses the following style for command syntax descriptions.



THIS STYLISTIC DEVICE...	INDICATES THIS SYNTACTICAL MEANING...
▪ ▪ ▪	These small squares indicate the continuation of code (this device is not shown in the previous syntax diagram).
♦	A small diamond to the left of an explanation indicates that the term is required for the syntax.
[;UID=*user-name*]	Brackets around a term indicate that the term is optional. (An example of this is not shown in the previous syntax diagram.)
data-source-name	A term in italics is a variable, or something for which you must supply a definition.

Table 0-1 Stylistic devices used and their meaning.

Where to Find Additional Information

For more information about *VisiBroker for C++*, refer to the following information sources:

- *VisiBroker for C++* ***Reference Guide***. This guide contains the information on developing distributed object applications in C++ for Windows and UNIX platforms.
- *VisiBroker for C++* ***Installation Guide***. This guide contains the instructions for installing *VisiBroker for C++* on Windows and UNIX.
- *VisiBroker for C++* ***Release Notes.*** These notes contain late-breaking information about the current release of *VisiBroker for C++*.

For more information about the CORBA specification, refer to the following sources:

- ***The Common Object Request Broker: Architecture and Specification*** - 96-03-04. This document is available from the Object Management Group and describes the architectural details of CORBA. You can access the CORBA specification using the World Wide Web at the following URL: www.omg.org/corbask.htm.
- ***IDL to C++ Language Mapping*** - 94-9-14. This document is available from the Object Management Group and describes the Interface Definition Language mappings for C++.

CONTACTING VISIGENIC TECHNICAL SUPPORT

Visigenic offers a variety of support options to help you get the most from your Visigenic products. For information about these options, see the service and support information available in the “Services” section of Visigenic’s web site at **http://www.visigenic.com** or contact our Sales Department at 1-800-632-2864. If you have purchased Premium or Incident Support for your Visigenic products, Visigenic's Technical Support group can be reached at:

- Phone: 415-286-1700
- E-mail: support@visigenic.com
- Fax: 415-286-2475

Please be prepared to provide complete information about your environment, the version of the Visigenic product you are using, and a detailed description of the problem you are having.

CHAPTER 1

VISIBROKER BASICS

This chapter introduces VisiBroker for C++, a complete implementation of the CORBA 2.0 specification for developing distributed object-based applications. It includes the following major sections:

WHAT IS CORBA?

The Common Object Request Broker Architecture (CORBA) specification was developed by the Object Management Group to address the complexity and high cost of developing software applications. CORBA specifies an object oriented approach to creating software components that can be reused and shared between applications. Each object encapsulates the details of its inner workings and presents a well defined interface, which reduces application complexity. The cost of developing applications is also reduced because once an object is implemented and tested, it may be used over and over again.

The Object Request Broker (ORB) in Figure 1-1 connects a client application with the objects it wishes to use. The client application does not need to know whether the object resides on the same computer or is located on a remote computer somewhere on the network. The client application only needs to know the object's name and understand how to use the object's interface. The ORB takes care of the details of locating the object, routing the request and returning the result.

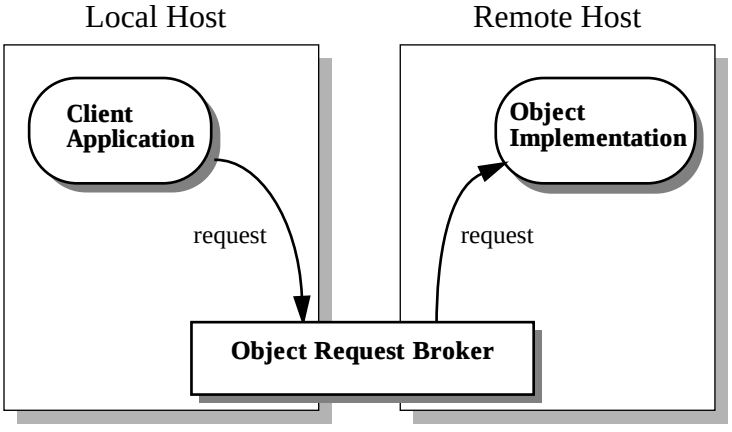


Figure 1-1 Client application acting on an object through an ORB.

Accessing Distributed Objects

To use an object, your application must first ***bind*** **itself** to the object, specifying the object's interface name. The ORB locates the host that offers an object with the requested interface name. If the server that implements the requested object is not currently executing, the ORB can ensure that the appropriate server is started. After the bind is completed, the client application can invoke operations on the object. The client's method invocations are translated into requests that are sent to the object server.

WHAT IS VISIBROKER?

VisiBroker is an ORB that offers a complete implementation of the CORBA specification. VisiBroker makes it easy for you to develop distributed, object-based client applications and servers. VisiBroker offers these important features.

- **Support for the C++ programming language**.
- Object naming.
- The ability to distribute objects across a network.
- Support for persistent objects.
- Support for dynamic object creation
- Interoperability with other ORB implementations.

DEVELOPING APPLICATIONS WITH VISIBROKER

The first step to creating an application with VisiBroker is to specify all of your objects interfaces using CORBA's **Interface Definition language** (IDL). The IDL mappings for the **C++** language are covered in Chapter 11.

The interface specification you create is used by the VisiBroker IDL compiler to generates stub routines for the client application and skeleton code for the object implementation. The stub routines are used by the client application for all method invocations. You use the skeleton code, along with code you write, to create the server that implements the objects.

The code for the client and object, once completed, is used as input to your **C++** compiler and linker to produce the executable client application and object server. These steps are shown in Figure 1-2 and are covered in detail in Chapter 2.

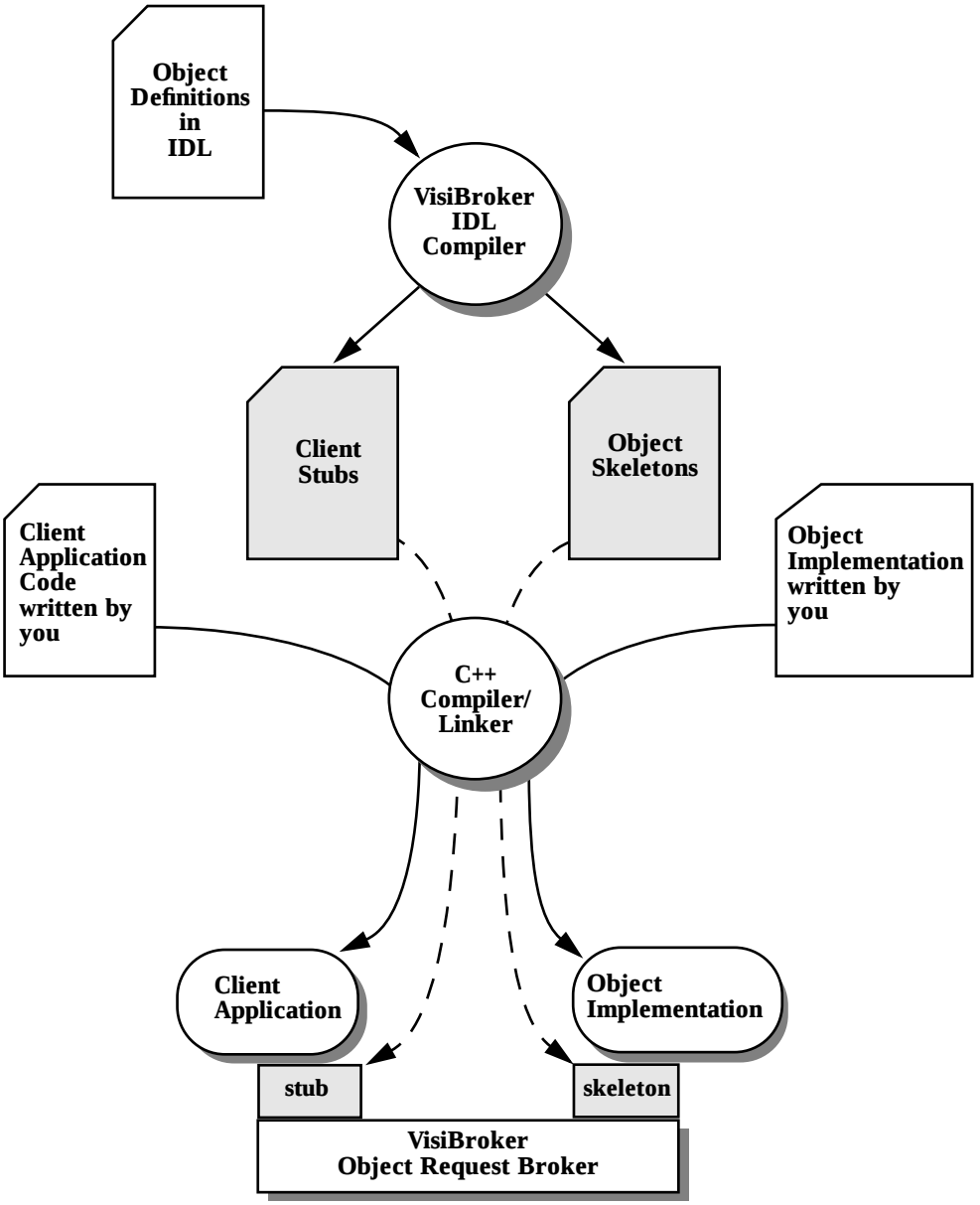


Figure 1-2 Creating an application using VisiBroker.

VISIBROKER FEATURES

In addition to providing the features defined in the CORBA specification, VisiBroker offers enhancements that increase application performance and reliability.

Fault Tolerance

VisiBroker can determine if the connection between your client application and an object server has been lost, due to a server crash or network failure. When a failure is detected, an attempt is made to restart the server or to connect your client to a suitable server on a different host. The details of fault tolerance are covered in Chapter 5.

Optimized Binding

When your application binds to an object, VisiBroker selects and establishes the most efficient communication mechanism. Depending on the platform and the location of the requested object, the bind may be established through a pointer reference, shared memory or a TCP/IP socket. Chapter 3 describes optimized binding in detail.

Dynamic Invocation Interface

VisiBroker maintains an interface repository that contains the IDL specifications for all of the objects that have been activated. This repository can be used by client applications to discover a recently added object, obtain the object's interface and dynamically construct requests to act on the object.The Dynamic Invocation Interface (DII) is covered in Chapter 9.

Support for Threads

On those platforms that support threads, VisiBroker is thread-safe and reentrant for both the client application and the server. For each thread within your client application, each bind is allocated its own thread within the server. When your client disconnects, the server thread allocated for your client will exit. Thread support is discussed in Chapter 8.

Event Handling Facilities

VisiBroker allows you to monitor the following events:

- Connection establishment and destruction.
- The entry to and exit from an object method.

You can use these events to provide debugging, logging, accounting, performance monitoring or security information for your applications. Chapter 7 describes event handling.

Event Loop Integration - For Single Threaded Applications Only

You may find that the objects you implement require interaction with an event-driven environments.Object implementations are also event driven because they must wait for client requests. VisiBroker gives you the ability to incorporate your object's event polling into the network or windowing component's event loop. This frees you from the complexity of managing nested event loops in your code. Chapter 8 explains the details of event loop integration.

CHAPTER 2

GETTING STARTED

This chapter describes the development of distributed, object-based applications with VisiBroker for **C++** on Unix platforms. A sample application is used to illustrate each step of the development process. It includes the following major sections:

THE LIBRARY APPLICATION

In this chapter, you will build a sample client application that adds a book to a library's book server. The book server could be used by a university's library to track the books in its inventory. The client application could be used by the university's purchasing department to add titles when new books arrive.

The directory **examples/library**, located within the directory where your VisiBroker package was installed, contains the files discussed in this chapter. If you do not know the location of the VisiBroker package on your system, see your administrator.

NOTE *Some of the file names used as examples in this chapter have more than eight characters and may not work with your platform.*

APPLICATION DEVELOPMENT

You will use the following steps to develop and run the sample application.

1 Identify the objects required by the application.

2 Write a specification for the objects using the Interface Definition Language (IDL).

3 Use the IDL compiler to generate the client stub code and server skeleton code.

4 Write the client application code.

5 Write the object server code.

6 Compile the client and server code.

7 Start the object server.

8 Run the client application.

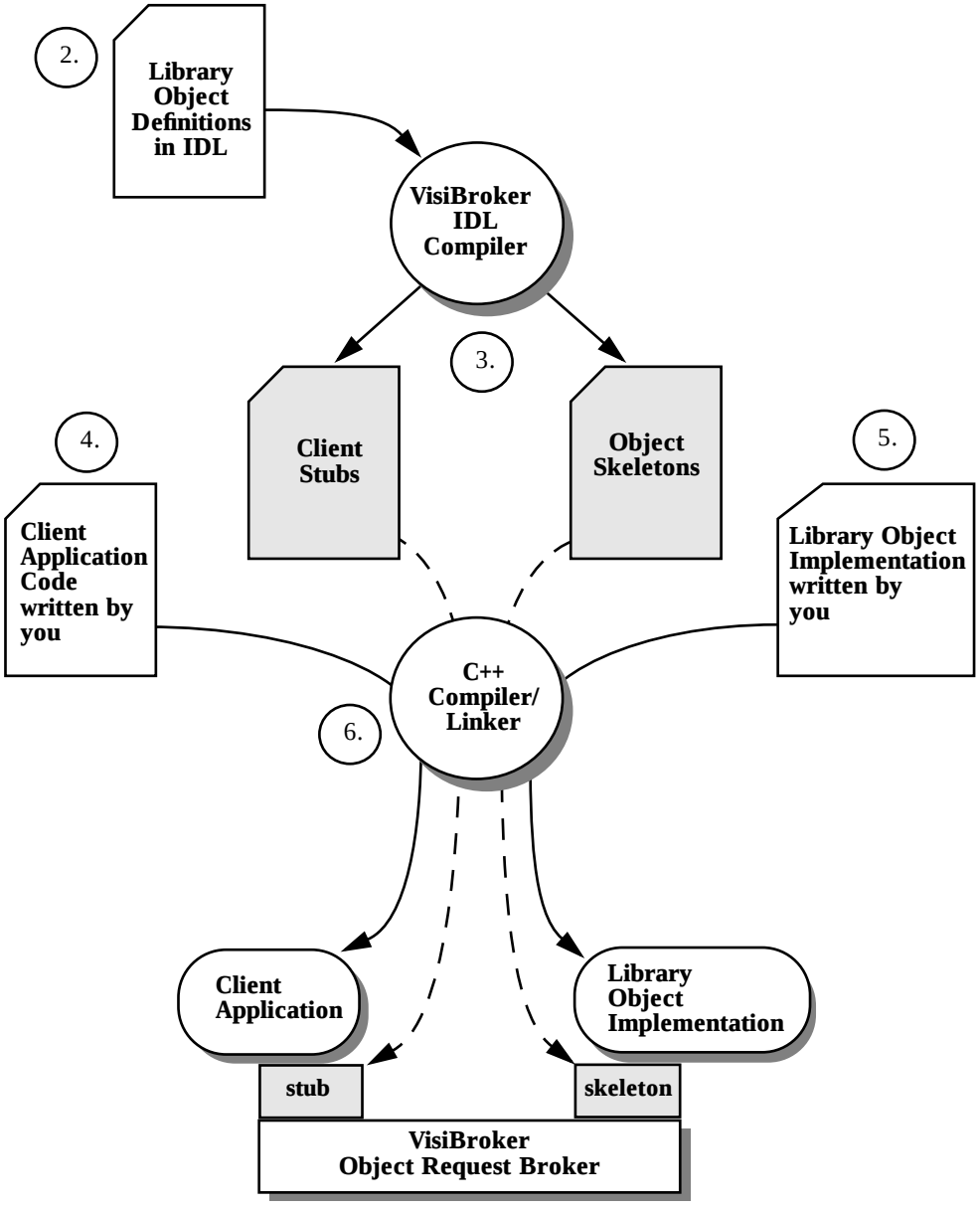


Figure 2-1 Developing the sample library book application.

The Library Object

One of the objects in your sample application is the library that contains a selection of books. A complete library server would probably offer methods to checkout and check-in books as well as add, remove and search for books. In this simple example, our `library` object will only offer a single method named `add_book` to add a book to the library.

Defining the Library Objects

The file **lib.idl**, shown below, contains the IDL specifications for the book structure and the library interface. The `book` structure consists of just two strings; one for the author of the book and one for the book's title. The `library` object's `add_book` method requires a `struct book` as its only argument.

```
struct book {
     string authors;
     string title;
};

interface library {
     boolean add_book(in book book_info);
};
```

Figure 2-2 IDL specification for the book structure and library object.

RUNNING THE IDL COMPILER

The VisiBroker IDL compiler is named *orbeline*. Since your **lib.idl** file requires no special handling, it can be complied by typing the following command.

```
prompt> orbeline lib.idl
```

For more information on the command line options for the IDL compiler, see the VisiBroker for C++ Reference Guide.

Code Generation

The IDL compiler generates four files; **lib_client.cc**, **lib_client.hh**, **lib_server.cc** and **lib_server.hh**. Two of the files are for building the client application and two are for building the object server. All generated files have either a "cc "or "hh" suffix to help you distinguish them from source files that you create, which should use the "C" and "h" extensions.

THE CLIENT FILES

The include file **lib_client.hh** contains the C++ type definitions for the `book` structure as well as a C++ definition for the `library` class. The IDL compiler also generates a `book_var` class that acts as a wrapper for the `book` structure.You may find it more convenient to use the `book_var` class rather than the `book` structure.

```
struct book {
     CORBA::String_var  author;
     CORBA::String_var  title;
     // operator= is generated for internal use
     .
     .
     .
};

class book_var
{
     public:
          book_var();
          book_var(book *ptr);
          book_var(const book_var& var);
          ~book_var();
          book_var& operator=(book *ptr);
          book_var& operator=(const book_var& var);
          book *operator->();
          operator book *();
          operator book &();
          . . .
          // other methods for internal use
     private:
          book *_ptr;
};
```

Figure 2-3 A portion of the lib_client.hh file generated by the IDL compiler.

The Library Class

The `library` class definition generated in **lib_client.hh** contains the `add_book` method specified in the IDL file, along with a variety of other methods. The **lib_client.cc** file contains the C++ implementation of methods for use by the client application as well as internally used methods. Your client application will use the `add_book` method to send an “add book” request to the library server.

```
class library: public virtual CORBA::Object
{
     private:
          ...          // methods used internally
     public:
          static library_ptr _duplicate(library_ptr obj);
          static library_ptr _nil();
          static library_ptr _narrow(CORBA::Object *obj);
          static library_ptr _bind(
                       const char *object_name = NULL,
                       const char *host_name = NULL,
                       const CORBA::BindOptions* opt=NULL);
          virtual CORBA::Boolean add_book(
                       const book& book_info);
          ...
};
```

Figure 2-4 The `library` *class.*

THE _BIND METHOD

When your application invokes the `_bind` method, the ORB locates and establishes a connection with the library server and returns a handle to the library object. If the ORB cannot locate or connect to the library server object, the `_bind` method will return `NULL` and a system exception will be raised. The binding process is described in detail in Chapter 3.

THE ADD_BOOK METHOD

The `add_book` method generated by the IDL compiler for your client application is actually a stub method. When your client application calls add_book, a request is sent to the ORB with all the necessary parameters. The ORB ensures that the request is sent to the library server object. Once the method is executed on the server, the ORB returns the results to your client application.

OTHER METHODS

Several other methods are provided that allow your client application to duplicate, initialize and narrow a `library` object reference. These methods are not used in the example client application, but they are discussed in detail in Chapter 3.

The library_var Class

A class named `library_var` is also generated by the IDL complier, though it is not used in the example application. The `library_var` class adds the ability to automatically delete object references when the object is deleted or re-initialized. The `_var` classes are described in detail in Chapter 10.

The Server Files

The include file **lib_server.hh** contains the C++ definitions for the `_sk_library` class that you use to derive the implementation of the library object server. This class contains a skeleton method `_add_book`. This skeleton method is used by the ORB on the server side to unpack the parameters from your client application's "add book" request and invoke the actual `add_book` method on the server object.

The add_book method is a pure virtual function. You create the actual implementation of this method for the library server object. The following excerpt shows the `_sk_library` class definition contained in the **lib_server.hh** file.

The **lib_server.cc** file contains the implementation for the `_add_book` method and other methods that are used internally by the ORB.

```
class _sk_library: public library
{
     ...
     public:
          ...
          /* The following operations need to be implemented
           * by the server */
          virtual CORBA::Boolean add_book(
                              const book& book_info) = 0;

          /* The following operations are implemented
           * automatically */
          static void _add_book(void *obj,
                       CORBA::MarshalStream &strm,
                       CORBA::Principal_ptr principal,
                       const char *oper);
};
```

Figure 2-5 The `_sk_library` *class.*

IMPLEMENTING THE SERVER

There are two tasks you must complete to implement the library object server; create the server's `Library` class and implement the `main` routine. To create the server's `Library` class, you must first understand its relationship with the client's `library` class and the `_sk_library` class.

The library Class Hierarchy

The `Library` server class that you implement is derived from the `_sk_library` class that was generated by the IDL compiler. Look closely at the `_sk_library` class definition and notice it is derived from the `library` class defined in the **lib_client.hh** file. Figure 2-6 shows the class hierarchy.

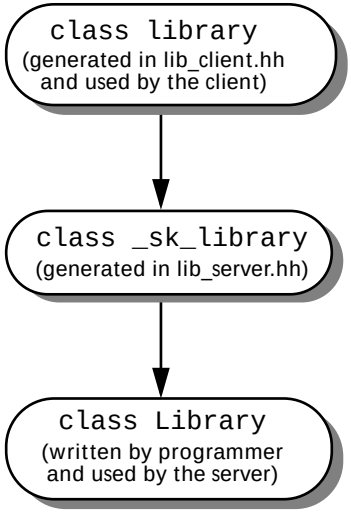


Figure 2-6 Class hierarchy for the library interface.

Creating the Library Class

The `Library` class is the actual implementation of the library object, defined in the **libsrv.h** file. This class uses a `book_list` class to provide a fixed array of `book` structures. The method `add_to_list` is called by the `Library::add_book` method. This file is not generated by the IDL compiler.

```
#include <lib_server.hh>

const CORBA::ULong MAX_BOOKS = 3;

class book_list
{
     private:
          short _book_count;
          book* _book_array[MAX_BOOKS];
     public:
          book_list() { _book_count = 0; }
          ~book_list() {
               for (int i=0; i < _book_count; i++) {
                    delete _book_array[i];
               }
          }
          CORBA::Boolean add_to_list (const book &bk) {
               if (_book_count >= MAX_BOOKS)
                    return 0;
               } else {
                    _book_array[_book_count] = new book(bk);
                    _book_count++;
                    return 1
          }
};

class Library: public _sk_library
{
     private:
          book_list bk_list;
     public:
          Library(const char *object_name = NULL);
          CORBA::Boolean add_book(const book& book_info);
};
```

Figure 2-7 The `Library` *and* `book_list` *classes.*

IMPLEMENTING LIBRARY METHODS

You must provide implementations for the server's `Library::add_book` method as well as the object's constructor. In our example these implementations are placed in the **lib_srvr.C** file, along with the `main` routine. This code is not generated by the IDL compiler.

The `Library::Library` constructor must call the `_sk_library` constructor to perform internal initialization and to register the object's interface with the ORB.

The `add_book` method involves a simple call to the `bk_list` object's `add_to_list` method, the `Library` class' internal representation of the list of books.

```
#include <lib_server.hh>
int main(int argc, char *const *argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_var boa = orb->BOA_init(argc, argv);

     // Instantiate Library Class
     Library library_server;

     // Notify BOA that object is ready
     boa->obj_is_ready(&library_server);

     // Begin event loop of receving messages
     boa->impl_is_ready();
     return(1);
}


// Library constructor
Library::Library(const char *object_name) :
                       _sk_library(object_name)
{
}

// Add book method
CORBA::Boolean Library::add_book(const book& book_info)
{
     return bk_list.add_to_list(book_info);
}
```

Figure 2-8 The `Library` *server implementation.*

THE MAIN ROUTINE

Before instantiating the `Library` object, the main routine must make two calls; one to the ORB and the other to the Basic Object Adaptor (BOA). The BOA is the interface between the object implementation and the ORB. The BOA allows your object to notify the ORB when it is ready to accept client requests.

The `argc` and `argv` parameters are the same parameters passed to the main routine. These parameters are described in “Advanced Networking Options” on page 5-9 and can be used to specify options for the ORB and BOA.

After instantiating the `Library` object, the server tells the BOA that the object is ready by invoking the `obj_is_ready` method.

Lastly, the server calls the `impl_is_ready` method to start the event loop that receives client requests. The details of the event loop are discussed in Chapter 4.

IMPLEMENTING THE CLIENT

The file named **lib_clnt.C** contains the library client application. The application accepts two parameters, the author and title of a book. These parameters will be used to initialize a `book` structure which will then be used as a parameter to the `library` object’s `add_book` method. Since your application will use the `library` class, it must include the **lib_clnt**.h file.

INITIALIZATION

The first thing your client application needs to do is initialize the ORB.

```
#include <iostream.h>
#include <lib_client.hh>
main(int argc, char *const *argv)
{
     CORBA::Boolean    ret;
     // Initialize the ORB
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
...
```

Figure 2-9 Initializing the ORB.

CHECKING THE PARAMETERS

After initializing the ORB, your application validates the author and book title parameters and creates a `book` structure. For more information on using `argc` and `argv`, see “ORB_init Options” on page 5-9. Notice that `argv[1]` and `argv[2]` are cast to `const char *`. This casting causes memory to be allocated automatically because the author and title are defined as `String_var` types (see Figure 2-3).

```
...
if(argc < 3) {
     cout << “You must specify an author and title”
          << endl;
     return(0);
}
book book_entry;
book_entry.author = (const char *)argv[1];
book_entry.title = (const char *)argv[2];
...
```

Figure 2-10 Checking the author and title input parameters.

BINDING TO THE LIBRARY SERVER

Before your client application invokes the `add_book` method, it must first invoke the `_bind` method. The implementation of the `_bind` method is generated automatically by the IDL complier. The `_bind` method requests the ORB to locate and establish a connection to the library server. If the server is successfully located and a connection is established, a proxy object is created to represent the server’s `Library` object. It is a reference to the proxy object that is returned to your client application.

