Rocket software

Orbix Programmer’s Guide C++ Edition

V3.3.17

Table of Contents

Preface

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

Audience

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

Document Conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`

Italic	Italic words in normal text represent emphasis and new terms. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/your_name` !!! note some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

This guide may use the following keying conventions:

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

Getting Started

Introduction to CORBA and Orbix

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

CORBA and Distributed Object Programming

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

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

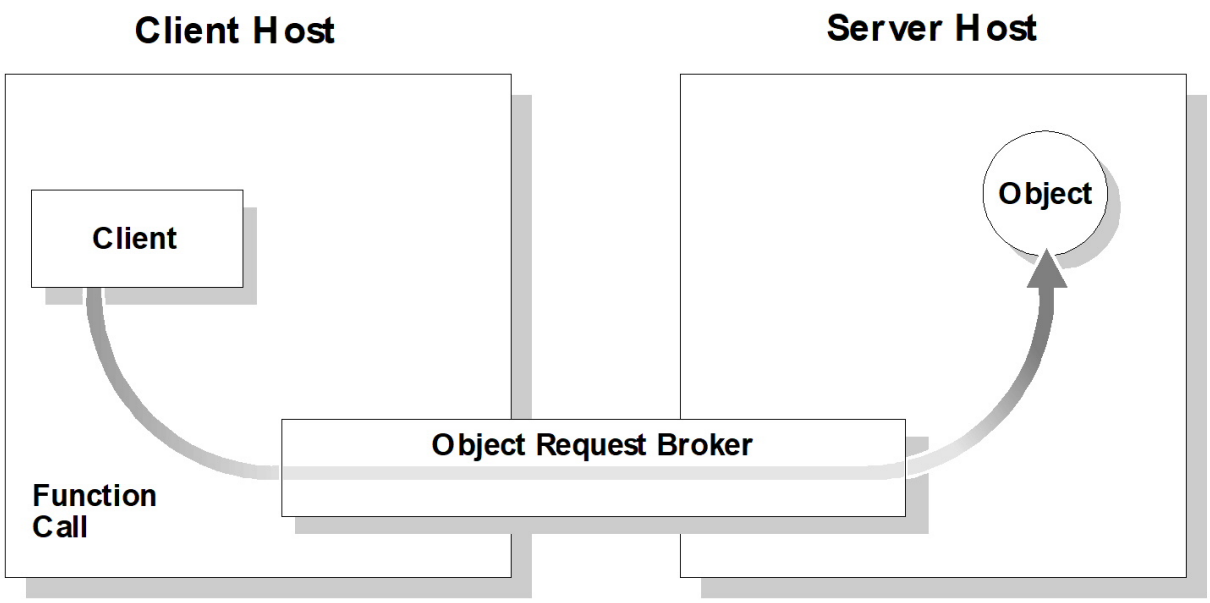
CORBA also brings the advantages of object-oriented techniques to a distributed environment. It allows you to design a distributed application as a set of cooperating objects and to re-use existing objects in new applications.

The Role of an Object Request Broker

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

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

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



The Nature of Objects in CORBA

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

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

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

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

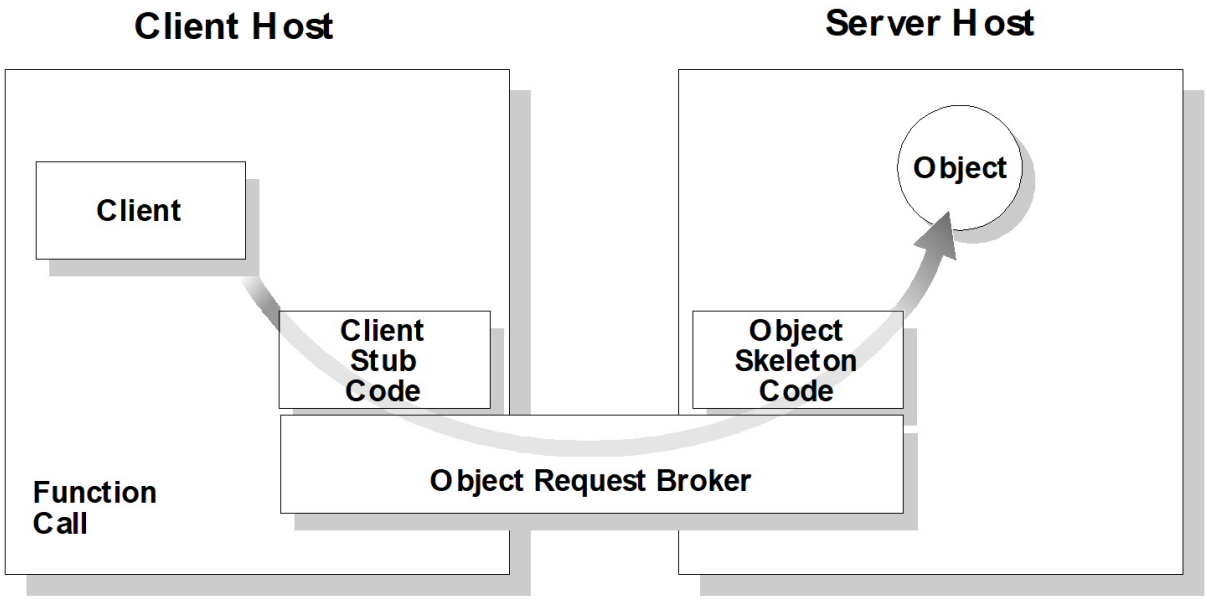
The separation between an object's interface and its implementation has several advantages. For example, it allows you to change the programming language in which an object is implemented without changing clients that access the object. It also allows you to make existing objects available across a network.