If the `_bind` method fails, a system exception is raised. You should use the `try` and `catch` statements to detect any failures, print a message and exit the client application.

```
...
library *library_object;

try {
          library_object = library::bind();
}
catch(const CORBA::Exception& excep) {
          cout << “Error binding to library object” << endl;
          return(0);
}
...
```

Figure 2-11 Binding to the `library` *object.*

NOTE *If your platform's C++ compiler does not support* `try` *and* `catch`*, you can use the VisiBroker macros* `PMCTRY` *and* `PMCCATCH`*, described in Appendix A of this guide.*

ADDING THE BOOK

The invocation of the `add_book` method, like the `_bind` method, should use `try` and `catch` to handle any exceptions that may be raised. The `add_book` method on the client side is actually a stub generated by the IDL compiler that marshals all the data required for the request so that it can be sent to the object server.

```
...
try {
          ret = library_object->add_book(book_entry);
}
catch(const CORBA::Exception& excep) {
          cout << “Error adding book” << endl;
          CORBA::release(library_object);
          return(0);
}
...
```

Figure 2-12 Invoking the `add_book` *method.*

```
#include <instream.h>
#include <lib_client.hh>
main(int argc, char *const *argv)
{
     CORBA::Boolean    ret;
     // Initialize the ORB
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     if(argc < 3) {
          cout << “You must specify an author and title”
          << endl;
     return(0);
     }
     book book_entry;
     book_entry.author = (const char *)argv[1];
     book_entry.title = (const char *)argv[2];

     // Declare the library object
     library *library_object;

     try {
          // Locate object and return a pointer to it
          library_object = library::_bind();
     }
     // Check for errors
     catch(const CORBA::Exception& excep) {
          cout << “Error binding to library object” << endl;
          return(0);
     }

     // perform the add_book invocation on library_object
     try {
          ret = library_object->add_book(book_entry);
     }
     // Check for errors
     catch(const CORBA::Exception& excep) {
          cout << “Error adding book” << endl;
          CORBA::release(library_object);
          return(0);
     }
     if(ret == 1) {
          cout << “Book added successfully” << endl;
     } else {
          cout << “Unable to add book” << endl;
     }
     CORBA::release(library_object);
     return(1);
}
```

Figure 2-13 The complete library client application.

COMPILING THE CLIENT AND SERVER

The **lib_clnt.C** file that you created and the l**ib_client.cc** file generated by the IDL compiler to create the client application are compiled and linked together. The **lib_srvr.Clib_srvr.C** file that you created, along with the **lib_server.cc** and the **lib_client.cc** files generated by the IDL compiler, are compiled and linked to create the library server. Both the client application and the library server must be linked with the Visi-Broker **liborb** library.

Selecting a Makefile

The **library** subdirectory of the **examples** directory of your VisiBroker release contains an appropriate makefile for your platform. You may need to customize the makefile to work with your environment. Shown below is a sample makefile for the Solaris™ SPARCworks **C++** compiler.

```
CC = CC                              # set to your C++ compiler
ORBDIR = /usr/local/vbroker          # directory where VisiBroker was installed
CCINCLUDES = -I. -I$(ORBDIR)/include
CCFLAGS = $(CCINCLUDES)              # compiler flags you might need
                                     # such as "-g"
ORBLIB = -L$(ORBDIR)/lib -lorb       # The VisiBroker library (single threaded)
LDFLAGS = -lsocket -lnsl -ldl        # System libraries required by Solaris

SRCS = lib_client.cc lib_server.cc library_client.cc library_server.cc

.SUFFIXES: .o .cc .hh

.cc.o:
     $(CC) $(CCFLAGS) -c -o $@ $<

.C.o:
     $(CC) $(CCFLAGS) -c -o $@ $<

all: lib_client lib_server

library_client: lib_client.o lib_clnt.o
     $(CC) -o lib_client lib_client.o \
          lib_clnt.o $(ORBLIB) $(LDFLAGS)

library_server: lib_server.o lib_srvr.o lib_client.o
     $(CC) -o lib_server lib_server.o \
          lib_srvr.o lib_client.o $(ORBLIB) $(LDFLAGS)

clean:
     rm -f *.o *.hh *.cc core lib_client lib_server
```

Figure 2-14 Sample makefile for Solaris™ SPARCworks compiler.

RUNNING THE CLIENT AND SERVER

Now that you have compiled your client application and server, you are ready to run your first VisiBroker application. Running the client application involves these steps:

1 Set your environment variables

2 Start the OSAgent.

3 Start the library server

4 Run the library client application.

Setting the VisiBroker Environment Variables

The environment ORBELINE must be set to point to the directory that contains your VisiBroker license file.

```
prompt> setenv ORBELINE /usr/local/vbroker/adm
```

Figure 2-15 Setting the ORBELINE environment variable with csh.

```
prompt> ORBELINE=/usr/local/vbroker/adm
prompt> export ORBELINE
```

Figure 2-16 Setting the ORBELINE environment variable with the Bourne shell.

Starting the osagent

Before you run either your client application or the library server, you must first start the directory service daemon, **osagent**. The osagent is described in Chapter 5.

```
prompt> osagent &
```

Starting the Library Server

Start your library server by typing:

```
prompt> lib_server &
```

Running the Client

T o run your client and add the book Metamorphosis by Ovid, type:

```
prompt> lib_client Ovid Metamorphosis
```

CONCLUSION

Congratulations! You have just completed the library application and have been introduced to all of the basic features of VisiBroker. The remaining chapters in this guide will cover the details you will need to create more complex and powerful applications.

CHAPTER 3

NAMING AND BINDING TO OBJECTS

This chapter describes how interface names and object names are used to identify objects, the options associated with binding a client to an object implementation, and the way these object references can be manipulated. This chapter includes the following major sections:

INTERFACE AND OBJECT NAMES

When you define an object's interface in an IDL specification, you must give it an **interface name**. For example, the library object introduced in Chapter 2 was given the name "**library"** in the IDL specification.

```
interface library {
     void add_book();
};
```

The interface name is the least specific name by which an object can be identified when a client application invokes the `_bind` method. An object name may also be used to further qualify an object. For information on obtaining interface and object names from an object reference, see page 3-15.

Interface Names

You define an object's interface name when you define the object in IDL. The interface name will be registered with the VisiBroker **osagent**, when the `BOA::object_is_ready` method is called by the server that implements the object. The interface name is also the name that client applications will use to bind to an object.

Object Names

In addition to the required interface name, you may specify an optional **object name** when instantiating an object. The **VisiBroker** IDL compiler generates a `NULL` object name as a default parameter. The use of an object name is required if your client application plans to bind to more than one instance of an object at a time. Object names must be assigned at the time an object is registered with the **Object Activation Daemon**, described in Chapter 4.

Using Qualified Object Names with Servers

Consider the library example from Chapter 2 and imagine that you need to have two library objects available; one for a library at Stanford and one for the Harvard library. You may even want to implement two separate object servers, possibly on different hosts. Each server would instantiate a library object, but each would use the `Library` object's constructor that accepts an object name. Figure 3-1 shows the use of the default constructor. Figure 3-2 shows the library server changes that you would need to make to create separate library objects for Stanford and Harvard.

```
#include <lib_srv.h>
int main(int argc, char **argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_var boa = orb->BOA_init(argc, argv);

     // Instantiate the Library class
     Library library_server();

     orb->obj_is_ready(&library_server);
     ...

};
```

Figure 3-1 The default use of an object's constructor.

```
#include <lib_srv.h>
int main(int argc, char **argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_var boa = orb->BOA_init(argc, argv);

     // Instantiate Harvard Library class
     Library library_server("Harvard");
     ...

     // Or

     // Instantiate the Stanford Library class
     Library library_server("Stanford");
     ...

     orb->obj_is_ready(&library_server);
};
```

Figure 3-2 Specifying an **object name** *when instantiating an object's implementation.*

Using Fully Qualified Names

Your client application is not required to specify an object name when binding to an object if the same service is available from multiple servers or if there is only one server that implements the object. The **VisiBroker** IDL compiler generates a `NULL` parameter for the object name, by default.

Expanding the library example to represent two different libraries will require that you modify the client application's `_bind` invocation to specify a particular library object. Figure 3-3 shows the original client code used to bind to a default object. Figure 3-4 shows how you would modify the client application to specify an object name with the `_bind` call.

```
...
     // Declare the library object
     library *library_object;

     try {
          // Locate object and return a pointer to it
          library_object = library::_bind();
     }
     // Check for errors
     catch(const CORBA::Exception& excep) {
          cout << "Error binding to library object" << endl;
          return(0);
     }
...
```

Figure 3-3 The use of the `_bind` method without an object name.

```
...
     // Declare the library object
     library *library_object;

     try {
          // Locate object and return a pointer to it
          library_object = library::_bind("Harvard");
     }
     // Check for errors
     catch(const CORBA::Exception& excep) {
          cout << "Error binding to library object" << endl;
          return(0);
     }
...
```

Figure 3-4 The modified `_bind` method, using an object name.

BINDING TO OBJECTS

Before your client application can invoke methods on an object, it must first obtain a reference to the object using the `_bind` method.

NOTE *Your client application will never call the class' constructor, it will always obtain an object reference using the static* `_bind` *method.*

The _bind Process

When your client application invokes the _bind method, the ORB performs several functions on behalf of your application.

- The ORB contacts the osagent, VisiBroker's directory service, to locate an object server that is offering the specified **interface name**. If an **object name** is specified, it will be used to further qualify the directory service search.
- When an object implementation is located, the ORB attempts to establish a connection between the object implementation that was located and your client application.
- If the connection is successfully established, the ORB will create a proxy object, if necessary, and return a reference to that object.

Client and Server on Different Hosts

If the ORB determines that the requested object implementation resides on a remote host, a TCP/IP connection will be established between the client and object server. The ORB will instantiate a proxy object for your client to use. All methods invoked on the proxy object will be packaged as requests and sent to the server on the remote host. The server on the remote host will unpack the request, invoke the desired method, and send the results back to the client.

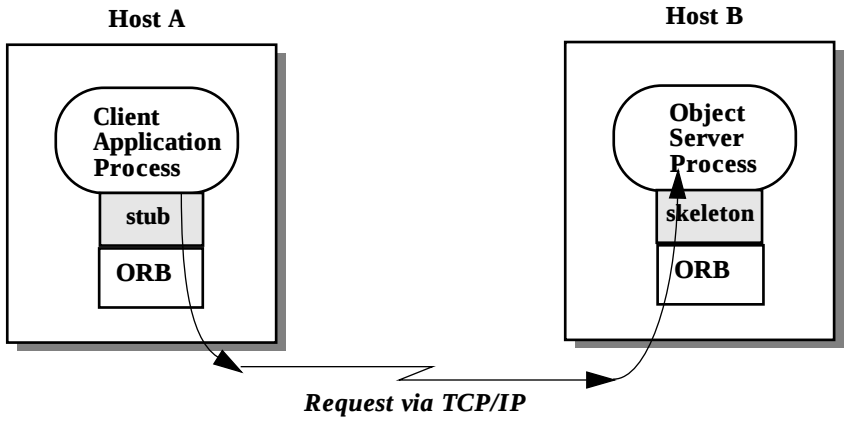


Figure 3-5 Client and Server processes on different hosts.

Client and Server on the Same Host

If the ORB determines that the requested object implementation resides on the local host, a connection will be established between the client and object server using shared memory —only if both the client and server are multithreaded. The ORB will instantiate a proxy object for your client to use. All methods invoked on the proxy object will be packaged as requests and sent to the server using shared memory.

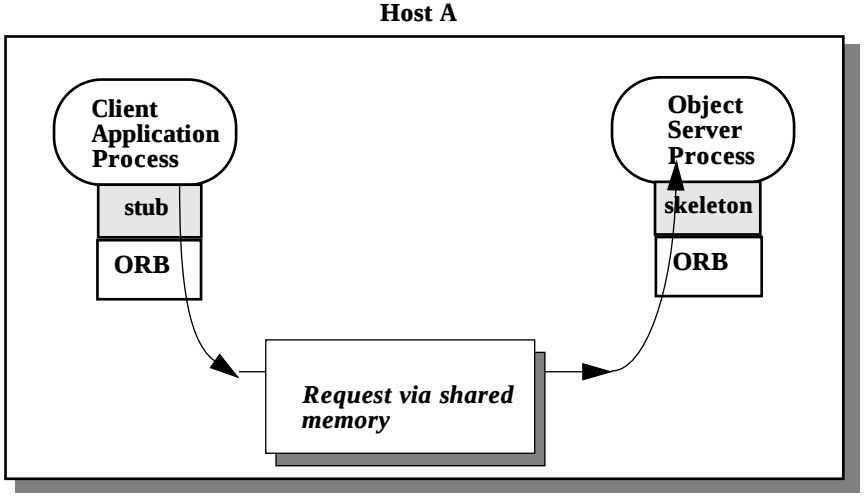


Figure 3-6 Client and Server processes on the same host.

Client and Server in the Single Process

The previous discussions have assumed that object implementations have taken the form of a server process. While this is often the case, a client application and the object implementation can both be packaged inside a single process. When your client application invokes a bind in this scenario, the ORB will return a pointer to the object implementation itself. That pointer will be widened to the object type used by your client application. All methods invoked on your client's object will get called directly as **C++** virtual functions on the object implementation. The ORB will be involved only during the bind process.

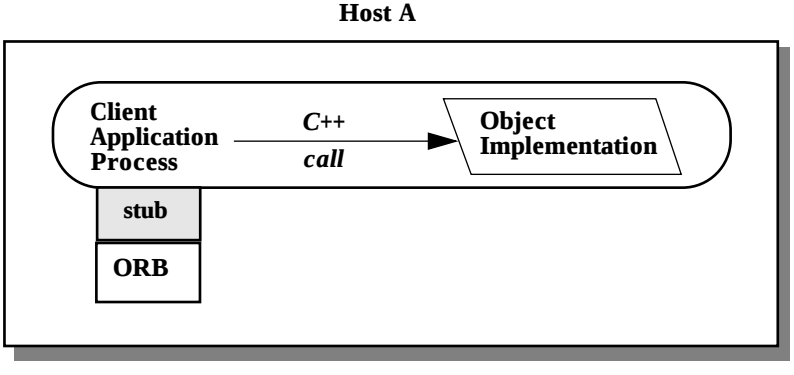


Figure 3-7 Client and object implementation in the same process.

SPECIFYING BIND OPTIONS

This section describes options that you can use to control the behavior of the `_bind` method. Figure 3-8 shows the `_bind` method generated for the library interface by the IDL compiler. The default value for all of the parameters is `NULL`.

```
class library
{
     static library_ptr _bind(
               const char *object_name = NULL,
               const char *host_name = NULL,
               const CORBA::BindOptions* opt = NULL);
     ...
};
```

Figure 3-8 The `_bind` *method generated for the* `library` *class.*

The interface name specified in the interface specification becomes the name of the class. The use of the `object_name` parameter is discussed on page 3-2.

Host Name

In addition to the object name, your client application can specify a particular host it wishes to use for the object implementation. This can be useful if your application knows that a particular object implementation is located on a particular host. If you do not specify a host name, the ORB will locate a host that meets all of the other bind parameters.

BindOptions

Figure 3-9 shows the `BindOptions` structure, used for the third parameter to the `_bind` method, which enables you to control various aspects of the connection between the client application and the object implementation. If the third parameter is `NULL`, the default bind options will be used. Each of the structure's members will be discussed in turn.

```
struct BindOptions {
          CORBA::Boolean      defer_bind;
          CORBA::Boolean      enable_rebind;
          CORBA::Long         max_bind_tries;
          CORBA::ULong        send_timeout;
          CORBA::ULong        receive_timeout;
          CORBA::ULong        connection_timeout;
};
```

Figure 3-9 The `BindOptions` *structure.*

DEFERRING BINDS

When you set `defer_bind` to `1`, the `_bind` method creates a proxy object (if necessary) and returns an object reference to your client application. A connection will not be established with the object implementation until your client application actually invokes a method on the object. If you set defer_bind to `0`, then the connection will be established when `_bind` is invoked.

The default behavior is to establish the connection at the time the `_bind` method is invoked.

ENABLING RE-BINDS

If the connection between your client application and the object implementation fails because of a network error, VisiBroker will automatically attempt to re-bind to the server process or a replica of that server. This fault tolerant processing is described in Chapter 5. If you wish to enable this re-binding process, you must set `enable_rebind` to `1`. If you wish to prevent this re-binding process, set `enable_rebind` to 0.

The default `_bind` behavior is to attempt to re-bind to the server if an error occurs.

MAXIMUM BIND ATTEMPTS

Object implementations may be registered with the Object Activation Daemon, described in Chapter 4, so that an object server process is automatically launched when your client binds to the object. You can set `max_bind_tries` to specify the number of attempts the oad should make to launch the server process.

The default `oad` behavior is to make no more than five attempts to launch a server process.

SEND TIME-OUTS

You set `send_timeout` to specify the number of seconds your client application will wait for a request to be delivered to an object server. If the time-out period expires before the message is delivered to the object server, a `CORBA::NO_RESPONSE` exception is raised.

By default, `send_timeout` is set to 0, which indicates that your client application wishes to block indefinitely.

RECEIVE TIME-OUTS

You set `receive_timeout` to specify the number of seconds your client application will wait for a response to be received from an object server. If the time-out period expires before the message is received from the object server, a `CORBA::NO_RESPONSE` exception is raised.

By default, `receive_timeout` is set to 0, which indicates that your client application wishes to block indefinitely.

CONNECTION TIME-OUTS

You set the `connection_timeout` option to specify the number of seconds your client application will wait for a connection to be established with an object server. If the time-out period expires before a connection is established, a `CORBA::NO_IMPLEMENT` exception is raised.

By default, `connection_timeout` is set to 0 to indicate that your client application wishes to use the default connection time-out.

Scope of BindOptions

VisiBroker allows you to specify three distinct levels of BindOptions. You can specify the options for each invocation of the _bind method, for a particular object reference or for all invocations of _bind by your client application.

PROCESS-LEVEL BINDOPTIONS

VisiBroker provides a global **BindOptions** structure that contains default values for the **_bind** method. These defaults are used if you do not explicitly specify a `BindOptions` parameter when you invoke the _bind method. Figure 3-10 shows the static methods you can use to query and set these defaults.

```
class Object {
         static const BindOptions  *_default_bind_options();
         static void               _default_bind_options(const BindOptions&);
    ...
};
```

Figure 3-10 Static methods for getting and setting the default bind options for a process.

BIND-LEVEL BINDOPTIONS

You can override the default, process-level bind options by passing a new `BindOptions` parameter when you invoke the `_bind` method.These new options will remain in effect for the life of the object reference returned by `_bind`, regardless of any changes to the process-level bind options.

OBJECT-LEVEL BINDOPTIONS

You can change bind options after you have invoked the `_bind` method. Figure 3-11 shows a method you can use on the object reference returned by `_bind`. This method allows you to change send and receive time-out values for any valid object reference. If you change the connection time-out and a re-bind occurs, the new connection time-out value will be apply. The bind options you set remain in effect for this object reference for as long as the reference is valid.

```
class Object {
     ...
          void          _bind_options(const CORBA::BindOptions& opt);
     ...
};
```

Figure 3-11 The method for setting object-level bind options.

OPERATIONS ON OBJECT REFERENCES

The object reference returned to your client application by the `_bind` method represents an ORB object. Your client application can use the object reference to invoke methods on the object that have been defined in the object's IDL interface specification. In addition, there are methods that all ORB objects inherit from the class `CORBA::Object` that you can use to manipulate the object.

Checking for Nil References

You can use the CORBA class static method shown to determine if an object reference is nil. This method returns `1` if the object reference passed is nil. It returns `0` if the object reference is not nil.

```
class CORBA {
          ...
          static Boolean            _nil(Object_ptr obj);
          ...
};
```

Figure 3-12 Method for checking for a nil object reference.

Obtaining a Nil Reference

You can obtain a nil object reference using the `CORBA::Object` method shown. It returns a `NULL` value that is cast to an `Object_ptr`.

```
class Object {
          ...
          static Object_ptr         _nil();
          ...
};
```

Figure 3-13 Method for obtaining a nil reference.

Duplicating a Reference

Your client application can use the `_duplicate` method to copy an object reference so that the copy can be stored in a data structure or passed as a parameter. When this method is invoked, the reference count for the object reference is incremented by one and the same object reference is returned to the caller.

The IDL compiler generates a `_duplicate` method for each object interface you specify. The `_duplicate` method shown accepts and returns a generic `Object_ptr`.

```
class Object {
          ...
          static Object_ptr         _duplicate(Object_ptr obj);
          ...
          };
};
```

Figure 3-14 The method for duplicating an object reference.

Releasing an Object Reference

You should release an object reference when it is no longer needed. One way of releasing an object reference is by invoking the `CORBA::Object` class method `_release`.

```
class CORBA {
          class Object {
          ...
                void                        _release();
          ...
          };
};
```

Figure 3-15 The method for releasing an object reference.

You may also use the `CORBA` class method `_release`, which is provided for compatibility with the CORBA specification.

```
class CORBA {
     ...
          static void               release();
     ...
};
```

Figure 3-16 The CORBA method for releasing an object reference.

Obtaining the Reference Count

Each object reference has a reference count that you can use to determine how many times the reference has been duplicated.When you first obtain an object reference by invoking `_bind`, the reference count is set to one. Releasing an object reference will decrement the reference count by one. Once the reference

count reaches 0, VisiBroker automatically deletes the object reference. Figure 3-17 shows the method for retrieving the reference count.

```
class Object {
     ...
          ULong                    _ref_count() const;
     ...
};
```

Figure 3-17 Method for obtaining the reference count.

Cloning Object References

The IDL compiler generates a `_clone` method for each object interface that you specify. Unlike the _duplicate method, _clone will create an exact copy of the object's entire state and establish a new, separate connection to the object implementation. The object reference returned and the original object reference will represent two distinct connections to the object implementation. Figure 3-18 shows the `_clone` method generated for the library interface introduced in Chapter 2.

```
class library: public virtual CORBA::Object
{
     public:
          ...
          library_ptr   _clone();
          ...
};
```

Figure 3-18 The `_clone` *method for the* `library` *class.*

Platforms that support multi-threaded client applications may increase their performance by cloning an object reference for each by each thread that is created to access a particular object. See Chapter 8 for information on multi-threaded applications.

Converting a Reference to a String

Object references are opaque and can vary from one ORB to another, so VisiBroker provides an ORB class with methods that allow you to convert an object reference to a string as well as convert a string back into an object reference. The CORBA specification refers to this process as "stringification." Figure 3-19 shows these conversion methods.

NOTE *Only object references representing persistent objects can be converted to a string. Any attempt to convert a transient object reference to a string will fail. Use the* `_is_persistent` *method to ensure that an object reference represents a persistent object before calling the* `object_to_string` *method.*

```
class ORB {
     public:
          // Convert an object reference to a string
          char          *object_to_string(Object_ptr obj);
          // Convert a char * to an object reference
          Object_ptr    string_to_object(const char *);
     ...
};
```

Figure 3-19 The methods for converting an object reference to a string and vice versa.

Obtaining Object and Interface Names

Figure 3-20 shows the methods provided by the `Object` class that you can use to obtain the interface and object names as well as the repository id associated with an object reference. The interface repository is discussed in Chapter 9 of this guide.

```
class Object {
     ...
     const char         *_interface_name() const;
     const char         *_object_name() const;
     const char         *_repository_id() const;
     ...
};
```

Figure 3-20 Methods for obtaining the interface name, object name and repository id.

Object Reference Equivalence and Casting

You can check whether an object reference is of a particular type by using the `_is_a` method. You must first obtain the repository id of the type you wish to check using the `_repository_id` method. This

method returns `1` if the object is either an instance of the type represented by `repository_id` or if it is a sub-type. `0` is returned if the object is not of the type specified.

```
class Object {
     ...
     Boolean            _is_a(const char *repository_id);
     ...
};
```

Figure 3-21 Method for determining the type of an object reference.

Figure 3-22 shows the `_is_equivalent` method which you can use to check if two object references are equivalent. This method returns `1` if the references are equivalent. This method returns 0 if the references are not identical.

```
class Object {
     ...
     Boolean            _is_equivalent(Object_ptr other_object);
     ...
};
```

Figure 3-22 Method for comparing object references.

You can use the `_hash` method shown in Figure 3-23 to obtain a hash value for an object reference. While this value is not guaranteed to be unique, it will remain consistent through the lifetime of the object reference.

```
class Object {
     ...
     ULong              _hash(ULong maximum);
     ...
};
```

Figure 3-23 The `_hash` method.

Determining the Location and State of Bound Objects

Given a valid object reference, your client application can use the method shown in Figure 3-26 to retrieve the current state of the bind for that object. The method returns `1` if the object is bound and `0` if the object is not bound.

```
class Object {
     public:
          ...
          Boolean       _is_bound() const;
          ...
};
```

Figure 3-24 The method for querying the state of the bind for an object reference.

Figure 3-25 shows two methods your client application can use after a successful `_bind` invocation to determine the location of the object implementation.

```
class Object {
          virtual Boolean     _is_local() const;
          Boolean             _is_remote() const;
     ...
};
```

Figure 3-25 Methods for determining the location of an object implementation.

NOTE *If the referred object is in the same process,* `_is_local` *returns TRUE.*

Obtaining the Current BindOptions

Given a valid object reference, your client application can use the method shown in Figure 3-26 to retrieve the bind options currently in effect for that object.

```
class Object {
     ...
          const CORBA::BindOptions _bind_options() const;
     ...
};
```

Figure 3-26 The method for retrieving an object's bind options.

WIDENING AND NARROWING OBJECT REFERENCES

Converting an object reference's type to a super-type is called **widening**. Figure 3-27 shows an example of widening a `library` pointer to an `Object` pointer. The pointer `lib` can be cast as an `Object` pointer because the `library` class inherits from the `Object` class.

```
library   *lib;
Object    *obj;

lib = library::_bind();
obj = (Object *)lib;
```

Figure 3-27 Widening an object reference.

The process of converting an object reference's type from a general super-type to a more specific sub-type is called **narrowing**. VisiBroker maintains a typegraph for each object interface so that narrowing can be accomplished by the object's `_narrow` method. If the `_narrow` method determines it is not possible to narrow an object to the type you request, it will return `NULL`.

```
library   *lib;
library   *libtwo;
Object    *obj;

lib = library::_bind();
obj = (Object *)lib;

libtwo = library::_narrow(obj);
```

Figure 3-28 Narrowing an object reference to a sub-type.

The `_narrow` method constructs a new **C++** object and returns a pointer to that object. When you no longer need the object, you must release the object reference returned by `_narrow` as well as the object reference you passed as an argument.

CHAPTER 4

OBJECT AND IMPLEMENTATION ACTIVATION

This chapter discusses how objects are implemented and made available to client applications. It includes the following major sections:

OBJECT IMPLEMENTATION

An object implementation provides the state and processing activities for the ORB objects used by client applications. An ORB object is created when its implementation class is instantiated in **C++** by an implementation process or server. An object implementation uses the **Basic Object Adaptor**, or **BOA**, to activate its ORB objects so that they can be used by client applications. ORB objects fall into two categories: transient and persistent.

Transient Objects

Objects that are only available during the lifetime of the process that created them are called **transient objects**. Transient objects are not registered with VisiBroker's directory service. Only those entities that possess an explicit object reference to a transient object may invoke its methods. Figure 4-1 shows you how to modify the library application so that `library_server` is created as a transient object. The scope must be set prior to instantiating the object.

```
#include <lib_srv.h>
int main(int argc, char **argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_var boa = orb->BOA_init(argc, argv);

     // Set local registration scope
     boa->scope(CORBA::BOA::SCOPE_LOCAL);

     // Instantiate the Library class as a transient object
     Library library_server();

     // Since Library object is transient, it won't get
     // registered with the directory service
     boa->obj_is_ready(&library_server);

     ...
};
```

Figure 4-1 Creating a transient object.

Handling Transient Object References

Clients can only access objects of transient objects when passed as an argument, as defined by the IDL; for instance, `object_to_string` will fail.

Persistent Objects

An object that remains valid beyond the lifetime of the process that created it is called a **persistent object**. These objects have a global scope and are registered with VisiBroker's directory service, which allow them to be located and used by client applications. Persistent objects may also be registered with the Object Activation Daemon, enabling the servers that implement them to be activated on demand. You can use persistent objects to implement long-running servers that provide long-term tasks. Figure 4-2 shows the creation and registration of a persistent object.

NOTE *Registration is handled by the* `boa::obj_is_ready` *method.*

```
#include <lib_srv.h>
int main(int argc, char **argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_var boa = orb->BOA_init(argc, argv);

     // Instantiate the Library class
     Library library_server();
     // Or
     // Library * library_server = new Library;

     // Register the Library object with directory service
     boa->obj_is_ready(&library_server);

     // Server is ready to receive requests
     boa->impl_is_ready();
};
```

Figure 4-2 Creating and activating a persistent object.

NOTE *You do not have to set the scope of a persistent object because the default scope is global, or persistent.*

Checking for Persistent Objects

Figure 4-3 shows a method your client application can use to determine whether a persistent or transient object implementation is associated with a given object reference. It is important to know whether or not an object is persistent because some methods for

manipulating object references will fail if the object is transient. The `_is_persistent` method returns `1` if the object is persistent and 0 if the object is transient.

```
class Object {
     ...
     Boolean            _is_persistent() const;
     ...
};
```

Figure 4-3 The method for checking for persistent object implementations.

Object Registration

Once a server has instantiated the ORB objects that it offers, the BOA must be notified when the objects have been initialized. Lastly, the BOA is notified when the server is ready to receive requests from client applications.

The `obj_is_ready` method notifies the BOA that a particular ORB object is ready to receive requests from client applications. If your server offers more than one ORB object, it must make a call to `obj_is_ready` for each object, passing the object reference as an argument.

If the object reference passed to `obj_is_ready` represents a persistent object, the BOA will register the object with VisiBroker's directory service. If the object is transient, no such registration will occur.

NOTE *When* `obj_is_ready` *is not called and a client attempts to call* `_bind`*, the exception* `NO_IMPLEMENT` *is raised.*

Once all of the objects have been instantiated and all the calls to `obj_is_ready` have been made, the server must call **`impl_is_ready`** to enter an event loop and await client requests. Chapter 7 in this guide discusses event handling in detail.

THE BASIC OBJECT ADAPTOR

VisiBroker's BOA provides several important functions to client applications and the object implementations they use. It is important to realize that an object may reside in the same process as its client application or it may reside in a separate process called a server. Servers may contain and offer a single object or multiple objects. Furthermore, servers may be activated by the BOA on demand or they may be started by some entity external to the BOA.

Object Server Activation Policies

The CORBA specification defines four activation policies that describe the way in which an object implementation is started and the manner in which it may be accessed by a client application. These activation policies only apply to persistent objects, not transient objects.

SHARED SERVER POLICY

When the shared server policy is specified, only one server is launched regardless of the number of clients; the clients share the server. Along with persistent servers, shared servers are the most common types of servers.

PERSISTENT SERVER POLICY

This policy describes servers that, like shared servers, implement multiple objects. Persistent servers are started by some entity outside of the Basic Object Adaptor, but they still register their objects and receive requests using the BOA.

UNSHARED SERVER POLICY

Unshared servers are processes that implement a single object. A client application causes this type of server to be activated. Once that client exits, the unshared server will exit.

SERVER-PER-METHOD POLICY

This activation policy requires a server process to be started for each method that is invoked. After the method has been completed, the server will exit. Subsequent method invocations on the same object will require a new server process to be started.

OBJECT ACTIVATION DAEMON

You can register an object implementation with VisiBroker's Object Activation Daemon to automatically activate the implementation when a client requests a bind to the object. Object implementations can be registered using a command-line interface or programmatically with the `BOA::create` method. There is also an ORB interface to the OAD, described in "ORB Interface to the OAD" on page 4-14. In each case, the interface name, object name, the activation policy, and the executable program representing the implementation must be specified.

The Implementation Repository

All object implementations registered with the OAD are stored in an implementation repository, maintained by the OAD. By default, the implementation repository data are stored in a file named `impl_rep`. This file's path name is dependent on where VisiBroker was installed on your system. If VisiBroker was installed in `/usr/local/visibroker/`, then the path to this file would be

`/usr/local/visibroker/adm/impl_dir/impl_rep`. These defaults can be overridden using OAD environment variables, described in the VisiBroker for **C++** Reference Guide.

OAD Registration with regobj

The `regobj` command can be used to register an object implementation from the command line or from within a script. The required parameters are the interface name, object name and path name. If the activation policy is not specified, the default policy of shared server will be used. For complete information on using this command, see the *VisiBroker for C++ Reference Guide*.

NOTE *The implementation of your object does not need to be modified in order for you to use regobj. You may write an implementation and start it manually during the development and testing phases. When your implementation is ready to be deployed, you can simply use* **regobj** *to register your implementation with the OAD.*

NOTE *When registering an object implementation, use the same object name as is used when the implementation object is contructed.*

U

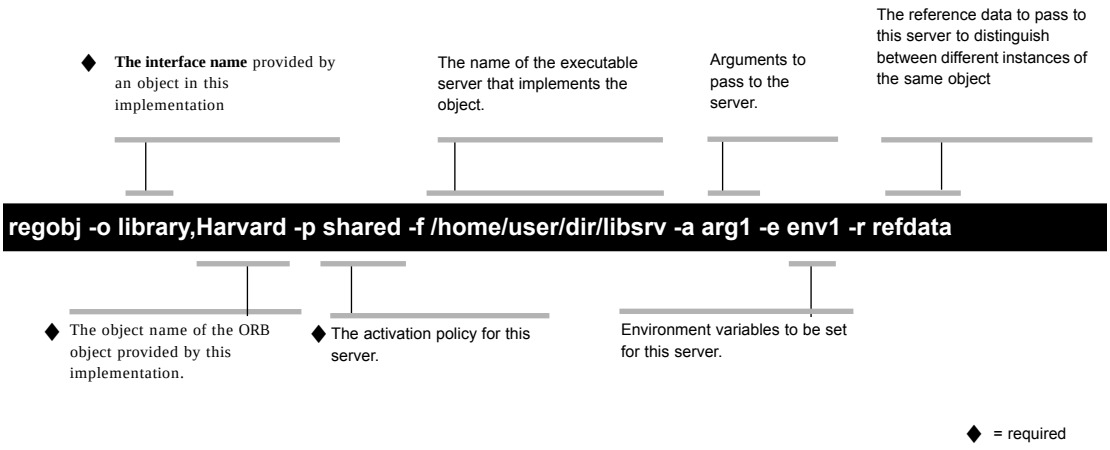


Figure 4-4 The **`regobj`** *command.*

W For information about the Windows implementation of `regobj`, see the “Commands” chapter of the *VisiBroker for C++ Reference Guide*.

OAD Registration using BOA::create

Instead of using the `regobj` command manually or in a script, VisiBroker allows applications written in C++ to use the `BOA::create` method to register one or more objects with the activation daemon. Using this method results in an object implementation being registered with the OAD and the VisiBroker directory service. The OAD will store the information in the implementation repository, allowing the object implementation to be located and activated when a client attempts to bind to the object.

```
class CORBA {
     ...
     typedef OctetSequence    ReferenceData;
     ...
     class BOA {
          virtual Object_ptr  create(
                        const ReferenceData&          ref_data,
                        InterfaceDef_ptr              inf_ptr,
                        ImplementationDef_ptr         impl_ptr) = 0;
          ...
     };
     ...
};
```

Figure 4-5 The `BOA::create` *method and its parameters.*

Reference Data Parameter

You can use the `ref_data` parameter to distinguish between multiple instances of the same object. The value of the reference data is chosen by the implementation at object creation time and remains constant during the lifetime of the object. The `ReferenceData typedef` is portable across platforms and ORBs.

NOTE *VisiBroker does not use the* `inf_ptr`*, defined by the CORBA specification to identify the interface of the object being created. Applications created with VisiBroker should always specify a* `NULL` *value for this parameter.*

Implementation Definition Parameter

The `impl_ptr` parameter supplies the information that the BOA needs to register an ORB object. The `ImplementationDef` class defines the interface name, object name, and reference id properties used by the BOA. Figure 4-6 shows the methods for querying and setting these properties.

```
class ImplementationDef
{
     public:
          static ImplementationDef_ptr    _duplicate(
                                                ImplementationDef_ptr obj);
          static void                     _release(
                                                ImplementationDef_ptr obj);
          static ImplementationDef_ptr    _nil();
          const char                      *interface_name() const;
          void                            interface_name(const char *val);
          const char                      *object_name() const;
          void                            object_name(const char *val);
          ReferenceData_ptr               id() const;
          void                            id(const ReferenceData_ptr& data);
          ...

     protected:
          String_var                      _interface_name;
          String_var                      _object_name;
          ReferenceData                   _id;
     ...
};
```

Figure 4-6 The `ImplementationDef` *class.*

The `_interface_name` property represents the name specified in the object's IDL specification. The `_object_name` property is the name of this object, provided by the implementor or the person installing the object. The `_id` property is chosen by the implementation and has no meaning to the BOA or the OAD. The implementor may use the `_id` property as they chose.

Creation Definition

The `ImplementationDef` class, as defined by the CORBA specification, does not supply all the information that the OAD needs to activate an object implementation when a client attempts to bind to the object. The `CreationImplDef` class is derived from `ImplementationDef` and adds the properties the OAD requires.The properties added are `_path_name`, `_policy`, `_args` and `_env.` Methods for setting and querying their values are also provided. These additional properties are used by the OAD to activate an ORB object. Figure 4-7 shows the `CreationImplDef` class, its properties and methods.

The `_path_name` property specifies the exact path name of the executable program that implements the object. The `_policy` property represents the server's activation policy, discussed in "Object Server Activation Policies" on page 4-5. The `_args` and `_env` properties represent optional arguments and environment settings to be passed to the server.

```
enum Policy {
     SHARED_SERVER,
     UNSHARED_SERVER,
     SERVER_PER_METHOD
};

class CreationImplDef: public ImplementationDef
{
     public:
          CreationImplDef();
          CreationImplDef(const char *interface_name,
                          const char *object_name,
                          const RefereneData& id,
                          const char *path_name,
                          const StringSequence& args,
                          const StringSequence& env);
          ~CreationImplDef() {};
          static CreationImplDef_ptr      _duplicate(CreationImplDef_ptr obj);
          static void                     _release(CreationImplDef_ptr obj);
          static CreationImplDef_ptr      _nil();
          static CreationImplDef_ptr      _narrow(ImplementationDef_ptr ptr);
          Policy                          activation_policy() const;
          void                            activation_policy(Policy p);
          const char                      *path_name() const;
          void                            path_name(const char *val);
          StringSequence                  *args() const;
          void                            *args(const StringSequence& val);
          StringSequence                  *env() const;
          void                            env(const StringSequence& val);
     ...
     protected:
          String_val          _path_name;
          Policy              _policy;
          StringSequence      _args;
          StringSequence      _env;
};
```

Figure 4-7 The `CreationImplDef` *class.*

BOA::create Example

Figure 4-8 shows how to use the `CreationImplDef` class and the `BOA::create` method to create an ORB object and register it with the OAD.

```
#include "libsrv.h"
void main(int argc, char * const * argv)
{
     CORBA::Object_ptr obj;

     // Initialize the ORB and BOA
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_var boa = orb->BOA_init(argc, argv);

     // Optional reference data
     ReferenceData id;

     CORBA::CreationImplDef impl_def("library", "Harvard", id,
                              "/usr/home/dir/libsrv",
                              NULL /* no args */, NULL /* no envs */);
     obj = boa->create(id, NULL, &impl_def);
     if (obj != NULL)
          cout << "ORB object created successfully"
     exit (1);
}
```

Figure 4-8 Creating an ORB object and registering with the OAD.

NOTE *If the* `impl_def` *parameter passed to* `BOA::create` *cannot be narrowed to a* `CreationImplDef` *reference, the create will fail and a* `CORBA::BAD_PARAM` *exception will be raised.*

Changing an ORB Implementation

Figure 4-9 shows the `BOA::change_implementation` method which can be used to dynamically change an object's implementation. You can use this method to change the object's activation policy, path name, arguments and environment variables.

If the `impl` parameter cannot be narrowed to a `CreationImplDef`, this method will fail and a `CORBA::BAD_PARAM` exception will be raised.

```
class BOA {
     ...
     virtual void change_implementation(
                         const Object &obj,
                         const ImplementationDef& impl);
     ...
};
```

Figure 4-9 The `change_implementation` *method.*

Though you can change an object's implementation name and object name with the this *method, you should exercise caution. Doing so will prevent client applications from locating the object with the old name.*

UNREGISTERING IMPLEMENTATIONS

When the services offered by an object are no longer available or temporarily suspended, the object should be unregistered with the OAD. When an ORB object is unregistered, it is removed from the OAD's list of objects. The object is also removed from the directory service and from the implementation repository. Once an object is unregistered, client applications will no longer be able to locate or use it. In addition, the **BOA::change_implementation** method will no longer be able to be used to change the object's implementation. As with the registration process, unregistering may be done with a command line or programmatically. There is also an ORB object interface to the OAD, described in "ORB Interface to the OAD" on page 4-14.

Unregistering with unregobj

Figure 4-10 shows the **unregobj** command, which can be used to unregister an object implementation from the command line or from within a script. If the interface name is specified by itself, all objects instances associated with that interface name will be unregistered. You can specify both the interface and object name if you only wish to unregister a specific object within an interface. For complete information on using this command, see the *VisiBroker for C++ Reference Guide*.

U

Figure 4-10 The unregobj command.

W For information about the Windows implementation of **unregobj**, see the "Commands" chapter of the *VisiBroker for C++ Reference Guide*.

Unregistering with the BOA::dispose Method

An object's implementation can use the **`BOA::dispose`** method to unregister an ORB object. Any connections that might exist between a client application and the object will be terminated as soon as the object is unregistered.

```
class CORBA {
     class BOA {
          ...
          virtual void       dispose(Object_ptr);
          ...
     };
};
```

Figure 4-11 The `BOA::dispose` *method.*

The listimpl Command

You can use the `listimpl` command to list the contents of a particular implementation repository. For each implementation in the repository the `listimpl` command lists all the object instance names, the path name of the executable program, the activation mode and the reference data. Any arguments or environment variables that are to be passed to the executable program are also listed. For complete details on using this command see the
VisiBroker for C++ Reference Guide.

U

Figure 4-12 The `listimpl` *command.*

W For information about the Windows implementation of **listimpl**, see the “Commands” chapter of the *VisiBroker for C++ Reference Guide*.

ORB INTERFACE TO THE OAD

The Object Activation Daemon is implemented as an ORB object. Figure 4-13 shows the IDL interface specification for the OAD. You can create a client application that binds to the OAD and uses this interface to query the status of objects that have been registered.

```
// IDL
module Activation
{
     enum State {
          ACTIVE,
          INACTIVE,
          WAITING_FOR_ACTIVATION
     };

     struct ObjectStatus {
          long         process_id;
          State        activation_state;
          Object       objRef;
     };
     typedef sequence<ObjectStatus> ObjectStatusList;

     struct ImplementationStatus {
          CORBA::CreationImplDef    impl;
          ObjectStatusList          status;
     };
     typedef sequence<ImplementationStatus> ImplStatusList;

     exception NotRegistered {};
     ...

     interface OAD {
          // Internal methods are not shown here.
          ...

          // Get status info for a given implementation
          ImplementationStatus      get_status(
                                          in string interface_name,
                                          in string object_name)
                                               raises (NotRegistered);
          // Get status of all implementations for a given interface
          ImplStatusList      get_status_interface(
                                    in string interface_name)
                                          raises (NotRegistered);
          // Get list of all registered interfaces.
          ImplStatusList      get_status_all();
     ...
```

Figure 4-13 The OAD interface specification.

ACTIVATING OBJECTS DIRECTLY

In the library example introduced in Chapter 2, the `Library` object was activated directly by the server. Direct activation of an object involves instantiating all the **C++** implementation classes, invoking the `boa::obj_is_ready` method for each object and then invoking `BOA::impl_is_ready` to begin receiving requests. Figure 4-14 shows how this processing would occur for a server offering two `Library` objects; one with the object name of “Stanford” and the other named ”Harvard.” Once the objects have been instantiated and activated, the server invokes `BOA::impl_is_ready` to begin receiving client requests.

NOTE *The* `BOA::obj_is_ready` *must be called for each object offered by the implementation.*

```
#include <lib_server.hh>
int main(int argc, char * const * argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_var boa = orb->BOA_init(argc, argv);

     // Instantiate Harvard Library Class
     Library harvard_lib(“Harvard”);
     boa->obj_is_ready(&harvard_lib);

     // Instantiate Stanford Library Class
     Library stanford_lib(“Stanford”);
     boa->obj_is_ready(&stanford_lib);

     // Begin event loop of receiving messages
     boa->impl_is_ready();
     return(1);
}
```

Figure 4-14 Server activating two objects and the implementation.

ACTIVATING OBJECTS WITH THE BOA

When you design your object implementation, you may want to defer the activation of one or more ORB objects until a client requests them. The `BOA::obj_is_ready` and `BOA::impl_is_ready` methods may be used with the `ActivationImplDef` class to instantiate objects upon receipt of a client request.

```
class CORBA {
     class BOA {
          ...
          virtual void obj_is_ready(Object_ptr,
                         ImplementationDef_ptr impl=NULL) = 0;
          virtual void impl_is_ready(ImplementationDef_ptr impl=NULL) = 0;
          ...
     };
};
```

Figure 4-15 The `BOA::obj_is_ready` *and* `BOA::impl_is_ready` *methods.*

In previous examples, the `obj_is_ready` method was only passed an object reference. The `impl_is_ready` was passed no parameters at all. It is possible to pass an `ActivationImplDef` pointer to the `obj_is_ready` method, which can be used to override the activation and deactivation methods used by the BOA. Figure 4-16 shows the `ActivationImplDef` class, which adds an `Activator` pointer and provides methods for setting and retrieving that pointer. Note that this class is derived from `ImplementationDef`.

```
class ActivationImplDef: public ImplementationDef
{
     public:
          ActivationImplDef();
          ActivationImplDef(const char *interface_name,
                         const char *object_name,
                         const ReferenceData& id,
                         Activator_ptr act);
          ~ActivationImplDef();
          static ActivationImplDef_ptr _duplicate(ActivationImplDef_ptr obj);
          static ActivationImplDef_ptr _nil();
          static ActivationImplDef_ptr _narrow(ImplementationDef_ptr ptr);
          Activator_ptr       activator_obj();
          void                activator_obj(Activator_ptr val);
     protected:
          Activator_ptr       _activator;
          ...
};
```

Figure 4-16 The `ActivationImplDef` *class.*

The Activator Class

Figure 4-17 shows the `Activator` class, which provides the two methods used by the BOA to activate and deactivate an ORB object.

```
class Activator {
     public:
          Activator();
          ~Activator();
          static Activator_ptr      _duplicate(Activator_ptr obj);
          static void               _release(Activator_ptr);
          static Activator_ptr      _nil();
          virtual Object_ptr        activate(ImplementationDef impl) = 0;
          virtual void              deactivate(Object_ptr,
                                          ImplementationDef_ptr impl);
};
```

Figure 4-17 The `Activator` *class.*

Deriving your own class from the `Activator` class lets you to override the `activate` and `deactivate` methods that the ORB will use for the `Library` object. This allows you to delay the instantiation of the `Library` object until the BOA activates the ORB object. It also allows you to provide clean-up processing when the ORB deactivates the object. Figure 4-18 shows how to create an `Activator` for the `Library` class.

```
class LibraryActivator : CORBA::Activator {
     public:
          virtual CORBA::Object_ptr activate(
                         CORBA::ImplementationDef_ptr impl);
          virtual void deactivate(CORBA::Object_ptr,
                         CORBA::ImplementationDef_ptr impl);
};

CORBA::Object_ptr LibraryActivator::activate(
                         CORBA::ImplementationDef_ptr impl)
{
     // When the BOA activates us, instantiate the Library object.
     return new Library(impl->object_name());
}

void LibraryActivator::deactivate(CORBA::Object_ptr obj,
                         CORBA::ImplementationDef_ptr impl)
{
     // When the BOA deactivates us, release the Library object.
     obj->release();
}
```

Figure 4-18 Deriving the `LibraryActivator` *class, implementing the* `activate` *and* `deactivate` *methods.*

Putting it All Together

Figure 4-19 shows how to use the `ActivationImplDef` class the `LibraryActivator` class, to defer the activation of the `Library` object until a client request is received. The instantiation of the `Library` object no longer appears in the `main` routine. Instead, the `Library` object will be instantiated when the BOA receives a client request and invokes the `activate` method.

In this example, the invocation of `BOA::obj_is_ready` is passed a `NULL` object reference as well as an `ActivationImplDef` reference. The creation of the Library object named “Harvard” will now be deferred until the first client request for that object is received.

```
void main(int argc, char * const * argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_var boa = orb->BOA_init(argc, argv);

     CORBA::ReferenceData id;
     CORBA::ActivationImplDef impl("library", "Harvard", id,
                         (CORBA::Activator_ptr) new LibraryActivator);

     // obj_is_ready is passed an ActivationImplDef object to override
     // the activation of the Library object.
     boa->obj_is_ready(NULL, &impl);

     // activate other objects
     ...

     // Begin event loop of receiving requests
     boa->impl_is_ready();
     return(1);
};
```

Figure 4-19 Using the `ActivationImplDef` *class with the* `BOA::obj_is_ready` *method.*

If an implementation has only one object, then the `impl_is_ready` method can be called with the `ActivationImplDef` reference to activate both the object and the implementation. Since `impl_is_ready` can accept only one object's implementation, this approach cannot be used if multiple objects reside in the same implementation. Figure 4-20 shows the use of a one invocation of the `impl_is_ready` method.

```
void main(int argc, char * const * argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_var boa = orb->BOA_init(argc, argv);

     CORBA::ReferenceData id;
     CORBA::ActivationImplDef impl("library", "Harvard", id,
                         (CORBA::Activator_ptr) new LibraryActivator);

     // impl_is_ready is passed an ActivationImplDef object to override
     // the activation of the Library object.
     boa->impl_is_ready(&impl);

     return(1);
};
```

Figure 4-20 The use of a single `impl_is_ready` *method.*

OBJECT AND IMPLEMENTATION DEACTIVATION

The correct approach to deactivating objects and implementations depends on how the object and implementation were activated. Objects and their implementations can be activated manually, through **C++** instantiation, by the OAD, or by the BOA.

Deactivating a Manually Started Implementation

An implementation that is started manually can be considered deactivated when the implementation exits. VisiBroker will automatically unregister the objects within that implementation from the list maintained by the directory service.

Deactivating C++ Instantiated Objects

If an object was created by instantiating its **C++** class, call `CORBA::release(Object_ptr)` on the object to unregister the object from the list of objects maintained by the directory service. This also calls the destructor if no other references to the object exist. Calling delete on the object is not recommended since the object could be deleted prematurely.

Deactivating Implementations Started by the OAD

Implementations started by the OAD can be deactivated by calling the `BOA::deactivate_impl` method. Once this method is called, the implementation will not be available to service client requests. The implementation can only be re-activated if it is restarted or if it again calls the `impl_is_ready` method.

Deactivating Objects Activated by the BOA

The `BOA::deactivate_obj` method is provided to deactivate objects activated by the BOA. After this method is called, the object will be removed from the directory service list of objects offered by that implementation. Figure 4-21 shows the definition of the `deactivate_obj` method.

```
class CORBA {
     class BOA {
          ...
          virtual void       deactivate_obj(Object_ptr);
          ...
     };
};
```

Figure 4-21 The `BOA::deactivate_obj` *method.*

CHAPTER 5

THE ORB SMART AGENT

This chapter describes the osagent, which provides directory service functions, fault tolerance and object migration facilities. It includes the following major sections:

SMART AGENT FEATURES

VisiBroker's **osagent** is a dynamic, distributed directory service that provides facilities for both client applications and object implementations. When a client application invokes the `_bind` method on an object, the osagent locates the specified implementation and object so that a connection can be established between the client and the implementation. Object implementations register their objects with the osagent so that client applications can locate and use those objects. When an object or implementation is destroyed, the osagent removes them from its list of available objects.

Agent Communication

An osagent may be started on any host. To locate an osagent, client applications and object implementations send a broadcast message, and the first osagent to respond will be used. Once an osagent has been located, a point-to-point UDP communication is established for registration and look-up requests. The UDP protocol is used because it consumes fewer network resources than a TCP connection. All registration and locate requests are dynamic, so there are no required configuration files or mappings to maintain.

NOTE *Broadcast messages are usually used only to locate an osagent. All other communication with the osagent makes use of a point-to-point communication. "Using Point-to-point Communications" on page 5-6 describes how to override the use of broadcast messages.*

Agent-to-Agent Cooperation

When multiple instances of the osagent are started on different hosts, each osagent will recognize a subset of the objects available and communicate with other osagents to locate objects it cannot find. If one of the osagent processes should terminate unexpectedly, all implementations registered with that agent will be notified and they will automatically re-register with another available osagent.

Cooperation with the OAD

The osagent maintains a list of all persistent object implementations that are registered with the Object Activation Daemon. When the ORB requests the osagent to locate an object that has been registered directly with the osagent, the address of the object is returned. If the requested object is registered with the OAD, the osagent will return the address of the OAD capable of activating the object and the ORB will contact that OAD.

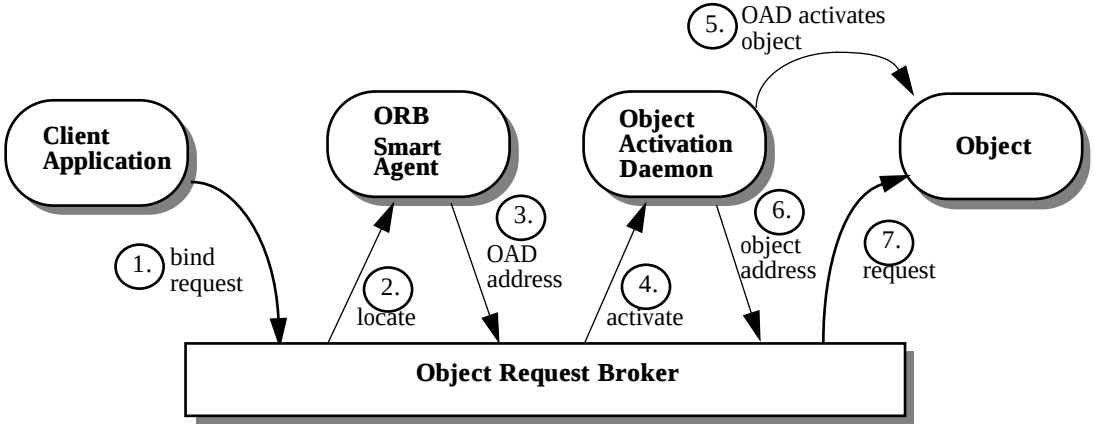


Figure 5-1 The sequence of locating and activating an object registered with the OAD.

Starting the osagent

At least one instance of the osagent should be running on your local network. Figure 5-2 shows how to start the osagent. The verbose mode option provides informational and diagnostic messages. Local network refers to the part of the network within which broadcast message can be sent—machines in the same subnet.

This flag indicates that information and diagnostic messages are to be displayed.

```
osagent -v
```

Figure 5-2 Starting the osagent.

Agent Fault Tolerance

If you run more than one instance of the osagent on a local network and one of those agents becomes unavailable, all object implementations registered with that agent will be automatically re-registered with another agent. Likewise, client applications using an osagent that becomes unavailable will be automatically switched to another agent by VisiBroker. No special coding techniques are required to take advantage of this osagent fault-tolerance, as long as more than one osagent exists on your local network.

ORB DOMAINS

It is often desirable to have two or more separate ORB domains running at the same time. One domain might consist of the production versions of client applications and object implementations while another domain might be made up of test versions of the same clients and objects that have not yet been released for general use.

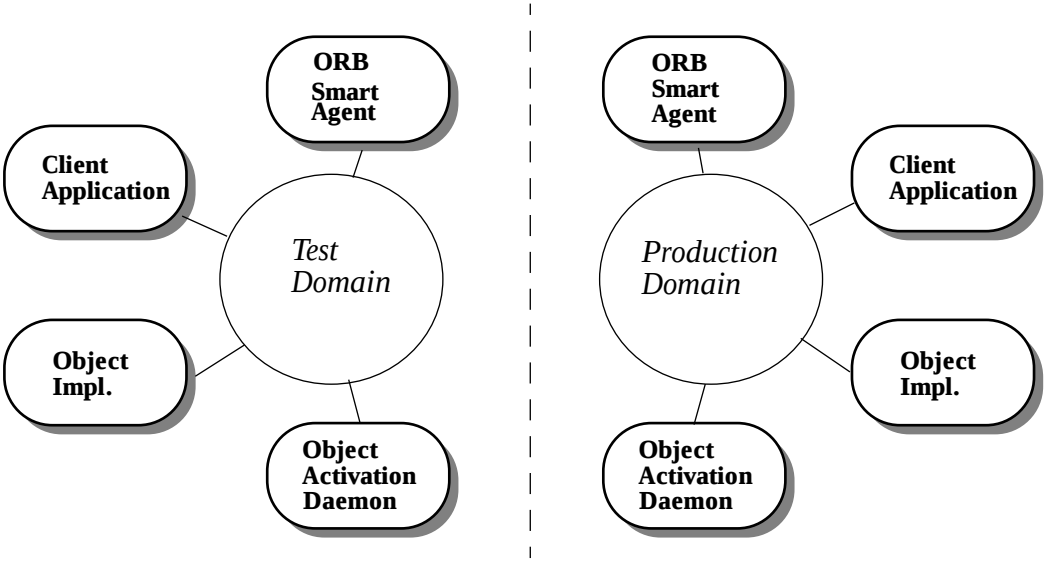


Figure 5-3 Separate ORB Domains.

VisiBroker allows you to distinguish between two or more ORB domains on the same network by using a unique UDP port number for the osagents for each domain. The environment variable OSAGENT_PORT must be set on each host running an osagent, an oad, object implementations, or client applications assigned to that ORB domain.

Check with your system administrator to determine what port numbers are available for your use.

```
prompt> setenv OSAGENT_PORT 5678
prompt> osagent &
prompt> oad &
```

Figure 5-4 Setting the OSAGENT_PORT environment variable for a UNIX system running csh.

CONNECTING AGENTS ON DIFFERENT LOCAL NETWORKS

If you start multiple osagents on your local network, they will discover each other by using UDP broadcast messages. Your network administrator configures a local network by specifying the scope of broadcast messages using the IP subnet mask. Figure 5-5 shows two local networks, connected by a network link.

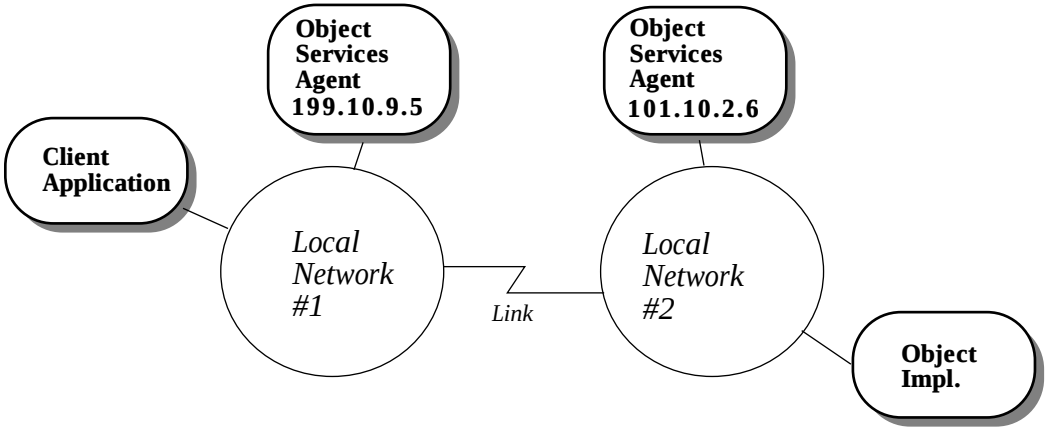


Figure 5-5 Two osagent processes and their IP addresses, located on separate, connected local networks.

To allow the osagent on one network to contact an osagent on another local network, you must make the IP address of the remote osagent available in a file named **agentaddr**. Figure 5-6 shows what this file would contain to allow the osagent on local network #1 to connect to the osagent on the other network. The path to this file is specified by the **ORBELINE** environment variable that is set for the osagent process.

```
101.10.2.6
```

Figure 5-6 Content of the agentaddr file for the osagent on network #1.

With the appropriate **agentaddr** file, the client application on network #1 could locate and use object implementations on network #2. For more information on environment variables, see the VisiBroker for **C++** Reference Guide.

NOTE *Even if a remote network has multiple osagents running, you need to list all of the osagents for that network in the agentaddr file.*

USING POINT-TO-POINT COMMUNICATIONS

VisiBroker provides you with three different strategies for circumventing the use of UDP broadcast messages for locating osagent processes. When an osagent is located with any of these alternate approaches, that agent will be used for all subsequent interactions. If an osagent cannot be located using any of these alternate approaches, the ORB will revert to using the broadcast message scheme to locate an osagent.

Specifying IP addresses with the agentaddr File

You can use the **agentaddr** file to circumvent the use of UDP broadcast message to locate an osagent. Simply create an **agentaddr** file containing the IP addresses of each node where an osagent is running and then set the **OSAGENT_ADDR** environment variable to point to the location of the **agentaddr** file. When a client application or object implementation has this environment variable set, the ORB will try each address in the file until an osagent is located.

Specifying IP addresses as Run-time Parameters

Figure 5-7 shows how you can specify an osagent's IP address as a run-time parameter for your client application or object implementation. You can have a machine with many IP addresses. If you do have many IP addresses on a machine, specify a particular address on the command line for any program using the orb. See "Advanced Networking Options" on page 5-9 for more information on this and other run-time parameters.

```
prompt> server -ORBagentaddr 101.10.2.6 &
...
     or

prompt> client -ORBagentaddr 199.10.9.5
```

Figure 5-7 Specifying an osagent's IP address as a run-time parameter.

OBJECT IMPLEMENTATION FAULT TOLERANCE

You can provide object implementation fault tolerance for stateless objects by simply starting instances of those objects on multiple hosts. The osagent will detect the loss of the connection between the client application and the object implementation and the ORB will automatically attempt to establish a connection with another instance of the object implementation. The client can continue invoking methods on the object without being concerned that a new instance of the object is being used.

The rebind option, discussed in "Enabling Re-Binds" on page 3-9, must be enabled if the ORB is to be able re-connect the client with a replica object implementation.

Object Implementations that Maintain State

Fault tolerance can still be achieved with object implementations that maintain state, but it will not be transparent to the client application. In these cases, the client application must register an event handler for the ORB object. When the connection to an object implementation fails and the ORB re-connects the client to a replica object implementation, the event handler's `rebind_succeeded` method will be invoked by the ORB. The client can implement this method to bring the state of the replica up to date. Event handlers are described in Chapter 7.

Replicating Instantiated Objects

If the ORB objects that you wish to be fault tolerant are created by a server process instantiating the implementation's **C++** class, you need only ensure that the server process is started on multiple hosts.

Replicating Objects Registered with the OAD

If the ORB objects that you wish to be fault tolerant are registered with the oad, you must ensure that the oad is started on multiple hosts. Furthermore, you must ensure that the ORB objects are registered with each of the oad processes.

NOTE *The type of object replication provided by VisiBroker does not provide a multi-cast or mirroring facility. At any given time there is always a one-to-one correspondence between a client application and a particular object implementation.*

OBJECT MIGRATION

Object migration is the process of terminating an object implementation on one host and then starting it on another host. Object migration can be used to move objects from overloaded hosts to hosts that have more resources or processing power. Object migration can also be used to keep objects available when a host has to be shutdown for hardware or software maintenance.

The rebind option, discussed in "Enabling Re-Binds" on page 3-9, must be enabled for the ORB to be able re-connect a client with a object implementation that has migrated to a new host.

The migration of objects that do not maintain state is transparent to the client application. If a client is connected to an object implementation that has migrated, the osagent will detect the loss of the connection and transparently re-connect the client to the new object on the new host.

Migrating Object that Maintain State

The migration of objects that maintain state is also possible, but it will not be transparent to a client application that has connected before the migration process begins. In these cases, the client application must register an event handler for the ORB object. When the connection to the original object is lost and the ORB re-connects the client to the object, the event handler's `rebind_succeeded` method will be invoked by the ORB. The client can implement this method to bring the state of the object up to date. Event handlers are described in Chapter 7.

Migrating Instantiated Objects

If the ORB objects that you wish to migrate were created by a server process instantiating the implementation's **C++** class, you need only terminate the server process and start it on a new host. When the original instance is terminated, it will be un-registered with the osagent. When the new instance is started on the new host, it will register with the osagent. From that point on, client invocations will be routed to the object implementation on the new host.

Migrating Objects Registered with the OAD

If the ORB objects that you wish to migrate are registered with the oad, you will need to ensure they are un-registered with the oad. Furthermore, you must ensure that the ORB objects are then registered with the oad on the new host. Here are the steps:

1 Un-register the object implementation from the OAD on the old host.

2 Terminate the object implementation on the old host.

3 Register the object implementation with the OAD on the new host.

See Chapter 4 for detailed information on registering and unregistering object implementations.

ADVANCED NETWORKING OPTIONS

Although VisiBroker provides reasonable default settings for the network resources it uses, you may fine-tune these setting through parameters passed to the `ORB_init` method. The object implementation can set similar options through parameters passed to the `BOA_init` method.

ORB_init Options

Figure 5-8 shows the definition of the ORB_init method and the arguments it accepts. The argc and argv parameters are passed in exactly the same format as arguments passed to your client's main routine. The argc parameter defines the number of arguments and argv is an array of char pointers to those arguments. ORB settings take the form of type-value pairs, enabling them to be distinguished from other arguments passed to your client application. In fact, the ORB_init method will ignore any arguments it does not recognize. Table 5-9 summarizes the `ORB_init` options.

```
class CORBA {
     ...
     static ORB_ptr ORB_init(int& argc, char *const *argv,
                        *orb_id = (char *)NULL);
     ...
};
```

Figure 5-8 The `ORB_init` *method definition.*

In the preceding example, orb_id identifies the type of ORB. Currently "Internet ORB" is the only supported value.

TYPE/VALUE PAIR	PURPOSE
-ORBagentaddr ip_address	Specifies the IP address of the host running the osagent this client should use. If an osagent is not found at the specified address or if this option is not specified, broadcast messages will be used to locate an osagent. You can have a machine with many IP addresses. If you do have many IP addresses on a machine, specify a particular address on the command line for any program using the orb.
-ORBagentport port_number	Specifies the port number of the osagent. This option can be used if multiple ORB domains are in use, described in “ORB Domains” on page 5-4.
-ORBsendbufsize buffer_size	Specifies the size of the buffer used to send client requests. If not specified, an appropriate buffer size will be used.
-ORBrcvbufsize buffer_size	Specifies the size of the buffer used to receive responses. If not specified, an appropriate buffer size will be used.
-ORBmbufsize buffer_size	Specifies the size of the intermediate buffer used by VisiBroker. An argument will be copied to the intermediate buffer if it is not too big, otherwise VisiBroker will maintain a pointer to the argument instead of copying it. Changing this parameter can seriously affect the performance of your system.
-ORBshmsize size	Specifies the size of the send and receive segments in shared memory. If your client applications and object implementations communicate via shared memory, you may use this option to enhance performance.

Table 5-9 ORB_init options

BOA_init Options

Figure 5-10 shows the definition of the BOA_init method and the arguments it accepts. Like the ORB_init method, the argc and argv parameters passed to BOA_init are in exactly the same format as arguments passed to your object implementation’s main routine. All but two of the BOA settings take the form of type-value pairs. The BOA_init method will ignore any arguments it does not recognize.

Table 5-11 summarizes the BOA_init options. boa_identifier identifies the type of object-adaptor to be used.

```
class CORBA {
     ...
     static BOA_ptr BOA_init(int& argc, char *const *argv,
                         const char *boa_identifier = “PMC_BOA”);
     ...
};
```

Figure 5-10 The BOA_init *method definition.*

TYPE/VALUE PAIR	PURPOSE
-OAipaddr ip_address	Specifies the IP address to be used for the Object Adaptor. Use this option if your machine has multiple network interfaces and the BOA is associated with just one address. If no option is specified, the host’s default address is used.
-OAport port_number	Specifies the port number to be used by this process. If none is specified, an unused port will be selected.
-OAsendbufsize buffer_size	Specifies the size of the buffer used to send messages. If not specified, an appropriate value will be used.
-OArcvbufsize buffer_size	Specifies the size of the buffer used to receive messages. If not specified, an appropriate value will be used.
-OAnoshm	Disables the use of shared memory as a message transport.
-OAshm	Enables the use of shared memory as a message transport. This option is the default.

Table 5-11 BOA_init options.

CHAPTER 6

ERROR HANDLING

This chapter describes how errors are reflected and handled in the CORBA model. User exceptions and system exceptions are discussed. If your platform does not support the C++ `try` and `catch` statements, an alternative error mechanism is discussed. This chapter includes the following major sections:

EXCEPTIONS IN THE CORBA MODEL

The CORBA specification defines a set of **system** exceptions that can be raised when errors occur in the processing of a client request. You can define **user** exceptions in the IDL interface for an objects you create and specify the circumstances under which those exceptions are to be raised. If an object raises an exception while handling a client request, the ORB is responsible for reflecting this information back to the client.

The Exception Class

VisiBroker uses **C++** classes to represent both **system** and **user** exceptions. Since both types of exceptions require similar functionality, `SystemException` and `UserException` classes are derived from a common `Exception` class. When an exception is raised, your application can **narrow**, or **cast down**, from the `Exception` class to a specific `UserException` or `SystemException`. Figure 6-1 shows portions of the `Exception` class definition.

```
class Exception
{
     ...
     public:
     Exception(const Exception &);
     ~Exception();
     Exception &operator=(const Exception &);
     ...

     friend ostream& operator<<(ostream& strm,
                        const Exception& exc);
     const char *_name() const;
     const char*_repository_id() const;
};
```

Figure 6-1 Portions of the `Exception` *class definition.*

METHODS PROVIDED BY THE EXCEPTION CLASS

All exceptions have a name and a repository ID, though the name of the exception name is sufficient for error reporting. The repository ID includes the name as well as additional information about the exception. You can invoke the `_name` and `_repository_id` methods on an exception to obtain this information.

Assume you have a client application that requests a bind for an object whose server is currently not running, causing an exception to be raised. If your application called the `_name` method on the exception object it would return a string containing “CORBA::NO_IMPLEMENT”. If your application called the `_repository_id` method, it would return a string containing “IDL:obg.omg/CORBA/NO_IMPLEMENT:1.0”.

SYSTEM EXCEPTIONS

System exceptions are usually raised by the ORB, though it is possible for object implementations to raise them using the Implementation Event Handler discussed in Chapter 7. When the ORB raises a `SystemException`, it will be one of the CORBA-defined error conditions shown in Figure 6-4 .

```
class SystemException: public Exception
{
     public:
          static const char   *_id;
          virtual             ~SystemException();
          ULong               minor() const;
          void                minor(ULong val);
          CompletionStatus    completed() const;
          void                completed(CompletionStatus status);
          ...
          static SystemException *_narrow(Exception *exc);

     private:
          ULong               _minor;
          completion_status   _status;
          ...
};
```

Figure 6-2 The `SystemException` *class.*

Completion Status

System exceptions have a **completion status** that tells you whether or not the operation that raised the exception was completed. The `CompletionStatus` enumerated values are shown below. `COMPLETED_MAYBE` is retuned when the status of the operation cannot be determined.

```
enum CompletionStatus {
          COMPLETED_YES = 0;
          COMPLETED_NO = 1;
          COMPLETED_MAYBE = 2;
};
```

Figure 6-3 The `CompletionStatus` *values.*

You can retrieve and set the completion status using these `SystemException` methods.

```
CompletionStatus       completed();
void                   completed(CompletionStatus status);
```

Getting and Setting the Minor Code

You can retrieve and set the minor code using these `SystemException` methods. Minor codes are used to provide better information about the type of error.

```
ULong     minor() const;
void      minor(ULong val);
```

Casting to a SystemException

The design of the VisiBroker exception classes allows your application to catch any type of exception and then determine its type by using the `_narrow` method. A static method, `_narrow` accepts a pointer to any `Exception` object. If the pointer is of type `SystemException`, `_narrow` will return the pointer to you. If the pointer is not of type `SystemException`, `_narrow` will return a `NULL` pointer.

EXCEPTION NAME	DESCRIPTION
UNKNOWN	Unknown exception.
BAD_PARAM	An invalid parameter was passed.
NO_MEMORY	Dynamic memory allocation failure.
IMP_LIMIT	Implementation limit violated.
COMM_FAILURE	Communication failure.
INV_OBJREF	Invalid object reference specified.
NO_PERMISSION	No permission for attempted operation.
INTERNAL	ORB internal error.
MARSHAL	Error marshalling parameter or result.
INITIALIZE	ORB initialization failure.
NO_IMPLEMENT	Operation implementation not available.
BAD_TYPECODE	Invalid typecode.
BAD_OPERATION	Invalid operation.
NO_RESOURCES	Insufficient resources to process request.
NO_RESPONSE	Response to request not yet available.
PERSIST_STORE	Persistent storage failure.
BAD_INV_ORDER	Routine invocations out of order.
TRANSIENT	Transient failure.
FREE_MEM	Unable to free memory.
INV_INDENT	Invalid identifier syntax.
INV_FLAG	Invalid flag was specified.
INTF_REPOS	Error accessing interface repository.
BAD_CONTEXT	Error processing context object.
OBJ_ADAPTOR	Failure detected by object adaptor.
DATA_CONVERSION	Data conversion error.
OBJECT_NOT_EXIST	Object is not available.

Figure 6-4 CORBA-defined system exceptions.

Handling System Exceptions

Your applications should always check for system exceptions after making ORB-related calls. Figure 6-5 illustrates how you might enhance the library client application, discussed in Chapter 2, to print an exception using the << operator.

NOTE *If the C++ compiler for your platform does not support exceptions, see page A–2 for a discussion on using CORBA-defined Environments for handling exceptions.*

```
....
library *library_object;
try {
     library_object = library::_bind();
}
// Check for errors
catch(const CORBA::Exception& excep) {
     cout << "Error binding to library:" << endl;
     cout << excep; << endl;
     return(0);
}
...
```

Figure 6-5 Printing an exception.

If you were to execute the client application with these modifications without a server present, the output shown in Figure 6-6 would explain that the operation did not complete and the reason for the exception.

```
Error binding to library:
Exception: CORBA::NO_IMPLEMENT
     Minor: 0
     Completion Status: NO
```

Figure 6-6 Output from modified library client application.

Narrowing to a System Exception

You can modify the library client application to attempt to narrow any exception that is caught to a `SystemException`. Figure 6-7 shows how you might modify the client application. Figure 6-8 shows how the output would appear if a system exception occurred.

```
....
library_var *library_object;
try {
library_object = library::_bind();
}
// Check for errors
catch(const CORBA::Exception& excep) {
CORBA::SystemException*sys_excep;
sys_excep = CORBA::SystemException::_narrow(&excep);
if(sys_excep != NULL) {
     cout << “System Exception occurred:” << endl;
     cout << “    exception name: “ <<
     sys_excep->name() << endl;
     cout << “    minor code: “ <<
     sys_excep->minor() << endl;
     cout << “    completion code: “ <<
     sys_excep->completed() << endl;
} else {
     cout << “Not a system exception” << endl;
}
return(0);
}
...
```

Figure 6-7 Narrowing an exception to a system exception.

```
System Exception occurred:
     exception name: CORBA::NO_IMPLEMENT
     minor code: 0
     completion code: 1
```

Figure 6-8 Output from the system exception.

Catching System Exceptions

Rather than catching all types of exceptions, you may choose to specifically catch each type of exception that you expect. Figure 6-9 shows this technique.

```
....
library_var *library_object;
try {
library_object = library::_bind();
}
// Check for errors
catch(const CORBA::SystemException& excep) {
cout << "System Exception occurred:" << endl;
cout << "exception name: " <<
     sys_excep->name() << endl;
cout << "minor code: " <<
     sys_excep->minor() << endl;
cout << "completion code: " <<
     sys_excep->completed() << endl;
}
// Try catching other types of exceptions.
...
```

Figure 6-9 Catching specific types of exceptions.

USER EXCEPTIONS

Exceptions that can be raised by an object are called ***user exceptions***. When you define your object's interface in IDL you can specify the user exceptions that the object may raise. Figure 6-10 shows the `UserException` class that the IDL compiler will use to derive the user exceptions you specify for your object.

```
class UserException: public Exception
{
     public:
          ...
          static const char         *_id;
          virtual                   ~UserException();
          static UserException      *_narrow(Exception *exc);
};
```

Figure 6-10 The `UserException` *class.*

Defining User Exceptions

Assume that you want to enhance the library application, introduced in Chapter 2, so that the library object will raise an exception. If the library object's book list is full and an attempt is made to add a book,

you want a user exception named `CapacityExceeded` to be raised. The additions to the IDL specification for the library interface are shown in bold letters.

```
// IDL specification for book and library objects
struct book {
     string author;
     string title;
};

interface library {
     exception CapacityExceeded {
     };
     boolean add_book( in book book_info)
                                     raises(CapacityExceeded);
};
```

Figure 6-11 Defining User Exceptions

The IDL compiler will generate this **C++** code for a CapacityExceeded exception class.

```
class library: public virtual CORBA::Object
{
     ...
     class CapacityExceeded: public CORBA::UserException
     {
          public:
               CapacityExceeded();
               ~CapacityExceeded();
               static CapacityExceeded *_narrow(CORBA::Exception *exc);
               ...
     };

     ...
};
```

Figure 6-12 The `CapacityExceeded` *class generated by the IDL compiler.*

On platforms that support **C++** exceptions, the `library` and `_sk_library` classes generated by the IDL compiler from this specification will incorporate the `throw` directive into the `add_book` methods signature.

```
virtual CORBA::Boolean       add_book(const book& book_info)
          throw (library::CapacityExceeded);
```

Figure 6-13 The new `add_book` *method signature.*

Modifying the Object implementation

The `Library` object must be modified to use the exception by changing the `add_book` function prototype and throwing the exception under the appropriate error conditions.

```
CORBA::Boolean Library::add_book(const book& book_info)
                        throw (library::CapacityExceeded)
{
     CORBA::Boolean ret;
     if( (ret = bk_list.add_to_list(book_info)) == 0 )
          throw library::CapacityExceeded();
     return ret;
}
```

Figure 6-14 Modifying the object implementation to throw an exception.

Catching User Exceptions

When an object implementation raises an exception, the ORB is responsible for reflecting the exception to your client application. Checking for a UserException is similar to checking for a SystemException. To modify the library client application to catch the CapacityExceeded exception, you would make modifications like those shown below.

```
...
try {
          ret = library_object->add_book(book_entry);
}
// Check for System Exceptions
catch(const library::CapacityExceeded& excep) {
          cout << “CapacityExceeded returned:” << endl;
          cout << excep; << endl;
          // Do any necessary clean-up
          return(0);
}
...
```

Figure 6-15 Catching a `UserException`.

Adding Fields to User Exceptions

You can associate values with user exceptions. Figure 6-16 shows how to modify the IDL interface specification to add a size value to the CapacityExceeded user exception.The object implementation that

raises the exception is responsible for setting the value. The new value is printed automatically when the exception is put on the output stream.

```
// IDL specification for book and library objects
struct book {
     string author;
     string title;
};

interface library {
     exception CapacityExceeded {
          long size;
     };
     boolean add_book( in book book_info)
                                    raises(CapacityExceeded);
};
```

Figure 6-16 Adding a value to the `CapacityExceeded` *exception.*

Portability considerations

You may want to consider always using these macros in your applications since they automatically adapt to the capabilities of your **C++** compiler. Applications that use these macros can be more easily ported to all supported platforms. There are two sets of compatibility macros; one set for compilers with exception support and one set for compilers without exception support. The defined constant `_PMC_NOEXCEPTIONS` determines which macro set will be used. If _PMC_NOEXCEPTIONS is not defined, then the compatibility macros will be mapped as shown in the following table.

MACRO NAME	MACRO EXPANSION
PMCTRY	try
PMCTHROW(type_name)	throw(type)
PMCTHROW_LAST	throw;
PMCCATCH(type_name, variable_name)	catch(const type &var)
PMCAND_CATCH	catch(const type &var)
PMCEND_CATCH	none
PMCTHROW_SPEC(x)	none or throw(x)

Figure 6-17 Compatibility macro mapping for compilers that support exceptions.

Understanding the Environment Class

The Environment class enables exceptions to be registered in your application's environment. Methods are provided that allow the PMC macros to determine if a system or user exception has occurred and obtain the details of the exception. If you use the PMC macros shown in Figure 6-17 , you should not have to explicitly call these methods yourself.

VisiBroker creates a default Environment object for each process. If your platform supports threads, an Environment object is created for each thread.

The include file **env.h** contains the Environment class' definition.

```
class Environment
{
     private:
          Exception     *_exception;
     public:
          Environment();
          ~Environment();
          ...
          Exception     *exception() const;
          void          exception(Exception *exp);
          void          clear();
          ...
};
```

Figure 6-18 The Environment *class.*

Environment Methods

The PMC macros make use of the Environment class internally. If you do not want to use the PMC macros and do not have exception support, you can use the Environment class.

The exception method is used by PMCTHROW to raise an exception.

```
void      exception(Exception *exp);
```

This method is used by PMCCATCH to return the exception that has been set for the environment. If no exception has been set, a NULL pointer is returned.

```
Exception              exception(Exception *exp);
```

The clear method clears any exception that has been raised in the environment. This method is invoked after the exception has been retrieved.

```
void                    clear();
```

The `is_nil` method determines if the supplied pointer is NULL. If the pointer is NULL, a value other than zero is returned. If the pointer is not NULL, zero is returned. The behavior of the `is_nil` method is defined in the CORBA specification.

```
static Boolean                      is_nil(Environment_ptr env);
```

You can use the following `CORBA` class static method to obtain a pointer to the `Environment` object for the current process or current thread, if threads are supported.

```
class CORBA {
     ...
     static Environment& current_environment();
     ...
}
```

CHAPTER 7

HANDLING EVENTS

VisiBroker transparently manages networking and communication issues for you, but there may be times when your client applications and object implementations need to define their own error and recovery processing. VisiBroker provides a set of event handling mechanisms for notifying clients and object implementations of system events. Event handing can be used to implement accounting, tracing, debugging, logging, security or encryption features. This chapter includes the following major sections:

EVENT HANDLER CONCEPTS

Event handlers objects allow client application and object implementations to define methods that the ORB will invoke to handle events such as the success or failure of a bind request or the failure of an object implementation. Two different event handler classes are provided because the types of events that can be handled are different for clients and object implementations. However, the procedure for using an event handler is similar for clients and object implementations.

1 Derive an event handler class for your object, defining the event methods you wish to handle.

2 Provide Implementations for the event methods you wish to handle.

3 Add code to the client or object implementation to register the event handler.

CLIENT-SIDE EVENTS	IMPLEMENTATION-SIDE EVENTS
BIND SUCCEEDED	BIND REQUEST RECEIVED
BIND FAILED	UNBIND REQUEST RECEIVED
SERVER ABORTED	CLIENT ABORTED
REBIND SUCCEEDED	PRE-METHOD
REBIND FAILED	POST-METHOD

Figure 7-1 A summary of the events that can be handled by client applications and object implementations.

CLIENT EVENT HANDLERS

Client applications can register an event handler with the ORB to handle events for a particular ORB object. The client can also globally register an event handler to handle events for all ORB objects the client uses.

Figure 7-2 shows a portion of the **pmcext.h** include file that contains the class definition for the `ClientEventHandler` class.

```
class PMC_EXT
{
     struct ConnectionInfo {
          CORBA::String_var   hostname;
          CORBA::UShort       port;
          CORBA::Long         fd;
          ...
     };

     class ClientEventHandler
     {
          public:
               virtual void bind_succeeded(CORBA::Object_ptr,
                        const ConnectionInfo&);
               virtual void bind_failed(CORBA::Object_ptr);
               virtual void server_aborted(CORBA::Object_ptr);
               virtual void rebind_succeeded(CORBA::Object_ptr,
                        const ConnectionInfo&);
               virtual void rebind_failed(CORBA::Object_ptr);
          ...
     };
     ...
};
```

Figure 7-2 The `ConnectionInfo` *structure and the* `ClientEventHandler` *class.*

The ConnectionInfo Structure

This structure represents all the information needed for a connection. It includes the host name where the object implementation resides, the port number and the file descriptor used for the connection. This structure is modified when the connection to the object implementation is lost and a re-bind operation is attempted.

ClientEventHandler Methods

When an event handler object has been registered for a particular ORB object, the ORB will call the ClientEventHandler methods when a specific event occurs. If the event handler is registered as a global event handler, the ClientEventHandler methods will be called for any event related to any object the client uses.

The `bind_succeeded` method is called by the ORB when the client's request to bind to the ORB object has completed successfully. A pointer to the object that has been bound is provided as a parameter as well as the connection information.

The `bind_failed` method is called if the client's bind request fails. A pointer to the object to which the event is related is provided as a parameter.

The `server_aborted` method is called if the connection to the object implementation is lost. A pointer to the object to which the event is related is provided as a parameter.

The `rebind_succeeded` method is called when an attempt to re-connect to an object implementation succeeds. A pointer to the object that has been re-bound is provided as a parameter as well as the new connection information.

The `rebind_failed` method is called when an attempt to re-connect to an object implementation fails. A pointer to the object to which the event is related is provided as a parameter.

Creating a Client Event Handler

To implement an event handler for your client application, you must derive your own event handler class for the class you wish to monitor. You will need to implement only those event handler methods you wish to override. If you do not override an event handler method, no special processing will occur and no performance overhead will be added to the application.

Figure 7-3 shows how you would define an event handler for the library client, introduced in Chapter 2. Only three of the possible five methods offered by ClientEventHandler have been overridden and Figure 7-4 shows simple implementations for these methods.

```
class LibraryClientHandler : public PMC_EXT::ClientEventHandler
{
     ...
     public:
          void bind_succeeded(CORBA::Object_ptr, const ConnectionInfo&);
          void bind_failed(CORBA::Object_ptr);
          void server_aborted(CORBA::Object_ptr);
};
```

Figure 7-3 An example event handler for the library client.

```
void LibraryClientHandler::bind_succeeded(CORBA::Object_ptr obj,
          const ConnectionInfo&)
{
          cout << “Event Handler bind_succeeded for: “
                    << obj->_interface_name() << endl;
}

void LibraryClientHandler::bind_failed(CORBA::Object_ptr obj)
{
          cout << “Event Handler bind_failed for: “
                    << obj->_interface_name() << endl;
}

void LibraryClientHandler::server_aborted(CORBA::Object_ptr obj)
{
          cout << “Event Handler server_aborted for: “
                    << obj->_interface_name() << endl;
}
```

Figure 7-4 Implementation for the `LibraryClientHandler` *method.*

The Handler Registry

You can use the static method `HandlerRegistry::instance` to obtain a pointer to the registry and then invoke the methods for registering and un-registering various types of event handlers.

```
class HandlerRegistry{
     ...
     public:
     ...
          static HandlerRegistry_ptr instance();
          void reg_obj_client_handler(CORBA::Object_ptr obj,
                    ClientEventHandler_ptr handler);
          void reg_glob_client_handler(ClientEventHandler_ptr handler);
          void unreg_obj_client_handler(CORBA::Object_ptr obj);
          void unreg_glob_client_handler();
          void reg_obj_impl_handler(CORBA::Object_ptr obj,
                    ImplEventHandler_ptr handler);
          void reg_glob_impl_handler(ImplEventHandler_ptr handler);
          void unreg_obj_impl_handler(CORBA::Object_ptr obj);
          void unreg_glob_impl_handler();
     ...
};
```

Figure 7-5 The `HandlerRegistry` *class.*

HandlerRegistry Methods for Clients Applications

The `reg_obj_client_handler` method can be called by your client application to register an event handler for a specific object. The parameters passed to this method are a reference to the object and a pointer to the object's `ClientEventHandler`. If the object reference is not valid, an `InvalidObject` exception will be raised. If an event handler has already been registered for the specified object, a `HandlerExists` exception will be raised. You can use the `unreg_obj_client_handler` method to un-register a previously registered event handler.

The `reg_glob_client_handler` method can be called by your client application to register an event handler for all object the client uses. The parameter passed to this method is a pointer to the object's `ClientEventHandler`. If a global handler has already been registered, a `HandlerExists` exception will be raised. You can use the `unreg_glob_client_handler` method to un-register a previously registered global event handler.

NOTE *If both an object event handler and a global event handler are registered, the object event handler will take precedence for events that occur which are related to its object. All other events will be handled by the global event handler.*

The `unreg_obj_client_handler` method can called by your client application to un-register an event handler for a specific object. A reference to the object whose event handler is to be removed is passed as a parameter. If the object reference is not valid, an `InvalidObject` exception will be raised. If no event handler has been registered for the specified object, a `NoHandler` exception will be raised.

The `unreg_glob_client_handler` method can called by your client application to unregister a global event handler. If no global event handler has been registered, a `NoHandler` exception will be raised.

Registering Client Event Handlers

There are methods for registering both global and per-object event handlers. In either case, the client uses the `PCM_EXT::HandlerRegistry::instance` method to obtain a pointer to the ORB's event handler registry. Figure 7-6 shows the registration process for a global event handler and shows how to register a per-object event handler. Both examples assume that the LibraryClientHandler class has been defined and implemented, as shown in Figure 7-3 and Figure 7-4 .

```
#include <fstream.h>
#include <lib_client.hh>
main(int argc, char *const *argv)
{
     CORBA::Boolean    ret;
     // Initialize the ORB
     CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

     // Parse arguments
     ...

     // Declare the library object
     library_var *library_object;

     // Declare the library event handler
     LibraryClientHandler client_handler;

     // Obtain a handle to the ORB's registry
     PCM_EXT::HandlerRegistery_ptr  registry_handle =
          PCM_EXT::HandlerRegistry::instance();
     try {
          // Register the global event handler
          registry_handle->reg_glob_client_handler(&client_handler);
     }
     catch(const PMC_EXT::HandlerExists& excep) {
          cout << "A global handler was already registered" << endl;
     }

     // Bind to the library object and invoke methods
     ...

     try {
          // Un-register the global event handler
          registry_handle->unreg_glob_client_handler();
     }
     catch(const PMC_EXT::NoHandler& excep) {
          cout << "No global handler was registered" << endl;
     }

     return(1);
}
```

Figure 7-6 Registering a global client event handler.

```
#include <fstream.h>
#include <lib_client.hh>
main(int argc, char *const *argv)
{
     CORBA::Boolean    ret;
     // Initialize the ORB
     CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
     // Parse arguments
     ...

     // Declare the library object
     library_var *library_object;
     // Bind to the library object
     ...

     // Declare the library event handler
     LibraryClientHandler client_handler;

     // Obtain a handle to the ORB's registry
     PCM_EXT::HandlerRegistery_ptr  registry_handle =
          PCM_EXT::HandlerRegistry:::instance();
     try {
          // Register the library event handler
          registry_handle->reg_obj_client_handler(library_object,
                              &client_handler);
     }
     catch(const PMC_EXT::HandlerExists& excep) {
          cout << "A handler was already registered" << endl;
     }

     // Invoke methods on library object
     ...

     try {
          // Un-register the library event handler
          registry_handle->unreg_obj_client_handler(library_object);
     }
     catch(const PMC_EXT::NoHandler& excep) {
          cout << "No handler was registered" << endl;
     }
     catch(const PMC_EXT::InvalidObject& excep) {
          cout << "Invalid object reference" << endl;
     }

     return(1);
}
```

Figure 7-7 Registering a per-object client event handler.

IMPLEMENTATION EVENT HANDLERS

Like client applications, object implementations can register an event handler with the ORB to handle events for a particular ORB object. The implementation can also globally register an event handler to handle events for all ORB objects implemented. Implementation-side event handling can be used for a variety of purposes. For example, the implementation may refuse a client connection request, based on the caller's identity.

The ImplEventHandler Class

Figure 7-8 shows the ImplEventHandler class you will use to derive an implementation event handler. All of the methods shown make use of the ConnectionInfo structure, discussed on page 7–3.

```
class PCM_EXT
{
     ...
     class ImplEventHandler
     {
          public:
               virtual void bind(const ConnectionInfo&,
                       CORBA::Principal_ptr, CORBA::Object_ptr);
               virtual void unbind(const ConnectionInfo&,
                       CORBA::Principal_ptr, CORBA::Object_ptr);
               virtual void client_aborted(const ConnectionInfo&,
                       CORBA::Principal_ptr, CORBA::Object_ptr);
               virtual void pre_method(const ConnectionInfo&,
                       CORBA::Principal_ptr, CORBA::Object_ptr,
                       const char *, CORBA::Object_ptr);
               virtual void post_method(const ConnectionInfo&,
                       CORBA::Principal_ptr, CORBA::Object_ptr,
                       const char *, CORBA::Object_ptr);
     };
     ...
};
```

Figure 7-8 The `ImplEventHandler` *class.*

In the preceding figure, `CORBA::Principal_ptr` allows clients to send information to the server that can retrieve this data and, as the implementor, you can determine if the server allows the action (like a bind) to be performed.

ImplEventHandler Methods

The `bind` method will be called every time a client wishes to connect to this object. This method allow your object implementation to do any special processing before the bind request is processed. Once this method returns, the BOA will proceed with the normal binding process. The parameters passed to this method are the connection information, the Principal value associated with the client and a pointer to the ORB object requested. This method may choose to reject the bind, based on the requestor's identity, by raising a `CORBA::NO_PERMISSION` exception.

The `unbind` method will be called every time a client application calls the CORBA::release method for a previously bound object. The BOA will pass control to this method before the un-bind occurs. The connection information and the object reference are passed to this method.

The `client_aborted` method will be called if the connection to a client application is lost. The connection information and the object reference are passed to this method.

The `pre_method` method will be called every time a client application invokes a method on the object for which the handler is registered. After this method returns, the BOA will proceed with the method invocation. The connection information, Principal of the client, method name and a pointer to the object are all passed to this method.

The `post_method` method will be called after every invocation of a method by a client on the object being traced. After this method returns, the results of the method invocation will be returned to the client. The connection information, `Principal` of the client, method name and a pointer to the object are all passed to this method.

NOTE *If the method invoked by the client raises an exception, post_method will not be called.*

Creating Implementation Event Handlers

Figure 7-9 shows how you can create an implementation event handler for the `Library` object by deriving your own class from the `ImplEventHandler` class. Figure 7-10 shows the implementation for the `LibraryImplHandler` methods defined. You only need to define and provide method implementations for those events you wish to handle.

```
class LibraryImplHandler : public PMC_EXT::ImplEventHandler
{
     public:
          void bind(const ConnectionInfo&, CORBA::Principal_ptr,
                              CORBA::Object_ptr);
          void unbind(const ConnectionInfo&, CORBA::Principal_ptr,
                              CORBA::Object_ptr);
};
```

Figure 7-9 Example event handler class for the `Library` *object implementation.*

```
void LibraryImplHandler::bind(const ConnectionInfo&,
          CORBA::Principal_ptr, CORBA::Object_ptr obj)
{
     cout << “Bind request arrived for “ << obj->_interface_name << endl;
     ...
};

void LibraryImplHandler::unbind(const ConnectionInfo&,
          CORBA::Principal_ptr, CORBA::Object_ptr obj)
{
     cout << “Un-Bind request arrived for “ << obj->_interface_name << endl;
...
};
```

Figure 7-10 Method implementations for the `LibraryImplHandler` *class methods.*

Using the Handler Registry

As with client applications, the HandlerRegistry is used to register implementation event handlers. The HandlerRegistry class is show in Figure 7-5 on page 7–5.

HandlerRegistry Methods for Object Implementations

The `reg_obj_impl_handler` method can be called by your object implementation to register an event handler with the BOA for a specific object. The parameters passed to this method are a reference to the object and a pointer to the object's `ImplEventHandler`. If the object reference is not valid, an `InvalidObject` exception will be raised. If an event handler has already been registered for the specified object, a `HandlerExists` exception will be raised. You can use the `unreg_obj_impl_handler` method to un-register a previously registered event handler.

The `reg_glob_impl_handler` method can be called by your object implementation to register an event handler with the BOA for all objects contained in the implementation. The parameter passed to this method is a pointer to the object's `ImplEventHandler`. If a global handler has already been

registered for this implementation, a `HandlerExists` exception will be raised. You can use the `unreg_glob_impl_handler` method to un-register a previously registered global event handler.

NOTE *If both an object event handler and a global event handler are registered, the object event handler will take precedence for events that occur which are related to its object. All other events will be handled by the global event handler.*

The `unreg_obj_impl_handler` method can be called by your object implementation to un-register an event handler for a specific object. A reference to the object whose event handler is to be removed is passed as a parameter. If the object reference is not valid, an `InvalidObject` exception will be raised. If no event handler has been registered for the specified object, a `NoHandler` exception will be raised.

The `unreg_glob_impl_handler` method can be called by your object implementation to un-register a global event handler. If no global event handler has been registered, a `NoHandler` exception will be raised.

Registering Implementation Event Handlers

Like client applications, your object implementations will use similar procedures for registering both global and per-object event handler. In both cases, the client uses the PCM_EXT::HandlerRegistry::instance method to obtain a pointer to the BOA's event handler registry. Figure 7-11 shows the registration process for a global event handler and Figure 7-12 shows how to register a per-object event handler. Both examples assume that the LibraryImplHandler class has been defined and implemented, as shown in Figure 7-10 .

```
#include <lib_server.hh>
int main(int argc, char **argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

     // Instantiate the Library object
     Library *library_obj = new Library("Harvard");

     // Define the impl_handler and registry instance
     LibraryImplHandler impl_handler;
     PMC_EXT::HandlerRegistry_ptr registry_handle;
     registry_handle = PMC_EXT::HandlerRegistry::instance();
     try {
          // Register a global event handler
          registry_handle->reg_glob_impl_handler(&impl_handler);
     }
     catch(const PMC_EXT::HandlerExists& excep) {
          cout << "Global handler already defined" << endl;
     }

     // Instantiate Library Class, activate object and implementation
     ...
     try {
          registry_handle->unreg_glob_impl_handler();
     }
     catch(const PMC_EXT::NoHandler& excep) {
          cout << "Removal of global event handler failed" << endl;
     }
     ...
}
```

Figure 7-11 Registering a global event handler for an object implementation.

```
#include <lib_server.hh>
int main(int argc, char **argv)
{
     // Initialize ORB and Basic Object Adaptor (BOA)
     CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

     // Instantiate the Library object
     Library *library_obj = new Library("Harvard");

     // Define the impl_handler and registry instance
     LibraryImplHandler impl_handler;
     PMC_EXT::HandlerRegistry_ptr registry_handle;
     registry_handle = PMC_EXT::HandlerRegistry::instance();
     try {
          // Register a global event handler
          registry_handle->reg_obj_impl_handler(library_obj, &impl_handler);
     }
     catch(const PMC_EXT::HandlerExists& excep) {
          cout << "Handler already defined for " <<
                         library_obj->_instance_name << endl;
     }

     // Instantiate Library Class, activate object and implementation
     ...
     try {
          registry_handle->unreg_obj_impl_handler(library_obj);
     }
     catch(const PMC_EXT::NoHandler& excep) {
          cout << "Removal of event handler failed for " <<
                         library_obj->_instance_name << endl;
     }
     ...
}
```

Figure 7-12 Registering a per-object event handler for an object implementation.

CHAPTER 8

ADVANCED PROGRAMMING TOPICS

This chapter discusses developing multi-threaded client applications and object implementations with VisiBroker. This chapter also covers the integration of an object implementation's event loop with other event-driven software. It includes the following major sections:

USING THREADS WITH VISIBROKER

For platforms that support threads, VisiBroker provides two sets of libraries; a single-threaded library and another library that is thread-safe and re-entrant. In addition to providing thread-safe facilities for client applications, the multi-threaded version of the library results in the internal use of threads by the VisiBroker core.

For applications that never intend to use threads, the single-threaded library offers slightly better performance. Both libraries provide identical interfaces, which allows your applications to take advantage of multi-threaded support in the future without worrying about interface changes.

NOTE *The Dispatcher class is useful for single-threaded applications only.*

THREADS IN AN OBJECT IMPLEMENTATION

In the multi-threaded version of VisiBroker, the main thread of an object implementation is responsible for initializing the ORB and BOA. The main thread then waits for connection requests from client applications. Each time a client connection request is received, a new worker thread is spawned within the object implementation to perform all processing for that client. When the client application destroys the connection, the worker thread exits. If several clients access the object implementation at one time, worker threads will be created for each client.

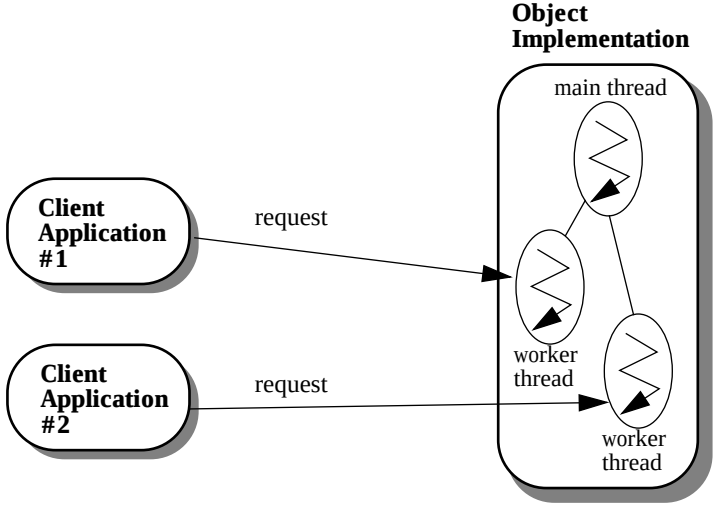


Figure 8-1 A multi-threaded object implementation.

THREADS IN A CLIENT APPLICATION

Client applications can use threads in two ways in relation to an object implementation. The client can use the object reference returned from a single bind in all of the threads it creates. Alternatively, each thread within the client can issue its own bind request.

One Bind with Multiple Client Threads

If your client application issues a single bind and then spawns several threads, it can pass the object reference received from the bind to each thread. A single connection to the object implementation is established and all client worker threads utilize a single worker thread in the object implementation.

NOTE *If a particular client thread issues multiple bind requests to the same object implementation, VisiBroker will not establish multiple connections. Instead, VisiBroker will detect that a connection already exists and will re-use that connection.*

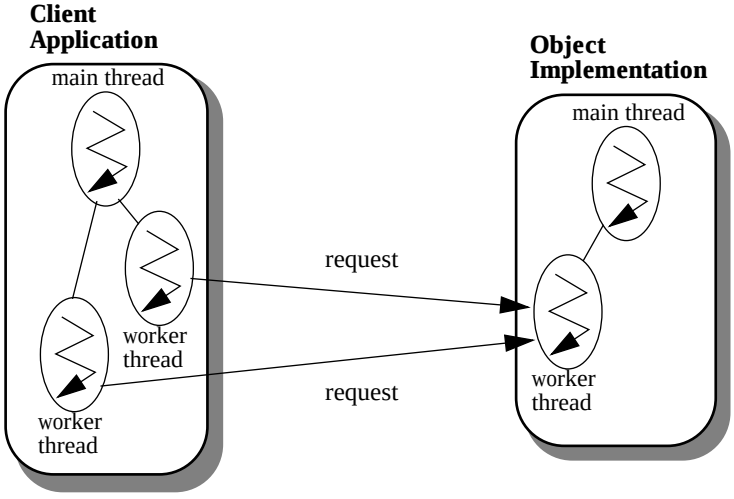


Figure 8-2 One bind for multiple client application threads.

Multiple Binds with Multiple Client Threads

Your client application can spawn several threads and have each thread issue its own bind request to the object implementation. In this case, VisiBroker will establish a separate connection to the object implementation for each client thread and the object implementation will create a worker thread for each client worker thread. This arrangement allows for maximum efficiency because if one client thread issues a blocking request it will not block other client threads that are accessing the same object implementation.

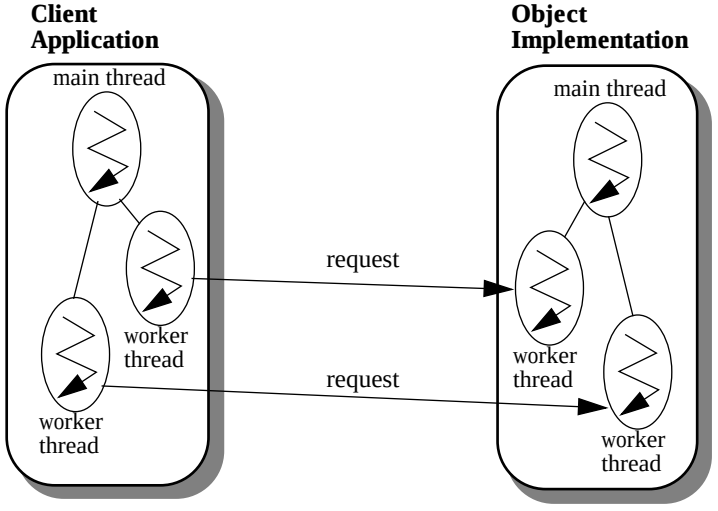


Figure 8-3 Multiple client threads, each making their own bind request.

Multiple Threads with Cloning

Cloning is another technique for achieving a separate object implementation worker thread for each client worker thread. Using this approach, the main thread binds to the object and passes the object reference to each client worker thread it creates. A client worker thread then invokes the `_clone` method on the object reference, resulting in a new connection with the object implementation and the spawning of a new worker thread. You should remember that the _duplicate method increases the reference count to the object and that _clone makes a complete copy of the object reference and results in a new, separate connection.

LINKING MULTI-THREADED APPLICATIONS

All the re-entrant versions of the VisiBroker libraries, regardless of platform, have a "_r" suffix.

U For Unix systems, these libraries should be used:

FILE NAME	DESCRIPTION
liborb_r.so	Re-entrant version of the library liborb.so
liborb_r.a	Re-entrant version of the library liborb.a

W For Windows and Windows/NT, this library should be used:

FILE NAME	DESCRIPTION
ORB_R.DLL	Re-entrant version of the library ORB.DLL

EVENT LOOP INTEGRATION

When your object implementation invokes the BOA::impl_is_ready method, an event loop is entered that waits for the arrival of requests from client applications. Your object implementation may also need to interact with another event-driven system. In a multi-threaded environment, you can solve this problem by simply using two threads; one thread waits for VisiBroker events and the other thread services other events. If your platform does not support threads, you may find it helpful to integrate all event driven processing by using the `Dispatcher` and `IOHandler` classes.

The Dispatcher Class

This class is designed to detect events on several file descriptors and dispatch those events to the appropriate handler. The `Dispatcher` maintains three lists of file descriptors; one list for reading data, one list for writing data and one for exceptions. You can use the `link` method to add a file descriptor to one of the `Dispatcher` class' lists and define the `IOHandler` object to be called to handle events on that file descriptor. You can find the include file for the Dispatcher class in `include/dispatcher/dispatch.h`.

NOTE *The Dispatcher class is useful for single-threaded applications only.*

An application should have only one instance of the `Dispatcher` class and the static `instance` method is provided to create the object, if necessary, and return a pointer to it.

```
class Dispatcher {
     public:
          enum DispatcherMask {
                ReadMask,
                WriteMask,
                ExceptMask
          };

     Dispatcher();
     virtual ~Dispatcher();

     virtual void             link(int fd, DispatcherMask, IOHandler*);
     virtual IOHandler        handler(int fd, DispatcherMask) const;
     virtual void             unlink(int fd);

     virtual void             startTimer(long sec, long usec, IOHandler*);
     virtual void             stopTimer(IOHandler*);

     virtual iv_boolean       dispatch(long& sec, long& usec);
     virtual iv_boolean       dispatch(timeval *);
     virtual iv_boolean       dispatch();

     static Dispatcher&       instance();

     ...
};
```

Figure 8-4 The `Dispatcher` *class.*

ADDING FILE DESCRIPTORS

When using the `link` method to add a file descriptor to the `Dispatcher`, you specify the file descriptor, the `DispatcherMask` and a pointer to an IOHandler `object`. The `DispatcherMask` value determines whether the file descriptor is added to the read, write or exception event list.

When an event occurs on the file descriptor, the `Dispatcher` will invoke the appropriate IOHandler method to service the event. The IOHandler object provides methods for reading data from or writing data to a file descriptor as well as for handling exceptions and expired timers. If an `IOHandler` method returns a negative value indicating it encountered an error, the `Dispatcher` will automatically unlink the `IOHandler` from its file descriptor.

NOTE *You must make multiple invocations of the link method if you want a particular file descriptor to be placed in more than one of the dispatcher's lists. DispatcherMask values cannot be ordered together when calling* `link`.

You can use the handler method to return the IOHandler object defined for a particular file descriptor and DispatcherMask combination.

SETTING TIMERS

You can set an interval timer for a particular `IOHandler` object by invoking the `startTimer` method. This method lets you specify a time interval in a combination of seconds and microseconds. When the interval expires, the `IOHandler` object's `timerExpired` method is invoked.The `Dispatcher` method `stopTimer` can be called to stop a timer.

NOTE *Timers are one shot, not periodic—you have to set them again if you want to time another interval.*

The use of the timer methods can be especially useful in single-threaded environments. Multi-threaded applications have the flexibility of starting timers in separate threads.

DISPATCHING

One form of the `dispatch` method accepts no arguments and blocks indefinitely or until an event occurs on one of its file descriptors. If a file descriptor event occurs, the appropriate `IOHandler` method is invoked before the `dispatch` method returns.

The other two forms of the `dispatch` method accept a time interval specification. If the time interval specified is zero, the `Dispatcher` will return immediately after checking all the file descriptors and timers. If the time interval is greater than zero, the `Dispatcher` will block until an event occurs on one of the file descriptors or until the time interval expires. The `dispatch` method returns a one if an event on a file descriptor caused the return. This method returns a zero if an expired timer caused the `dispatch` method to return.

REMOVING FILE DESCRIPTORS

The `unlink` method removes the specified file descriptor from all lists maintained by the `Dispatcher`.

The IOHandler Class

You derive your own class from `IOHandler` class to handle events on a particular file descriptor. You associate your `IOHandler` object with a file descriptor, using the `Dispatcher` object's `link` method.

You can find the include file for this `IOHandler` in the following location:
include/dispatch/iohandler.h

```
class IOHandler {
     protected:
          IOHandler();
     public:
          virtual      ~IOHandler();
          virtual int  inputReady(int fd);
          virtual int  outputReady(int fd);
          virtual int  exceptionRaised(int fd);
          virtual void timerExpired(long sec, long usec);
};
```

Figure 8-5 The `IOHandler` *class.*

IMPLEMENTING THE IOHANDLER METHODS

You must provide implementations for the IOHandler methods that you want to handle for your file descriptor. Table 8-6 describes each of the methods and Table 8-7 shows the return code conventions that the `Dispatcher` class assumes your methods will follow.

METHOD NAME	DESCRIPTION
inputReady	Called when the Dispatcher detects that data is ready to be read from the file descriptor associated with this handler.
outputReady	Called when the Dispatcher detects that the file descriptor associated with this handler is ready to accept more data.
exceptionRaised	Called when the Dispatcher detects that an I/O exception has occurred on the file descriptor associated with this handler.
timer expired	Called when the Dispatcher is notified that an interval timer for this handler has expired. The current time in seconds and microseconds since January 1, 1970 is passed.

Table 8-6 The IOHandler class methods.

RETURN VALUE	MEANING
-1 or negative value	The method encountered an error and does not want to handle any more events.
0	The method has completed successfully and currently has no more work to do.
1 or a positive value	The method has completed successfully, but has more data to read or write. The dispatcher will keep calling this method, after checking all other file descriptors, until this method returns 0 or a negative value.

Table 8-7 Return code conventions for IOHandler methods.

Using an IOHandler

To create your own `IOHandler`, simply derive your own class and implement those methods you intend to use. Figure 8-8 shows an example IOHandler-derived class.

```
#include <dispatch/iohandle.h>
...

class MyHandler : public IOHandler
{
     public:
          MyHandler();
          virtual ~MyHandler();
          virtual int inputReady(int fd) {
                        // read from file using fd
                        ...
                        if(done) {
                              return(0);
                        } else if(more_left_to_read) {
                              return(1);
                        } else if(failure) {
                              return(-1);
                        }
          }
     ...
};
```

Figure 8-8 An example `IOHandler`*-derived class.*

Figure 8-9 shows how you might instantiate your handler and link it to a file descriptor. In this example, when an input event occurs on `myfd` the dispatcher will call `my_handler::inputReady` method to handle the event.

```
...

MyHandler my_handler;
Dispatcher &disp = Dispatcher::instance();
disp.link(myfd, Dispatcher::ReadMask, my_handler);

...
```

Figure 8-9 Instantiating an linking a handler to a file descriptor.

INTEGRATION WITH XWINDOWS

NOTE *This implementation is for single-threaded servers only.*

UNIX VisiBroker provides an `XDispatcher` class that you can use to integrate your application with the XWindows XtMainLoop. The `XDispatcher` registers the file descriptors it uses for its connections with the Xt event loop and installs the appropriate event handlers. The result is that the Xt event loop receives and dispatches events for both XWindow and VisiBroker events. When an event occurs on one of VisiBroker's file descriptors, the Xt event loop will call the appropriate VisiBroker method to process the data.

Figure 8-10 shows how you might use the `XDispatcher` class in you object implementation. Applications that use the `XDispatcher` class should link with the library **libxdispatch.a** in addition to all the other appropriate VisiBroker libraries.

```
#include <dispatch/xdisp.h>
int main(int argc, char * const *argv)
{
     // Instantiate XDispatcher before invoking any VisiBroker methods.
     XDispatcher xdisp;

     // Initialize ORB and BOA.
     CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

     ...

     boa->impl_is_ready();
     // You can call XtMainLoop() instead of impl_is_ready().

     ...
}
```

Figure 8-10 Using the `XDispatcher` *class.*

INTEGRATION WITH THE WINDOWS/NT EVENT LOOP

NOTE *This implementation is for single-threaded servers only.*

WIN VisiBroker provides a `WDispatcher` class that you can use to integrate VisiBroker events with Windows message events. The `WDispatcher` must be instantiated before any ORB object implementations are instantiated and before ORB or BOA methods are invoked. When you instantiate the WDispatcher object, you must pass it the window handle.

NOTE *There are significant advantages to building a multithreaded server rather than integrating the orb with the Windows event loop. For more information, see Multithreaded Servers and Windows 95/Windows NT later in this chapter.*

```
#include <dispatch/wdisp.h>
...
     // Windows main entry point
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR,
                         int nCmdShow)
{
     static char szAppName[] = “Library”;
     HWND hwnd;
     MSG msg;
     WNDCLASS wndclass;

     // Initialize wndclass
     ...

     hwnd = CreateWindow(szAppName, “LibraryServer”, WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT, 200, 200,
                         NULL, NULL, hInstance, NULL);

     WDispatech *winDispatechr = new WDispatcher(hwnd);
     CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
     CORBA::BOA_var orb = orb->BOA_init(__argc, __argv);

     Library server(“Harvard”);

     boa->obj_is_ready(&server);

     ShowWindow(hwnd, nCmdShow);
     UpdateWindow(hwnd);

     // Enter message loop
     while(GetMessage(&msy, NULL, 0, 0) ) {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
     return msg.wParam;
}
```

Figure 8-11 Using the `WDispatcher` *class with the Windows event loop.*

INTEGRATION WITH MICROSOFT FOUNDATION CLASSES

NOTE *This implementation is for single-threaded servers only.*

You may also use the WDispatcher class when developing client applications with the Microsoft Foundation Classes. When you derive your application class from the Microsoft CWinApp class, you need to provide an InitInstance method. The WDispatcher object should be instantiated in the InitInstance method.

```
#include <afxwin.h>
#include <dispatch/wdisp.h>
...
// Application class
class LibraryClientApp : public CWinApp
{
     public:
          BOOL InitInstance();
};

BOOL LibraryClientApp::InitInstance()
{
     m_pMainWnd = new MainWindow;
     m_pMainWnd->showWindow(m_nCmdShow);
     m_pMainWnd->UpdateWindow();

     // WDispatcher instantiation goes here?
     WDispatcher *winDispatcher = new WDispatcher(m_pMainWnd);
     CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
     return 1;
}

...
```

Figure 8-12 Using the `WDispatcher` *with MFC-based applications.*

MULTITHREADED SERVERS: WINDOWS 95 AND WINDOWS NT

It is straightforward to build multithreaded servers using VisiBroker for C++. There are significant advantages to building a multithreaded server rather than integrating the orb with the Windows event loop. The advantages are:

- Because VisiBroker for C++ automatically creates worker threads to handle incoming calls, servers can handle multiple incoming requests. On multiprocessor systems running Windows NT, servers automatically distribute work among processors. A server implemented with VisiBroker functions integrated into the Windows event loop can process only a single request at a time.
- If a single worker thread should fail, the server can continue processing other requests. A server implemented with VisiBroker functions integrated into the Windows event loop may fail if any request should fail, because there is no way for the hung thread to process other requests.

You may build multithreaded servers either directly on the Win32 API, or using MFC. The following code examples show how to initialize the orb in both cases.

The following code example shows how to initialize the orb for a multithreaded Win32 server without using MFC.

EXAMPLE

```
// Windows main entry point
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR,
                              int nCmdShow)
{
                        static char szAppName[] = "Library";
                        HWND hwnd;
                        MSG msg;
                        WNDCLASS wndclass;
                        // Initialize wndclass
                        ...
                        hwnd = CreateWindow(szAppName, "LibraryServer",
WS_OVERLAPPEDWINDOW,
                                    CW_USEDEFAULT, CW_USEDEFAULT, 200, 200,
                                    NULL, NULL, hInstance, NULL);
                        CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
                        CORBA::BOA_var orb = orb->BOA_init(__argc, __argv);
                        Library server("Harvard");
                        boa->obj_is_ready(&server);


                        ShowWindow(hwnd, nCmdShow);
                        UpdateWindow(hwnd);
                        // Enter message loop
                        while(GetMessage(&msy, NULL, 0, 0) ) {
                              TranslateMessage(&msg);
                              DispatchMessage(&msg);
                        }
                        return msg.wParam;
}
```

This example is identical to the preceding WDispatcher example, except it does not create a WDispatcher object. The event loop handles normal Windows messages as usual. Orb requests, however, do not flow through the event loop. Rather, the orb automatically creates worker threads when a request comes in. These threads are completely independent of the Windows event loop.

Creating a multithreaded server using MFC is equally straightforward:

EXAMPLE
```
#include <stdafx.h>
...
// Application class
class LibraryClientApp : public CWinApp
{
                        public:
                              BOOL InitInstance();
};
BOOL LibraryClientApp::InitInstance()
{
                        m_pMainWnd = new MainWindow;
                        m_pMainWnd->showWindow(m_nCmdShow);
                        m_pMainWnd->UpdateWindow();
                        CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
                        CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
                        Library server("Harvard");
                        boa->obj_is_ready(&server);
                        return 1;
}
...
```

The InitInstance() method provides all orb initialization. After initializing the orb and boa objects, the initialization code creates the Library object and declares the object is ready.

Whether the server is built directly on the Win32 API or using MFC, the multithreaded VisiBroker library listens for incoming requests and creates worker threads to handle each request. The application need do no other thread-specific coding.

Thread-safe Code

All code within the server that implements an ORB-visible object must be thread-safe. Incoming requests execute within a VisiBroker-generated thread. Simultaneous incoming requests execute simultaneously in separate threads.

Developers must take special care when accessing a system-wide resource within an object implementation. For example, many database access methods are not thread-safe. If an object implementation must access such a database, it must lock access to the resource using a mutex or critical section.

Multithreaded Servers and Windows User Interfaces

A multithreaded server can implement a complex Windows user interface either directly on the Win32 API or using MFC.

A key point in building a multithreaded server with an MFC-based Windows user interface is that only certain threads may do user interface update. Within an MFC application, either the main application thread or a CWinThread-derived class that the application created may do user interface updates. These restrictions are because MFC threads contain thread-local storage important in doing user interface updates. Performing a user interface update from a non-MFC thread causes errors because the system does not have the required local storage within the thread.

Because VisiBroker for C++ creates a worker thread for each incoming connection, these threads are not MFC threads. Such a worker thread cannot perform user interface updates directly.

It is straightforward to interface between VisiBroker threads and MFC threads. The VisiBroker thread can post an invalidate message to the window to update. The message may contain either the needed information for update or the two threads may use a common object or data structure to pass information.

For example, a server needs to update the user interface when it handles a request. The object implementation that handles the request updates a shared data structure that contains the request count. It then posts a WM_PAINT message to the window to paint. In an MFC-based server, the worker thread can call the MFC function CWnd::Invalidate() to post the WM_PAINT message.

In either case, the window's painting code accesses the common data structure containing the needed counter and repaints the window. Because the window updates in response to a message, the painting occurs within the window's own thread.

The counter data structure is a global resource shared between threads. All code updating the counter must synchronize accesses with a mutex or a critical section. Within an MFC-based server, the classes derived from CSyncObject provide C++ wrappers for Win32 mutexes and critical sections.

INTEGRATION WITH OTHER ENVIRONMENTS

To integrate your application with another system’s event loop, you need to derive your own class from the Dispatcher class. The methods of your new class need to be implemented using the methods and interfaces provided by the event handling mechanism with which you are integrating. The details of the imple-

mentation will depend on the event system with which VisiBroker is being integrated. Figure 8-13 shows how you might create your own dispatcher class.

```
class MyDispatcher : public Dispatcher
{
     public:
          MyDispatcher();
          virtual ~MyDispatcher();
          virtual void       link(int fd, DispatcherMask, IOHandler*);
          virtual IOHandler* handler(int fd, DispatcherMask) const;
          virtual void       ulink(int fd);
          virtual void       startTimer(long sec, long usec, IOHandler *);
          virtual void       stopTimer(IOHandler *);
          virtual iv_boolean setReady(int, DispatcherMask)
                             { return 0; } // No need to implement
          virtual void       dispatch();
          virtual iv_boolean dispatch(long&, long& )
                             { return 0; } // No need to implement
          virtual iv_boolean dispatch(timeval *val);
     private:
          ...
};
```

Figure 8-13 Deriving your own dispatcher class from `Dispatcher`.

CHAPTER 9

DYNAMIC INTERFACES

This chapter discusses how client applications can use the Interface Repository to discover object interfaces and dynamically create requests for using those interfaces. It includes the following major sections:

DYNAMIC INVOCATION INTERFACE

The Dynamic Invocation Interface lets your client applications use any registered object without having to first link the client stubs created for that object by the IDL compiler. With the DII, your client application can dynamically build requests for any object interface that has been stored in the Interface Repository. Even recently registered object can be accessed by a client application using the DII. Your object implementations are not required to provide any extra code to handle DII requests.

While client applications that use the DII are not as efficient as applications that use statically-linked client stubs, they offer some important advantages. Clients are not restricted to using just those objects that were defined at the time the client application was compiled. In addition, client applications do not need to be re-compiled in order to access newly added object implementations.

Steps for Dynamic Invocation

There are five steps that a client follows for dynamic invocation.

1 Retrieve an object's interface definition from the interface repository.

2 Identify and retrieve the desired operation definition from the object's interface definition.

3 Bind to the object and obtain an object reference.

4 Create the dynamic invocation request.

5 Invoke the request and receive the results.

THE INTERFACE REPOSITORY

The Interface Repository (IR) contains information on a variety of objects that the ORB or a client application may need to access. The IR offers an object interface that provides your client applications with a variety of methods for obtaining the interfaces offered by all currently active objects. Your client application can bind to the `Repository` and then invoke the methods defined by the `Repository` class to locate object implementations. Table 9-1 shows the various types of objects that can be contained in the IR. A complete description of this class can be found in the `VisiBroker for C++ Reference Guide`.

OBJECT TYPE	DESCRIPTION
Repository	Represents the top-level module that contains all other objects in this repository.
ModuleDef	Contains a grouping of interfaces. Can also contain constants, typedefs and even other ModuleDef objects.
InterfaceDef	Contains a list of operations, exceptions, typedefs, constants and attributes that make up an interface.
AttributeDef	Defines an attribute associated with an interface.
OperationDef	Defines an operation on an interface. It includes a list of parameters required for this operation and a list of exceptions that may be raised by this operation.
TypedefDef	Defines a base interface for named types that are not interfaces.
ConstantDef	Defines a named constant.
ExceptionDef	Defines an exception that may be raised by an operation.

Table 9-1 Objects that can be stored in the IR.

```
class CORBA {

    class Repository : public Container {
              Contained_ptr lookup_id(const char * search_id);
              PrimitiveDef_ptr get_primitive(PrimitiveKind kind);
              StringDef_ptr create_string(ULong bound);
              SequenceDef_ptr create_sequence(CORBA::ULong bound,
                      IDLType_ptr element_type);
              ArrayDef_ptr create_array(ULong length,
                      IDLType_ptr element_type);
    };
    ...
};
```

Figure 9-2 The `Repository` *Class.*

Obtaining an Object's Interface

The library client application, introduced in Chapter 2, could be enhanced to dynamically obtain the `Library` interface and obtain information about the `add_book` operation.

```
#include <lib_client.hh>
main(int argc, char *const *argv)
{
     CORBA::Boolean    ret;
     // Initialize the ORB
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

     // Declare the library object
     library_var library_object;

     // Declare an interface repository pointer
     CORBA::Repository_var rep_object;

     try {
          // Attempt to bind to the interface repository
          rep_object = CORBA::Repository::_bind();
     }
     // Check for errors
     catch(const CORBA::Exception& excep) {
          cout << "Error binding to interface repository" << endl;
          return(0);
     }

     // Locate the add_book operation definition. Can the operation be
     // located without first locating the interface?
     CORBA::Contained_var add_req = rep_object
          ->lookup_id("Library::add_book");

     try {
          // Bind to the library object.
          library_object = library::bind();
     }
     // Check for errors
     catch(const CORBA::Exception& excep) {
          cout << "Error binding to library object" << endl;
          return(0);
     }

     // Create a request, initializing the operation name.
     Request_var req = library_object->_request(add_req->name());
     ...
}
```

Figure 9-3 Dynamically obtaining the `Library::add_book` *method.*

THE REQUEST CLASS

When your client application invokes a method on an object, a `Request` must be created to represent the method invocation. This `Request` is written to a buffer and sent to the object implementation. When your client application uses client stubs, this processing occurs transparently. Client applications that use the DII must create and send the Request themselves. Figure 9-4 shows the `Request` class.

NOTE *There is no constructor for this class. The* `Object::_request` *method or Object::_create_request methods are used to create a* `Request` *object, given an object reference.*

```
class CORBA {
     class Request {
          public:
               CORBA::Object_ptr          target() const;
               const char*                operation() const;
               CORBA::NVList_ptr          arguments();
               CORBA::NamedValue_ptr      result();
               CORBA::Environment_ptr     env();

               void                       ctx(CORBA::Context_ptr ctx);
               CORBA::Context_ptr         ctx() const;

               CORBA::Status              invoke();
               CORBA::Status              send_oneway();
               CORBA::Status              send_deferred();
               CORBA::Status              get_response();
               CORBA::Status              poll_response();
               ...
     };
};
```

Figure 9-4 The `Request` *class.*

The `target` is set implicitly from the object reference used to create the `Request`. The name of the `operation` must be specified when the Request is created. The initialization of the remaining properties is covered in “Initializing a DII Request” on page 9-7. A complete description of this class can be found in the *VisiBroker for C++ Reference Guide*.

CREATING A DII REQUEST

Once you have obtained the interface to an object, issued a bind to that object and obtained an object reference, you can use one of two methods for creating a `Request` object. Figure 9-5 shows the methods offered by the `Object` class.

```
class CORBA {
     class Object {
          ...
          Status        _create_request(Context_ptr    ctx,
                              const char *             operation,
                              NVList_ptr               arg_list,
                              NamedValue_ptr           result,
                              Request_ptr              request,
                              Flags                    req_flags);

          Request_ptr   _request(Identifier operation);
          ...
     };
};
```

Figure 9-5 Two methods for creating a `Request` *object.*

You can use the _create_request method to create a `Request` object, initializing the `Context`, the `operation` name, the argument list to be passed and the result. The request parameter points to the `Request` object that was created for this operation. The `req_flags` must be set to OUT_LIST_MEMORY if one or more of the arguments in the `arg_list` are output parameters.

You can also use the `_request` method to create a `Request` object, specifying only the operation name. You must then perform the rest of the initialization manually.

INITIALIZING A DII REQUEST

Setting the Context

The `Context` object contains a list of properties, stored as `NamedValue` objects, that are passed to the object implementation as part of the `Request`. These properties represent information that would otherwise be difficult to communicate to the object implementation. A complete description of this class can be found in the `VisiBroker for C++ Reference Guide`.

```
class CORBA {
     class Context {
          public:
               const char           *context_name() const;
               CORBA::Context_prt   parent();
               CORBA::Status        create_child(const char *name,
                                          CORBA::Context_ptr&);
               CORBA::Status        set_one_value(const char *name,
                                          const CORBA::Any&);
               CORBA::Status        set_values(CORBA::NVList_ptr);
               CORBA::Status        delete_values(const char *name);
               CORBA::Status        get_values(const char *start_scope,
                                          CORBA::Flags,
                                          const char *name,
                                          CORBA::NVList_ptr&) const;
};
```

Figure 9-6 The `Context` *class.*

Setting the Arguments

The arguments for a `Request` are represented with a `NVList` object, which stores name-value pairs as `NamedValue` objects. You can use the arguments method to obtain a pointer to the arguments. This pointer can then be used to set the names and values of each of the arguments.

THE NVLIST

This class implements a list of NamedValue objects that represent the arguments for a method invocation. Methods are provided for adding, removing and querying the objects in the list. A complete description of this class can be found in the *VisiBroker for C++ Reference Guide*.

```
class NVList {
     public:
          Long                count() const;
          NamedValue_ptr      add(Flags);
          NamedValue_ptr      add_item(const char *name, Flags);
          NamedValue_ptr      add_value(const char *name, const Any&, Flags);
          NamedValue_ptr      item(Long);
          Status              remove(Long);
          Status              free_out_memory();
};
```

THE NAMEDVALUE

This class implements a name-value pair that represents both input and output arguments for a method invocation request.The NamedValue class is also used to represent the result of a request that is returned to the client application. The name property is simply a character string and the value property is represented by an Any class. A complete description of this class can be found in the *VisiBroker for C++ Reference Guide*.

```
class NamedValue {
     public:
          const char    *name() const;
          Any           *value() const;
          Flags         flags() const;
};
```

Figure 9-7 The NamedValue *class.*

METHOD	DESCRIPTION
name()	Returns a pointer to the name of the item that you can then use to initialize the name.
value()	Returns a pointer to an Any object representing the item's value that you can then use to initialize the value. For more information, see "The Any Class" on page 9-10.
flags()	Indicates if this item in an input argument, an output argument or both an input and output argument. If the item is both an input and output argument, you can specify a flag indicating that the ORB should make a copy of the argument and leave the caller's memory intact. Flags are: ARG_IN ARG_OUT ARG_INOUT IN_COPY_VALUE

Table 9-8 The `NamedValue` *class methods.*

The Any Class

This class is used to represent any IDL type so that they may be passed in a type-safe manner. Objects of this class have a pointer to a TypeCode that defines the object's type and a pointer to the value associated with the object. Methods are provide to construct, copy and destroy an object as well as initialize and query the object's properties. In addition, streaming operators are provided to write the object to a stream. A complete description of this class can be found in the *VisiBroker for C++ Reference Guide.*

```
class Any
{
     public:
          Any();
          Any(const Any&);
          Any(TypeCode_ptr tc, void *value, Boolean release=0);
          ~Any();

          Any& operator=(const Any&);
          // Overloaded operators for all data types
          void operator<<=(Short);
          void operator<<=(UShort);
          void operator<<=(Long);
          void operator<<=(ULong);
          ...
          TypeCode_ptr       type();
          const void         *value() const;
          static Any_ptr     _nil();
          static Any_ptr     _duplicate(Any *ptr);
          static void        _release(Any *ptr);

          // Streaming operators to write Anys to stdout, etc.
          ostream& operator<<(ostream&, const Any&);
          istream& operator>>(istream& strm, Any& any);
          istream& operator>>(istream& strm, Any_ptr& any);
          ...
}
```

Figure 9-9 The Any *class.*

The TypeCode Class

This class is used by the Interface Repository and the IDL compiler to represent the type of arguments or attributes.TypeCode objects are also used in the DII to specify an argument's type in conjunction with the Any class. TypeCode objects have a kind property and parameter list property. A complete description of this class can be found in the VisiBroker for C++ Reference Guide.

TYPECODE CONST	KIND	PARAMETER LIST
TC_null	tk_null	None
TC_void	tk_void	None
TC_short	tk_short	None
TC_long	tk_long	None
TC_ushort	tk_ushort	None
TC_ulong	tk_ulong	None
TC_float	tk_float	None
TC_double	tk_double	None
TC_boolean	tk_boolean	None
TC_char	tk_char	None
TC_octet	tk_octet	None
TC_any	tk_any	None
TC_TypeCode	tk_TypeCode	None
TC_Principal	tk_Principal	None
TC_Object	tk_objref	interface_id
Structure (const generated)	tk_struct	struct-name, {member, TypeCode}
Union (const generated)	tk_union	union-name, switch TypeCode, {label-value, member-name, TypeCode}
Enum (const generator)	tk_enum	{enum-name, enum-id}
TC_string	tk_string	maxlen
Sequence (const generator)	tk_sequence	TypeCode, maxlen
Array (const generator)	tk_array	TypeCode, length

Table 9-10 TypeCode *kinds and their associated parameter lists.*

TYPE	NAME
TypeCode_ptr	_tc_null
TypeCode_ptr	_tc_void
TypeCode_ptr	_tc_short
TypeCode_ptr	_tc_long
TypeCode_ptr	_tc_ushort
TypeCode_ptr	_tc_ulong
TypeCode_ptr	_tc_float
TypeCode_ptr	_tc_double
TypeCode_ptr	_tc_boolean
TypeCode_ptr	_tc_char
TypeCode_ptr	_tc_octet
TypeCode_ptr	_tc_Any
TypeCode_ptr	_tc_TypeCode
TypeCode_ptr	_tc_Principal
TypeCode_ptr	_tc_Object
TypeCode_ptr	_tc_string
TypeCode_ptr	_tc_NamedValue

Table 9-11 TypeCode *constants for IDL data types.*

SENDING A DII REQUEST

The `Request` class provides several methods for sending the request, once it has been properly initialized. The simplest of these is the `invoke` method which sends the request and blocks waiting for a response before returning to your client application. The non-blocking method `send_deferred` allows your client to send the request and then use the `poll_response` method to determine when the response is available. The `get_response` method blocks until a response is received.

The `send_oneway` method can be used to send a oneway request. Oneway requests do not involve a response being sent from the object implementation.

The `result` method returns a pointer to a `NamedValue` object that represents the return value.

EXAMPLE

```
                          ...
                          // Assumes that req has been set to Request
                          // See page 9-4
                          // Create TypeCode for structure
                          CORBA::StructMemberSeq members;
                          members.length(2);
                          members[0].name = (const char *)"author";
                          members[0].type =
CORBA::TypeCode::_duplicate(CORBA::_tc_string);
                          members[1].name = (const char *)"title";
                          members[1].type =
CORBA::TypeCode::_duplicate(CORBA::_tc_string);


                          bookTypeCode = orb->create_struct_tc(
                                "book", "book", members);

                          // Write out author and title to a MarshalOutBuffer
                          CORBA::MarshalOutBuffer buf;
                          buf << argv[1];// Author
                          buf << argv[2];// Title
                          bookValue.replace(bookTypeCode, buf);
                          // Get Argument list from request.
                          CORBA::NVList_var arguments = req->arguments();
```

```
                        arguments->add_value("book", bookValue, CORBA::ARG_IN);

                        // Set result
                        // NOTE: All parameters types (IN, OUT, INOUT and
RETURN) need
                        //    to be set so that DII knows the data types of all
                        //    arguments.
                        CORBA::Boolean ret=0;
                        CORBA::NamedValue_var result(req->result());
                        CORBA::Any_var  resultAny(result->value());
                        resultAny->replace(CORBA::_tc_boolean, &result);

                        // Execute the function
                        req->invoke();
                        CORBA::Environment_var env = req->env();
                        if ( env->exception() )
                              cout << "Exception occured" << endl;
                        else  {
                              // Get the return value;
                              ret = *(CORBA::Boolean *)resultAny->value();
                        }
                        cout << "Return value from invoke: " << (int)ret <<
endl;
                        return(1);
}
```

Sending and Receiving Multiple Requests

A sequence of DII `Request` objects can be created using `RequestSeq`, defined in the `CORBA::ORB` class and shown in Figure 9-12 . A sequence of requests can be sent using the ORB methods `send_multiple_requests_oneway` or `send_multiple_requests_deferred`. If the sequence of requests is sent as oneway requests, no response is expected from the server to any of the requests.

If the requests in the sequence are sent using `send_multiple_requests_deferred`, the `poll_next_response` and `get_next_response` methods are used to receive the response the server sends for each request.

The ORB method `poll_next_response` can be used to determine if a response has been received from the server. This method returns one if one or more responses are available. This method returns zero if there are no responses available.

The ORB method `get_next_response` can be used to receive a response. If no response is available, this method will block until a response is received. If you do not wish your client application to block, use the `poll_next_response` method to determine when a response is available.

```
class CORBA {
     class ORB {
          ...
          typedef       sequence<Request_ptr> RequestSeq;
          Status        send_multiple_requests_oneway(const RequestSeq &);
          Status        send_multiple_requests_deferred(const RequestSeq &);
          Boolean       poll_next_response();
          Status        get_next_response();
          ...
     };
};
```

Figure 9-12 ORB methods for sending multiple requests and receiving the results.

CHAPTER 10

THE IDL COMPILER

This chapter discusses the VisiBroker IDL compiler and includes the following major sections:

THE IDL COMPILER

You use the Interface Definition Language, IDL, to define the object interfaces that client applications may use. The IDL compiler uses your interface definition to generate C++ code. Figure 10-1 shows how the compiler generates code for the client application and for the object implementation, or server. The file names used for discussion in this chapter apply to systems that support long file names.

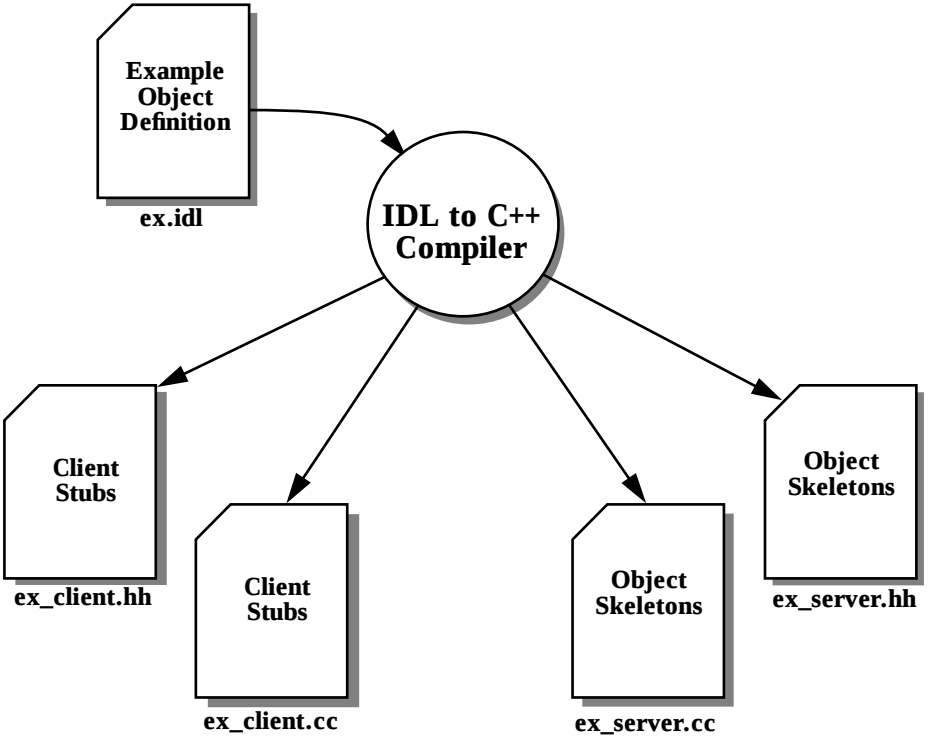


Figure 10-1 **C++** *files generated by the IDL compiler.*

The Interface Definition

Your interface definition defines the name of the object as well as all of the methods the object offers. Each method specifies the parameters that will be passed to the method, their type and whether they are for input or output. Figure 10-2 shows an IDL specification for an object named `example`. The `example` object has only one method, `op1`.

```
// IDL specification for the example object
interface example
{
     long op1(in char x, out short y);
};
```

Figure 10-2 The `example` *IDL specification.*

CODE GENERATED FOR CLIENTS

Figure 10-1 shows how the IDL compiler generates two client files; **ex_client.hh** and **ex_client.cc**. These two files provide an `example` class in **C++** that the client will use. Files generated by the IDL compiler always have either a ".cc" or ".hh" suffix to make them easy to distinguish from file you create yourself.

You should not modify the contents of the files generated by the IDL compiler.

```
class example : public virtual CORBA::Object
{
     private:
          // Methods used internally by VisiBroker to store type information
          ...
     public:
          // More methods used internally by VisiBroker to create object
          // references and manage type information
     protected:
          example(const char *obj_name = NULL) : CORBA::Object(obj_name, 1);
          example(NCistream& strm) :CORBA::Object(strm);
          virtual ~example();
     public:
          static example_ptr _bind(const char *object_name = NULL,
                                   const char *host_name = NULL,
                                   const CORBA::BindOptions* opt = NULL);
          static example_ptr _duplicate(example_ptr obj);
          static example_ptr _nil();
          static example_ptr _narrow(CORBA::Object *obj)
          virtual CORBA::Long op1(CORBA::Char x, CORBA::Short& y);
};
```

Figure 10-3 The `example` *class generated in ex_client.hh.*

Methods Generated

Figure 10-3 shows the `op1` method generated by the IDL complier, along with several other methods. The `op1` method is called a *stub* because when your client application invokes it, it actually packages the interface request and arguments into a message, sends the message to the object implementation, waits for a response, decodes the response, and reflects the results to your application.

Since the example class is derived from the CORBA::Object class, several inherited methods are available for your use. The CORBA::Object class methods are described in the VisiBroker for **C++** Reference Guide.

The _ptr Definition

The IDL compiler always provides a pointer type definition. Figure 10-4 shows the type definition for the example class.

```
typedef example *example_ptr;
```

Figure 10-4 The _ptr type definition.

The _var Class

The IDL compiler also generates a class named `example_var`, which you can use instead of the `example` class. The `example_var` class will automatically manage the memory associated with the object reference.When the an `example_var` object is deleted, the object associated with `example_ptr` is released. When an `example_var` object is assigned, the old object reference pointed to by `example_ptr` is released after the assignment takes place. A casting operator is also provided to allow you to assign an `example_var` to a type `example_ptr`.

```
class example_var
{
     public:
          example_var();
          example_var(example_ptr ptr);
          example_var(const example_var& var);
          ~example_var();
          example_var& operator=(example_ptr p);
          example operator=(const example_ptr p);
          example_ptr operator->();
          ...
     protected:
          example_ptr   _ptr;
     private:
          ...
};
```

Figure 10-5 The `example_var` *class.*

METHOD	DESCRIPTION
example_var()	Constructor that initializes the _ptr to NULL.
example_var(example_ptr ptr)	Constructor that creates an object with the _ptr initialized to the argument passed. When the object is destroyed, the object to which `_ptr` points will destroyed.
example_var(const example_var& var)	Constructor that makes a copy of the object passed as a parameter `var` and points `_ptr` to the newly copied object. When this object is destroyed, the object to which `_ptr` points will be destroyed.
~example()	Destructor that frees any memory associated with `_ptr` before destroying this object.
operator=(example_ptr p)	Assignment operator that frees any memory associated with _ptr before performing the assignment.
operator=(const example_ptr p)	Assignment that frees any memory associated with _ptr before making a complete copy of the specified object and assigning _ptr to point to the newly created object.
operator->()	Returns the _ptr stored in this class. This operator should not be called until this class has been properly initialized.

Table 10-6 The _var class method descriptions.

CODE GENERATED FOR SERVERS

Figure 10-1 shows how the IDL compiler generates two server files: `ex_server.hh` and `ex_server.cc`. These two files provide an `_sk_example` class in **C++** that the server will use to derive an implementation class. The `_sk_example` class is derived from the client's `example` class.

CAUTION *You should not modify the contents of the files generated by the IDL compiler.*

```
class _sk_example : public example
{
     protected:
          _sk_example(const char *object_name = (const char *)NULL);
          virtual ~_sk_example();
     public:
          static const CORBA::TypeInfo _skel_info;
          virtual CORBA::Long op1(CORBA::Char x, CORBA::Short& y) = 0;
          static void _op1(void *obj), CORBA::MarshalStream &strm,
                              CORBA::Principal_ptr principal,
                              const char *oper);
};
```

Figure 10-7 The `_sk_example` *class generated in ex_server.hh.*

Generated Methods

Notice that the `op1` method defined in the IDL specification in Figure 10-2 is generated, along with an `_op1` method. The `op1` method is a pure virtual method and must be implemented by the class you derive from `_sk_example`.

The `_op1` method is called a **skeleton** and is invoked by the BOA when a client request is received. This method will marshal all the parameters from the request, invoke the `op1` method and then marshal the return parameters or exceptions into a response message. The ORB will then send the response to the client application. Skeleton methods should not be explicitly invoked by the server or object implementation.

The constructor and destructor are both protected. The constructor accepts an object name so that multiple objects can be instantiated by a server.

The Class Template

In addition to the _sk_example class, the IDL compiler generates a class template named _tie_example. This template can be used if you wish to avoid the overhead associated with deriving a class from _sk_example. Templates can also be useful for providing a wrapper class for existing applications that cannot be modified to inherit from a new class. Figure 10-8 shows the template class generated by the IDL compiler for the example class.

```
template <class T>
class _tie_example : public example
{
     public:
          _tie_example(T& t, const char *obj_name=(char *)NULL);
          ~_tie_example();
          CORBA::Long op1(CORBA::Char x, CORBA::Short& y);
     private:
          T& _ref;
};
```

Figure 10-8 A template class generated for the example class.

USING THE TEMPLATE

To use the _tie_example template class you must first create your own Example class. Figure 10-9 shows what your Example class might look like. Notice that, unlike most object implementation classes, this Example class does not inherit from the client's example class or any class supplied by VisiBroker.

```
class Example
{
     public:
          Example();
          CORBA::Long op1(CORBA::Char x, CORBA::Short& y);
};
```

Figure 10-9 A class to be used with the _tie_example template.

Given the _tie_example template generated by the IDL compiler and the Example class you defined, Figure 10-10 shows the server's main routine.

```
void main(int argc, char * const *argv)
{
     // Initialize ORB and BOA
     CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
     CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

     // Instantiate the Example class
     Example myExample;

     // Instantiate the template, passing a reference to the
     // example object instantiated above and an ORB object instance
     // name.
     _tie_example<Example> tieExample(myExample, "test");
     boa->impl_is_ready();
     return(1);
};
```

Figure 10-10 Using the `_tie_example` *template class.*

INTERFACE ATTRIBUTES

In addition to operations, an interface specification can also define attributes as part of the interface. By default, all attributes are *read-write* and the IDL compiler will generate two methods; one to set the attribute's value and one to get the attribute's value. You can also specify *read-only* attributes.

Figure 10-11 shows an IDL specification that defines two attributes; one read-write and one read-only. Figure 10-12 shows the resulting class definition generated by the IDL compiler for the client application. Figure 10-13 shows the class definition generated for the object implementation.

```
// IDL
interface test
{
     attribute long count;
     readonly attribute string name;
};
```

Figure 10-11 IDL specification with two attributes; one read-write and one read-only.

```
class test : public virtual CORBA::Object
{
     ...
     // Methods for read-write attribute
     virtual CORBA::Long count();
     virtual void count(CORBA::Long val);

     // Method for read-only attribute.
     virtual char * name();
     ...
};
```

Figure 10-12 The class generated for the client application.

```
class _sk_test : public test
{
     virtual CORBA::Long count() = 0;
     virtual void count(CORBA::Long val) = 0;
     virtual char * name() = 0;
};
```

Figure 10-13 The class generated the server.

ONEWAY METHODS

The IDL allows you to specify operations that have no return value, called *oneway* methods. These operations may only have input parameters. When a oneway method is invoked, a request is sent to the server but there is no confirmation from the object implementation that the request was actually received. VisiBroker uses TCP/IP for connecting clients to servers. This provides guaranteed delivery of all datagrams so the client can be sure the request will be delivered to the server—as long as the server remains available. Still, the client has no way of knowing if the request was actually processed by the object implementation itself.

NOTE *Oneway operations cannot throw exceptions.*

```
// IDL
interface oneway_example
{
     oneway void set_value(in long val);
};
```

Figure 10-14 Defining a oneway *operation.*

```
class oneway_example : public virtual CORBA::Object
{
     virtual void set_value(CORBA::Long val);
     ...
};
```

Figure 10-15 Code generated for the client application.

```
class _sk_oneway_example : public oneway_example
{
     virtual void set_value(CORBA::Long val) = 0;
};
```

Figure 10-16 Base class generated for the implementation.

MAPPING OBJECT REFERENCES

In addition to generating **C++** classes from your interface specification, the IDL compiler will also create object references for your classes. Figure 10-17 shows the object references generated by the IDL compiler when the example interface specification shown in
Figure 10-2 is compiled.

```
typedef example *example_ptr;
typedef example_ptr exampleRef;
```

Figure 10-17 Object references generated by the IDL compiler.

INTERFACE INHERITANCE

IDL allows you to specify an interface that inherits from another interface. The **C++** classes generated by the IDL compiler will reflect the inheritance relationship. All methods, data type definitions, constants and enumerations declared by the parent interface will be visible to the derived interface.

```
// IDL
interface parent
{
     void operation1();
}

interface child : parent
{
     ...
     long operation2(in short s);
};
```

Figure 10-18 An example of inheritance in an interface specification.

```
...
class parent : public virtual CORBA::Object
{
     ...
     void operation1(CORBA::Environment& _env);
     ...
};

class child : public virtual parent
{
     ...
     CORBA::Long operation2(CORBA::Short s, CORBA::Environment& _env);
     ...
};
```

Figure 10-19 The **C++** *code generated from Figure 10-19 .*

CHAPTER 11

IDL TO C++ LANGUAGE MAPPING

This chapter discusses the IDL to C++ language mapping provided by the VisiBroker IDL to C++ compiler, which complies strictly with the CORBA C++ language mapping specification. This chapter includes the following major sections:

PRIMITIVE DATA TYPES

The basic data types provided by the Interface Definition Language are summarized in Table 11-1 . Due to hardware differences between platform, some of the IDL primitive data types have a definition that is marked “platform dependent.” On a platform that has 64-bit integral representations, for example, the `CORBA::Long` type would still be only 32 bits. You should consult the include file **orbtypes.h** for an exact mapping of these primitive data types for your particular platform.

IDL TYPE	VISIBROKER TYPE	C++ DEFINITION
SHORT	CORBA::SHORT	SHORT
LONG	CORBA::LONG	PLATFORM DEPENDENT
UNSIGNED SHORT	CORBA::USHORT	UNSIGNED SHORT
UNSIGNED LONG	CORBA::ULONG	UNSIGNED LONG
FLOAT	CORBA::FLOAT	FLOAT
DOUBLE	CORBA::DOUBLE	DOUBLE
CHAR	CORBA::CHAR	CHAR
BOOLEAN	CORBA::BOOLEAN	UNSIGNED CHAR
OCTET	CORBA::OCTET	UNSIGNED CHAR
LONGLONG	CORBA::LONGLONG	PLATFORM DEPENDENT
ULONGLONG	CORBA::ULONGLONG	PLATFORM DEPENDENT

Table 11-1 The mapping of primitive data types.

The IDL boolean type is defined by the CORBA specification to have only one of two values: 1 or 0. Using other values for a boolean will result in undefined behavior.

STRINGS

String types in IDL may specify a length or may be unbounded, but both are mapped to the **C++** type `char *`. You must use the functions shown in Figure 11-2 for dynamically allocating strings to ensure that your applications and VisiBroker use the same memory management facilities. All CORBA string types are null-terminated.

```
class CORBA
{
     ...
     static char *string_alloc(CORB::ULong len);
     static void string_free(char *data);
     ...
};
```

Figure 11-2 Methods for allocating and freeing memory for strings.

METHOD	DESCRIPTION
CORBA::string_alloc	Dynamically allocates a string and returns a pointer to the string. A `NULL` pointer is returned if the allocation fails. The length specified by the `len` parameter does not need to include the `NULL` terminator.
CORBA::string_free	Releases the memory associated with a string that was allocated with `CORBA::string_alloc`.

Table 11-3 CORBA string allocation and release methods.

String_var Class

In addition to mapping an IDL `string` to a `char *`, the VisiBroker IDL to **C++** compiler generates a `String_var` class that contains a pointer to the memory allocated to hold the string. When a String_var object is destroyed or goes out of scope, the memory allocated to the string is automatically freed. Figure 11-4 shows the `String_var` class and the methods it supports. For detailed information on the _var classes, see “The _var Class” on page 10-4.

```
class CORBA {
     class String_var {
          protected:
               char     *_p;
               ...
          public:
               String_var();
               String_var(char *p);
               ~String_var();
               String_var&    operator=(const char *p);
               String_var&    operator=(char *p);
               String_var&    operator=(const String_var& s);
               operator const char *() const;
               operator char *();
               char &operator[](CORBA::ULong index);
               char operator[](CORBA::ULong index) const;
               friend ostream&      operator<<(ostream&, const String_var& );
               inline friend Boolean operator==(const String_var& s1,
                                                const String_var& s2);
               ...
     };
     ...
};
```

Figure 11-4 The `String_var` *class.*

CONSTANTS

Figure 11-6 shows how IDL constants defined outside of any interface specification will be mapped directly to a C++ constant declaration. Figure 11-6 shows how constants defined within an interface specification are declared in the include file and assigned a value in the source file.

```
// These top-level definitions in IDL
const string           str_example = “this is an example”;
const long             long_example = 100;
const boolean          bool_example = TRUE;
...

// Result in the generation of this C++ code
const char *           str_example = “this is an example”;
const CORBA::Long      long_example = 100;
const CORBA::Boolean   bool_example = 1;
```

Figure 11-5 Top-lever constant definitions in IDL and the resulting C++ *code.*

```
// These definitions are in the IDL file example.idl
interface example {
     const string       str_example = “this is an example”;
     const long         long_example = 100;
     const boolean      bool_example = TRUE;
};
...

// Result in the generation of this C++ code in example_client.hh
class example :: public virtual CORBA::Object
{
     ...
     static const char *            str_example; /* “this is an example” */
     static const CORBA::Long       long_example; /* 100 */
     static const CORBA::Boolean    bool_example; /* 1 */
     ...
};

...

// And the generation of this C++ code in example_client.cc
const char *           example::str_example = “this is an example”;
const CORBA::Long      example::long_example = 100;
const CORBA::Boolean   example::bool_example = 1;
```

Figure 11-6 Constant definitions within an interface specification and the resulting **C++** *code.*

Under some circumstances, the IDL compiler must generate **C++** code containing the value of an IDL constant rather than the name of the constant. Figure 11-7 shows how the value of the constant len must be generated for the typedef V to allow the **C++** code to compile properly.

```
// IDL
interface foo {
     const long length = 10;
     typedef long V[length];
};
...

// Results in this C++ code being generated by the IDL compiler
class foo : public virtual CORBA::Object
{
     const CORBA::Long length;
     typedef CORBA::Long V[10];
};
```

Figure 11-7 Sometimes the value of an IDL constant must be generated in **C++**.

ENUMERATIONS

Figure 11-8 shows how enumerations in IDL map directly to **C++** enumerations.

```
// IDL
enum enum_type {
     first,
     second,
     third
};

// Results in this C++ code
enum enum_type {
     first,
     second,
     third
};
```

Figure 11-8 Enumerations in IDL map directly to **C++**.

TYPE DEFINITIONS

Figure 11-9 shows how type definitions in IDL are mapped directly to **C++** type definitions. If the original IDL type definition maps to several **C++** types, the IDL compiler generates the corresponding aliases for each type in **C++**. Figure 11-10 and Figure 11-11 show other type definition mapping examples.

```
// IDL
typedef octet          example_octet;
typedef enum enum_values {
     first,
     second,
     third
} enum_example;

// Results in the generation of this C++ code
typedef octet          example_octet;
enum enum_values {
     first,
     second,
     third
};
typedef enum_values enum_example;
```

Figure 11-9 The mapping of simple type definitions from IDL to **C++**

```
// IDL
interface A1;
typedef A1 A2;
...

// Results in the generation of this C++ code
class A1;
typedef A1 *A1_ptr;
typedef A1_ptr A1Ref;
class A1_var;

typedef A1 A2;
typedef A1_ptr A2_ptr;
typedef A1Ref A2Ref;
typedef A1_var A2_var;
```

Figure 11-10 Mapping an IDL interface type definition to **C++**.

```
// IDL
typedef sequence<long> S1;
typedef S1 S2;
...

// Results in the generation of this C++ code
class S1;
typedef S1 *S1_ptr;
typedef S1_ptr S1Ref;
class S1_var;

typedef S1 S2;
typedef S1_ptr S2_ptr;
typedef S1Ref S2Ref;
typedef S1_var S2_var;
```

Figure 11-11 Mapping an IDL sequence type definition to **C++**.

For more information, see “The _var Class” on page 10-4.

MODULES

The OMG IDL to **C++** language mapping specifies that an IDL `module` should be mapped to a **C++** `namespace` with the same name. Since few compilers currently support the `namespace`, the **C++** language mapping allows the use of `class` in its place. Figure 11-12 shows how VisiBroker’s IDL compiler maps `module` to `class`.

```
// IDL
module ABC
{
     // Definitions
     ...
};

...

// Results in the generation of this C++ code
class ABC
{
     // Definitions
     ...
};
```

Figure 11-12 Mapping an IDL `module` *to a* **C++** `class`.

COMPLEX DATA TYPES

The **C++** mappings for IDL structures, unions, sequences and arrays depend on whether or not the data members they contain are of a fixed or variable length. These types are considered to have variable lengths and, consequently, any complex data type that contains them will also have a variable length.

- The `any` type.
- The `string` type, bounded or unbounded.
- The `sequence` type, bounded or unbounded.
- An object reference.
- Other `structures` or `unions` that contain a variable-length member.
- An `array` with variable-length elements.
- A `typedef` with variable-length elements.

Table 11-13 shows a summary of the **C++** mappings for complex data types.

IDL TYPE	C++ MAPPING
STRUCT (FIXED LENGTH)	STRUCT
STRUCT (VARIABLE LENGTH)	STRUCT (_VAR TYPES FOR VARIABLE LENGTH MEMBERS)
UNION	CLASS AND _VAR CLASS
SEQUENCE	CLASS AND _VAR CLASS
ARRAY	ARRAY

*Table 11-13 Summary of the **C++** mappings for complex data types.*

Fixed-length Structures

Figure 11-14 shows how fixed-length structures in IDL are mapped to **C++** code. In addition to the structure, VisiBroker's IDL compiler will also generate an example_var class for the structure. For more information, see "The _var Class" on page 10-4. Figure 11-15 shows how you might use the struct and class.

```
// IDL
struct example {
     short a;
     long b;
};
...

// Results in the generation of this C++ code.
struct example {
     CORBA::Short a;
     CORBA::Long b;
};

class example_var
{
     ...
     private:
          example *_ptr;
};
```

Figure 11-14 Mapping fixed-length a IDL structure to **C++**.

```
// Declare an example struct and initialize its fields.
example ex1 = { 2, 5 };

// Declare a _var class and assign it to a newly created example structure.
// This results in the _ptr pointing to an allocated struct with
// uninitialized fields.
example_var ex2 = new example;

// Initialize the fields of ex2 from ex1
ex2->a = ex1.b;
```

Figure 11-15 Use of the `example` *structure and the* `example_var` *class.*

To access the fields of the `_var` class `ex2`, the `->` operator must always be used. When `ex2` goes out of scope, the memory allocated to it will be freed automatically.

Variable Length Structures

Figure 11-16 shows how you could modify the `example` structure, replacing the `long` member with a `string` and adding an object reference, to change to a variable-length structure.

```
// IDL
interface ABC {
     // Definitions ...
};
struct vexample {
     short              a;
     ABC                c;
     string             name;
};
...

// Results in the generation of this C++ code
struct vexample {
     CORBA::Short       a;
     ABC_var            c;
     CORBA::String_var  name;
     vexample& operator=(const vexample& s);
};

class vexample_var {
     ...
};
```

Figure 11-16 Mapping a variable-length structure to **C++**.

Notice how the ABC object reference is mapped to an `ABC_var` class. In a similar fashion, the string `name` is mapped to a `CORBA::String_var` class. In addition, an assignment operator is also generated for variable-length structures.

MEMORY MANAGEMENT FOR STRUCTURES

The use of _var classes in variable-length structures ensures that memory allocated to the variable-length members are managed transparently.

- If a structure goes out of scope, all memory associated with variable-length members is automatically freed.
- If a structure is initialized or assigned and then re-initialized or re-assigned, the memory associated with the original data is always freed.
- When a variable-length member is assigned to an object reference, a copy is always made of the object reference. If a variable-length member is assigned to a pointer, no copying takes place.

Unions

Figure 11-18 shows how an IDL `union` is mapped to a C++ `class` with methods for setting and retrieving the value of the data members. A data member named `_d` of the discriminant type is also defined. The value of this discriminant is not set when the union is first created, so an application must set it before using the union. Setting any data member using one of the provided methods automatically sets the discriminant. Table 11-17 describes some of the methods in the un_ex class.

METHOD	DESCRIPTION
un_ex()	The default constructor sets the discriminant to zero but does not initialize any of the other data members.
un_ex(const un_ex& obj)	The copy constructor performs a deep copy of the source object.
~un_ex()	The destructor frees all memory owned by the union.
operator=(const un_ex& obj)	The assignment operator performs a deep copy, releasing old storage, if necessary.

Table 11-17 The `un_ex` *methods.*

VisiBroker's IDL compiler may also generate hash and compare methods for unions, which you can control with compiler options. See the VisiBroker for C++ Reference Guide for more information on compiler options.

```
// IDL
struct st_ex
{
     long abc;
};
union un_ex switch(long)
{
     case 1: long       x; // a primitive data type
     case 2: string     y; // a simple data type
     case 3: st_ex      z; // a complex data type
};
...

// Results in the generation of this C++ code
struct st_ex
{
     CORBA::Long        abc;
};

class un_ex
{
     private:
          CORBA::Long         _d;
          CORBA::Long         _x;
          CORBA::String_var   _y;
          st_ex               _z;
     public:
          un_ex();
          ~un_ex();
          un_ex(const un_ex& obj);
          un_ex&        operator=(const un_ex& obj);
          void          _d(CORBA::Long val);
          CORBA::Long   _d() const;
          void          x(CORBA::Long val);
          CORBA::Long   x() const;
          void          y(char *val);
          void          y(const char *val);
          void          y(const CORBA::String_var& val);
          const char    *y() const;
          const st_ex&  z() const;
          st_ex&        z();
     ...
};
```

Figure 11-18 Mapping an IDL union to a **C++** *class.*

MANAGED TYPES FOR UNIONS

In addition to the `un_ex` class shown in Figure 11-18 , a `un_ex_var` class would also be generated. See “The _var Class” on page 10-4 for details on the `_var` classes.

MEMORY MANAGEMENT FOR UNIONS

Here are some important points to remember about memory management of complex data types within a union:

- When you use an accessor method to set the value of a data member, a deep copy is performed. You should pass parameters to accessor methods by value, for smaller types, or by a constant reference, for larger types.
- When you set a data member using an accessor method, any memory previously associated with that member is freed. If the member being assigned is an object reference, the reference count of that object will be incremented before the accesor method returns.
- A `char *` accessor method will free any storage before ownership of the passed pointer is assumed.
- Both `const char *` and `String_var` accessor methods will free any old memory before the new parameter's storage is copied.
- Accessor methods for array data members will return a pointer to the array slice. For more information, see "Array Slices" on page 11-18.

Sequences

IDL sequences, both bounded and unbounded, are mapped to a **C++** class that has a current length and a maximum length. The maximum length of a bounded sequence is defined by the sequence's type. Unbounded sequences can specify their maximum length when their **C++** constructor is called. The current length can be modified programmatically. Figure 11-19 shows how an IDL sequence is mapped to a **C++** class with accessor methods.

NOTE *When the length of an unbounded sequence exceeds the maximum length you specify, VisiBroker will transparently allocate a larger buffer, copy the old buffer to the new buffer and free the memory allocated to the old buffer. However, no attempt will be made to free any unused memory if the maximum length decreases.*

```
// IDL
typedef sequence<long> LongSeq;
...

// Results in the generation of this C++ code
class LongSeq
{
     public:
          LongSeq(CORBA::ULong max=0);
          LongSeq(CORBA::ULong max=0, CORBA::ULong length,
                       CORBA::Long *data, CORBA::Boolean release = 0);
          LongSeq(const LongSeq&);
          ~LongSeq();
          LongSeq&                 operator=(const LongSeq&);
          CORBA::ULong             maximum() const;
          void                     length(CORBA::ULong len);
          CORBA::ULong             length() const;
          const CORBA::ULong&      operator[](CORBA::ULong index) const;
          ...
          static LongSeq           *_duplicate(LongSeq* ptr);
          static void              _release(LongSeq *ptr);
          static CORBA::Long       *allocbuf(CORBA::ULong nelems);
          static void              freebuf(CORBA::Long *data);
     private:
          CORBA::Long        *_contents;
          CORBA::ULong       _count;
          CORBA::ULong       _num_allocated:
          CORBA::Boolean     _release_flag;
          CORBA::Long        _ref_count;
};
```

Figure 11-19 Mapping an IDL unbounded sequence to a **C++** *class.*

METHOD	DESCRIPTION
LongSeq(CORBA::ULong max=0)	The constructor for an unbounded sequence takes a maximum length as an argument. Bounded sequences have a defined maximum length.
LongSeq(CORBA::ULong max=0, CORBA::ULong length, CORBA::Long *data, CORBA::Boolean release=0)	This constructor allows you to set the maximum length, the current length, a pointer to the data buffer associated and a release flag. If release is not zero, VisiBroker will free memory associated with the data buffer when increasing the size of the sequence. If release is zero, the old data buffer's memory is not freed. Bounded sequences have all of these parameters except for max.
LongSeq(const LongSeq&)	The copy constructor performs a deep copy of the source object.
~LongSeq();	The destructor frees all memory owned by the sequence only if the release flag had a non-zero value when constructed.
operator=(const LongSeq&j)	The assignment operator performs a deep copy, releasing old storage, if necessary.
maximum()	Returns the size of the sequence
length()	Two methods are defined for setting and returning the length of the sequence.
operator[]()	Two indexing operators are provided for accessing an element within a sequence. One operator allows the element to be modified and one allows only read access to the element.
_release()	Releases the sequence. If the constructor's release flag was non-zero when the object was created and the sequence element type is a string or object reference, each element will be released before the buffer is released.
allocbuf() freebuf()	You should use these two static methods to allocate or free any memory used by a sequence.

Table 11-20 The LongSeq *methods.*

MANAGED TYPES FOR SEQUENCES

In addition to the `LongSeq` class shown in Figure 11-19 , a `LongSeq_var` class would also be generated. See “The _var Class” on page 10-4 for details on the `_var` classes. In addition to the usual _var methods, there are two indexing methods defined for sequences.

```
CORBA::Long& operator[](CORBA::ULong index);
const CORBA::Long& operator[](CORBA::ULong idex) const ;
```

Figure 11-21 The two indexing methods added for _var classes representing sequences.

MEMORY MANAGEMENT FOR SEQUENCES

You should carefully consider the memory management issues listed below. Figure 11-22 contains sample **C++** code that illustrates these points.

- If the release flag was set to a non-zero value when the sequence was created, the sequence will assume management of the user’s memory. When an element is assigned, the old memory is freed before ownership of the memory on the right hand side of the expression is assumed.
- If the release flag was set to a non-zero value when the sequence was created and the sequence elements are strings or object references, each element will be released before the sequence’s contents buffer is released and the object is destroyed.
- Avoid assigning a sequence element using the `[]` operator unless the release flag was set to one, or memory management errors may occur.
- Sequences created with the release flag set to zero should not be used as input/output parameters because memory management errors in the object server may result.
- Always use `allocbuf` and `freebuf` to create and free storage used with sequences.

```
// Given this IDL specification for a bounded sequence
typedef sequence<string, 3> String_seq;
...

// Consider this C++ code
char *static_array[] = (“1”, “2”, “3”};
char *dynamic_array = StringSeq::allocbuf(3);

// Create a sequence, release flag is set to FALSE by default
StringSeq static_seq(3, static_array);
// Create another sequence, release flag set to TRUE
StringSeq dynamic_seq(3, dynamic_array, 1);

static_seq[1] = “1”;   // old memory not freed, no copying occurs

char *str = string_alloc(2);
dynamic_seq[1] = str;  // old memory is freed, no copying occurs
```

Figure 11-22 An example of memory management with two bounded sequences.

Arrays

IDL arrays are mapped to **C++** arrays, which can be statically initialized. If the array elements are strings or object references, the elements of the **C++** array will be of the type _var. Figure 11-23 shows three arrays with different element types.

```
// IDL
interface Intf
{
     // definitions...
};
typedef long L[10];
typedef string S[10];
typedef Intf A[10];
...

// Results in the generation of this C++ code
typedef CORBA::Long L[10];
typedef CORBA::String_var S[10];
typedef Intf_var A[10];
```

Figure 11-23 Mapping IDL arrays to **C++** *arrays.*

The use of the managed type, _var, for strings and object references, allows memory to be managed transparently when array elements are assigned.

ARRAY SLICES

The array_slice type is used when passing parameters for multi-dimensional arrays. VisiBroker's IDL compiler also generates a _slice type for arrays that contains all but the first dimension of the array. The array _slice type provides a convenient way to pass and return parameters. Figure 11-24 shows two examples of the _slice type.

```
// IDL
typedef long L[10];
typedef string str[1][2][3];
...

// Results in the generation of these slices
typedef CORBA::Long L_slice[10];
typedef CORBA::String_var str_slice[2][3];
typedef str_slice *str_slice_ptr;
```

Figure 11-24 The _slice type.

MANAGED TYPES FOR ARRAYS

In addition to generating a C++ array for IDL arrays, VisiBroker's IDL compiler will also generate a `_var` class. This class offers some additional features for array.

- The `operator[]` is overloaded to provide intuitive access to array elements.
- A constructor and assignment operator are provided that take a pointer to an array `_slice` object as an argument.

```
// IDL
typedef long L[10];
...

// Results in the generation of this C++ code
class L_var
{
     public:
          L_var();
          L_var(L_slice *slice);
          L_var(const L_var& var);
          ~L_var();
          L_var&       operator=(L_slice *slice);
          L_var&       operator=(const L_var& var);
          CORBA::Long& operator[](CORBA::ULong index);
          operator L_slice *();
          operator L &() const;
          ...
     private:
          L_slice      *_ptr;
};
```

Figure 11-25 The `_var` class generated for arrays.

TYPE-SAFE ARRAYS

A special `_forany` class is generated to handle arrays with elements mapped to the type `any`. As with the `_var` class, the `_forany` class allows you to access the underlying array type. The `_forany` class does not release any memory upon destruction because the any type maintains ownership of the memory. The `_forany` class is not implemented as a `typedef` because it must be distinguishable from other types for overloading to functions properly.

```
// IDL
typedef long L[10];
...

// Results in the generation of this C++ code
class L_forany
{
     public:
          L_forany();
          L_forany(L_slice *slice);
          ~L_forany();
          CORBA::Long&  operator[](CORBA::ULong index);
          const CORBA::Long&operator[](CORBA::ULong index) const;
          operator L_slice *();
          operator L &() const;
          operator const L & () const;
          operator const L& () const;
          L_forany& operator=(const L_forany obj);
          ...
     private:
          L_slice      *_ptr;
};
```

Figure 11-26 The `_forany` *class generated for an IDL array.*

MEMORY MANAGEMENT FOR ARRAYS

VisiBroker's IDL compiler generates two functions for allocating and releasing the memory associated with arrays. These functions allow the ORB to manage memory without having to override the `new` and `delete` operators.

```
// IDL
typedef long L[10];
...

// Results in the generation of this C++ code
inline L_slice *L_alloc();         // Dynamically allocates array. Returns
                                   // NULL on failure.

inline void L_free(L_slice *data); // Releases array memory allocated with
                                   // L_alloc.
```

Figure 11-27 Methods generated for allocating and releasing array memory.

Principal

A Principal represents information about principals requesting operations. The IDL interface of Principal does not define any operations. The Principal is implemented as a sequence of octets. The Principal is set by the client application and checked by the ORB implementation. VisiBroker for C++ treats the Principal as an opaque type. To see an example of the Principal interface in use, see Figure 7-8 in this guide.

CHAPTER 12

PARAMETER PASSING RULES

This chapter discusses the parameter passing rules followed by the VisiBroker IDL to **C++** compiler. It includes the following major sections:

IMPLICIT ARGUMENTS

Arguments can be passed using contexts as defined in IDL. For more information, see ***The Common Object Request Broker: Architecture and Specification*** - 96-03-04. This document is available from the Object Management Group and describes the architectural details of CORBA.

EXPLICIT ARGUMENTS

When you specify an interface in IDL, arguments you pass to methods that are returned may be one of the following:

MODE	DESCRIPTION
in	Parameter used as input only.
out	Parameter used to hold an output result.
inout	Parameter used both as input and to hold an output result.
return	Result of an operation on an interface.

Table 12-1 Argument types.

PRIMITIVE DATA TYPES

Table 12-2 summarizes the parameter passing mode for primitive data types.

DATA TYPE	IN	INOUT	OUT	RETURN
short	Short	Short&	Short&	Short
unsigned short	UShort	UShort&	UShort&	UShort
long	Long	Long&	Long&	Long
unsigned long	ULong	ULong&	ULong&	ULong
float	Float	Float&	Float&	Float
double	Double	Double&	Double&	Double
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum

Table 12-2 Parameter passing modes for primitive data types.

Memory Management

The following are the memory management rules for all primitive data types and parameter passing modes.

MODE	DESCRIPTION
in	The caller allocates the necessary storage and initializes it. The callee uses the value.
out	The caller allocates the necessary storage, but need not initialize it. The callee must set the value.
inout	The caller allocates the necessary storage and initializes it. The callee may change the value.
return	The callee initializes and returns the data by value. The caller receives the value.

Table 12-3 Memory management rules for primitive data types.

COMPLEX DATA TYPES

Parameter and memory management rules for aggregate data types are more complex. The issue of when memory is allocated and freed deserves special attention. Table 12-4 summarizes the parameter passing rules for complex data types.

DATA TYPE	IN	INOUT	OUT	RETURN
object reference pointer	objref_ptr	objref_ptr &	objref_ptr &	objref_ptr
struct, fixed length	const struct &	struct &	struct &	struct
struct, variable length	const struct &	struct &	struct *&	struct *
union, fixed length	const union &	union &	union &	union
union, variable length	const union &	union &	union *&	union *
string	const char *	char *&	char *&	char *
sequence	const sequence &	sequence &	sequence *&	sequence *
array, fixed length	const array	array	array	array slice *
array, variable length	const array	array	array slice *&	array slice *
any	const any &	any &	any *&	any *

Table 12-4 Parameter passing modes for complex data types.

Memory Management

The memory management rules for complex data types vary, depending on the passing mode and the type of the parameter. The following tables describe the rules for each parameter type.

Object Reference Pointers

MODE	DESCRIPTION
in	The caller allocates the necessary storage for the object reference and is responsible for freeing it when finished. The caller receives the parameter from the ORB and cannot modify it. The memory associated with the parameter is freed by the ORB upon returning. The callee can preserve the object reference by invoking the _duplicate method.
out	The caller allocates the necessary storage, but need not initialize it. Once the method returns, the storage will hold an object reference and the caller is responsible for releasing it when finished. On the server side, the ORB allocates the memory and the callee must provide the object reference. Once the data has been sent to the client, the ORB invokes the _release method on the reference to decrement its reference count.
inout	The caller allocates the necessary storage and initializes it. If the callee modifies the object reference, the ORB will release the old object and assign it a new value. If the caller wants to continue to use the object reference, it must invoke the _duplicate method prior to passing it to the callee. On the server side, the ORB will allocate memory for the reference. If the callee wishes to assign a new value to the object reference, it must first invoke the _release method.
return	On the server side, the callee initializes and returns the object reference. The ORB will invoke _release on the object reference once it has been returned to the caller. The caller receives the object reference and is responsible for releasing it.

Table 12-5 Memory management rules for object reference pointers.

Fixed Structures and Unions

MODE	DESCRIPTION
in	The caller allocates the necessary storage for the structure and is responsible for freeing it when finished. The callee receives the parameter from the ORB and cannot modify it. The memory associated with the parameter is freed by the ORB upon returning. The callee can preserve the structure by copying it.
out	The caller allocates the necessary storage, but need not initialize it. Once the method returns, the storage will hold the structure and the caller is responsible for freeing the memory when finished. On the server side, the ORB allocates the memory and the callee must set the value. Once the data has been sent to the client, the ORB releases the memory.
inout	The caller allocates the necessary storage and initializes it. Upon return, the caller must release the memory. On the server side, the ORB will allocate memory for the structure. The callee may assign a new value to the structure. Once the structure is returned to the client, the ORB releases the memory.
return	On the server side, the callee initializes and returns the data by value. The caller receives the structure by value.

Table 12-6 Memory management rules for fixed-length structures and unions.

Variable Structures and Unions

MODE	DESCRIPTION
in	The caller allocates the necessary storage for the structure and is responsible for freeing it when finished. The callee receives the parameter from the ORB and cannot modify it. The memory associated with the parameter is freed by the ORB upon returning. The callee can preserve the structure by copying it.
out	The caller allocates a pointer and passes it by reference to the ORB. Once the method returns, the caller is responsible for freeing the memory when finished. On the server side, the ORB allocates a pointer and passes it by reference to the callee. The callee sets the pointer to a valid instance of the parameter's type. If the callee wishes to keep the data buffer, it is must make a copy.
inout	The caller allocates the necessary storage and initializes it. Upon return, the caller must release the memory. On the server side, the ORB will allocate memory for the structure. If the callee wishes to change the value, it must first release the old data prior to assigning it a new value. Once the data is returned to the client, the ORB releases the memory.
return	On the server side, the callee returns to the ORB a pointer to the data buffer. The ORB will free the memory upon returning. The client cannot return a NULL pointer. The caller receives a pointer to the structure or union. If the caller wishes to modify any of the values, it must make a copy of the structure or union and modify the copy. The caller is responsible for releasing the memory.

Table 12-7 Memory management rules for variable-length structures and unions.

Strings

MODE	DESCRIPTION
in	The caller allocates the necessary storage for the string and is responsible for freeing it when finished. The callee receives the parameter from the ORB and cannot modify it. The memory associated with the parameter is freed by the ORB upon returning. The callee can preserve the string by copying it.
out	The caller allocates a pointer and passes it by reference to the ORB. Once the method returns, the caller is responsible for freeing the memory when finished. On the server side, the ORB allocates a pointer and passes it by reference to the callee. The callee sets the pointer to a valid instance of the parameter's type. If the callee wishes to keep the data buffer, it is must make a copy. The callee is not allowed to return a NULL pointer.
inout	The caller allocates the necessary storage for both input string the char * pointing to it. Upon return, the caller must release the memory using the string_free method. The ORB will delete the old buffer and allocate a new buffer for the out parameter. The size of the output string may be larger that the input string. On the server side, the ORB will allocate memory for the string. To return a new string, the callee must free the old memory using string_free and allocate new storage using the string_alloc method. Once the data is returned to the client, the ORB releases the memory. The callee may not return a NULL pointer.
return	On the server side, the callee returns to the ORB a pointer to the data buffer. The buffer must have been allocated using string_alloc. The ORB will free the memory upon returning. The client cannot return a NULL pointer. The caller receives a char * pointer to the string. If the caller wishes to modify any of the values, it must make a copy of the string and modify the copy. The caller is responsible for releasing the memory using string_free.

Table 12-8 Memory management rules for strings.

Sequences and Type-safe Arrays

MODE	DESCRIPTION
in	The caller allocates the necessary storage for the structure and is responsible for freeing it when finished. The callee receives the parameter from the ORB and cannot modify it. The memory associated with the parameter is freed by the ORB upon returning. The callee can preserve the data buffer by copying it or increasing the object's reference count.
out	The caller allocates a pointer and passes it by reference to the ORB. Once the method returns, the caller is responsible for freeing the memory when finished. On the server side, the ORB allocates a pointer and passes it by reference to the callee. The callee sets the pointer to a valid instance of the parameter's type. If the callee wishes to keep the data buffer, it is must make a copy or increase the object's reference count.
inout	The caller allocates the sequence or any and initializes it. The ORB may free the old buffer and allocate a new buffer for the output parameter., depending on the state of the boolean release parameter used to construct the object. Upon return, the caller must release the memory. On the server side, the ORB will allocate memory for the structure. The callee may free the old buffer and allocate a new buffer, depending on the state of the boolean release parameter used to construct the object. Once the data is returned to the client, the ORB releases the memory.
return	On the server side, the callee returns to the ORB a pointer to the sequence or any. The ORB will free the memory upon returning. The client cannot return a NULL pointer. The caller receives a pointer to the sequence or any. If the caller wishes to modify any of the values, it must make a copy of the object and modify the copy. The caller is responsible for releasing the returned object's memory.

Table 12-9 Memory management rules for sequences and any *arrays.*

Fixed Arrays

MODE	DESCRIPTION
in	The caller allocates the necessary storage for the array and is responsible for freeing it when finished. The callee receives the array from the ORB and cannot modify it. The memory associated with the array is freed by the ORB upon returning. The callee can preserve the array by copying it.
out	The caller allocates the necessary storage, but need not initialize it. Once the method returns, the storage will hold the array and the caller is responsible for freeing the memory when finished. On the server side, the ORB allocates the memory and the callee initializes the array. Once the array has been sent to the client, the ORB releases the memory.
inout	The caller allocates the necessary storage and initializes it. Upon return, the caller must release the memory. On the server side, the ORB will allocate memory for the array. The callee may the elements in the array. Once the array has been returned to the client, the ORB releases the memory.
return	On the server side, the callee returns a pointer to the array slice. The callee may not return a NULL pointer. The caller receives a pointer to the array slice, but may not modify it. If the caller wishes to modify any of the elements, it must make a copy of the array slice and modify the copy. The caller is responsible for releasing the returned array slice's memory.

Table 12-10 Memory management rules for fixed-length arrays.

Variable-Length Arrays

MODE	DESCRIPTION
in	The caller allocates the necessary storage for the array and initializes it. The caller is responsible for freeing the memory when finished. The callee receives the parameter from the ORB and cannot modify it. The memory associated with the parameter is freed by the ORB upon returning. The callee can preserve the array by copying it.
out	The caller allocates a pointer to an array slice and passes it by reference to the ORB. Once the method returns, the caller is responsible for freeing the memory when finished. On the server side, the ORB allocates a pointer to an array slice and passes it by reference to the callee. The callee sets the pointer to a valid instance of an array. Once the data is returned, the ORB will free the storage. The callee is not allowed to return a NULL pointer.
inout	The caller allocates the array and initializes it. Upon return, the caller must release the memory. On the server side, the ORB will allocate memory for the array. The callee may modify elements of the array. Once the data is returned to the client, the ORB releases the memory.
return	On the server side, the callee returns to the ORB a pointer to an array slice. The ORB will free the memory upon returning. The client cannot return a NULL pointer. The caller receives a pointer to the array slice. If the caller wishes to modify any of the elements, it must make a copy of the array and modify the copy. The caller is responsible for releasing the memory.

Table 12-11 Memory management rules for variable-length arrays.

T_VAR DATA TYPES

Table 12-2 summarizes the parameter passing mode for T_var data types.

DATA TYPE	IN	INOUT	OUT	RETURN
object ref var	const objref_var&	objref_var&	objref_var&‘	objref_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
sequence_var	const sequence_var&	sequence_var&	sequence_var&	sequence_var
array_var	const array_var&	array_var&	array_var&	array_var
any_var	const any_var&	any_var&	any_var&	any_var

Table 12-12 Parameter passing modes for T_var types.

Memory Management for T_var Types

MODE	DESCRIPTION
in	The caller allocates the necessary storage for the object and is responsible for freeing it when finished. The callee receives the object from the ORB and cannot modify it. The memory associated with the object is freed by the ORB upon returning. The callee can preserve the object by copying it or increasing the object's reference count.
out	The caller allocates a pointer and passes it by reference to the ORB. Once the method returns, the caller is responsible for freeing the memory when finished. On the server side, the ORB allocates a pointer and passes it by reference to the callee. The callee sets the pointer to a valid instance of the parameter's type. If the callee wishes to preserve the object, it is must make a copy or increase the object's reference count.
inout	The caller allocates the sequence or any and initializes it. Upon return, the caller must release the memory. On the server side, the ORB will allocate memory for the object. Once the data is returned to the client, the ORB releases the memory. If the callee wishes to preserve the object, it is must make a copy or increase the object's reference count.
return	On the server side, the callee returns to the ORB a pointer to the object. The ORB will free the memory upon returning. The client cannot return a NULL pointer. The caller receives a pointer to the object. The caller is responsible for releasing the returned object's memory.

Table 12-13 Memory management rules for T_var Types.

APPENDIX A

PLATFORMS WITHOUT C++ EXCEPTION SUPPORT

This appendix provides information about the Environment class. The Environment class is used when your compiler does not support exceptions through the `try` and `catch` statements.

FOR PLATFORMS WITHOUT C++ EXCEPTION SUPPORT

Not all C++ compilers support exceptions through the `try` and `catch` statements, so the CORBA specification defines an `Environment` class for reflecting exceptions. VisiBroker uses the Environment class, along with a set of macros, to provide your applications with exception handling capabilities when `try` and `catch` are not supported.

The Exception Macros

The `Environment` class is used internally by the ORB and is transparent to you as a programmer. The only requirement is that you use these exception macros to throw, try and catch exceptions. These macros will transparently manipulate the `Environment` class for you if your compiler does not support exceptions.

MACRO NAME	PURPOSE
PMCTRY	Use this as you would use the try statement.
PMCTHROW(type_name)	Throws the specified exception.
PMCTHROW_LAST	Used to re-throw the specified exception. Used only in an event handler or in a method called by an event handler.
PMCCATCH(type_name, variable_name)	Use this to catch an exception of the specified type.
PMCAND_CATCH	If several exceptions are to be specified for a PMCTRY block, use PMCCATCH for the first catch statement and PMCAND_CATCH for all subsequent catch statements.
PMCEND_CATCH	Used to terminate a PMCTRY block.

Figure A-1 The PMC exception macros.

Using the Exception Macros

You can modify the application client code shown in Figure 6-9 to use the compatibility macros. shows the modifications in bold type.

```
....
library *library_object;
PMCTRY {
library_object = library::_bind();
}
// Check for errors
PMCCATCH(CORBA::SystemException excep) {
cout << “System Exception occurred:” << endl;
cout << “exception name: “ <<
     excep._name() << endl;
cout << “minor code: “ <<
     excep.minor() << endl;
cout << “completion code: “ <<
     excep.completed() << endl;
}
PMCENDCATCH
...
```

Figure A-2 Using the compatibility macros to catch a system exception.

Object Implementation Considerations

The IDL compiler detects whether or not your **C++** compiler supports exceptions and generates code accordingly. The object implementation code shown in Figure 6-13 would appear as follows for a compiler without **C++** support. Note that the `throw` statement is not generated.

```
virtual CORBA::Boolean       add_book(const book& book_info);
```

The object’s implementation of `add_book` would use `PMCTHROW` to raise the exception.

```
CORBA::Boolean Library::add_book(const book& book_info)
{
     CORBA::Boolean ret;
     if( (ret = bk_list.add_to_list(book_info)) == 0 )
          PMCTHROW (library::CapacityExceeded();)
     return ret;
}
```

D

P