The Structure of a CORBA Application

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

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

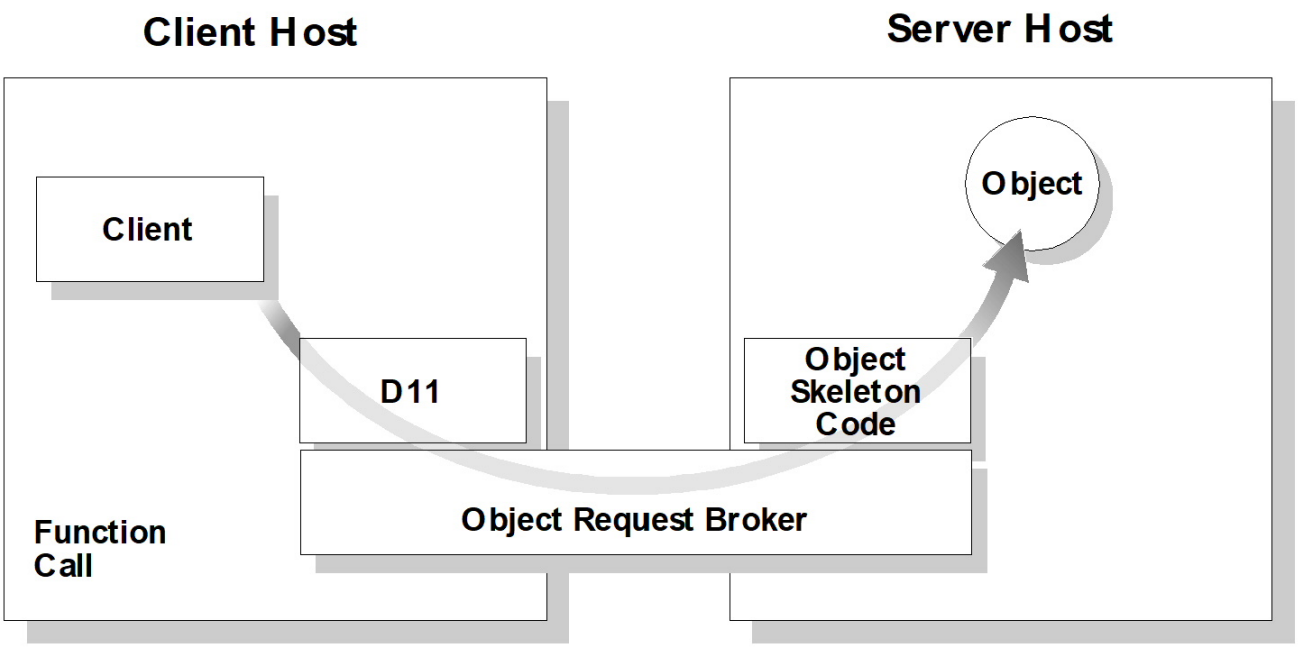
As shown in Figure 2, when a client calls a member function on a CORBA object, the call is transferred through the client stub code to the ORB. If the client has not accessed the object before, the ORB refers to a database, known as the *Implementation Repository*, to determine exactly which object should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



The Structure of a Dynamic CORBA Application

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

The CORBA .

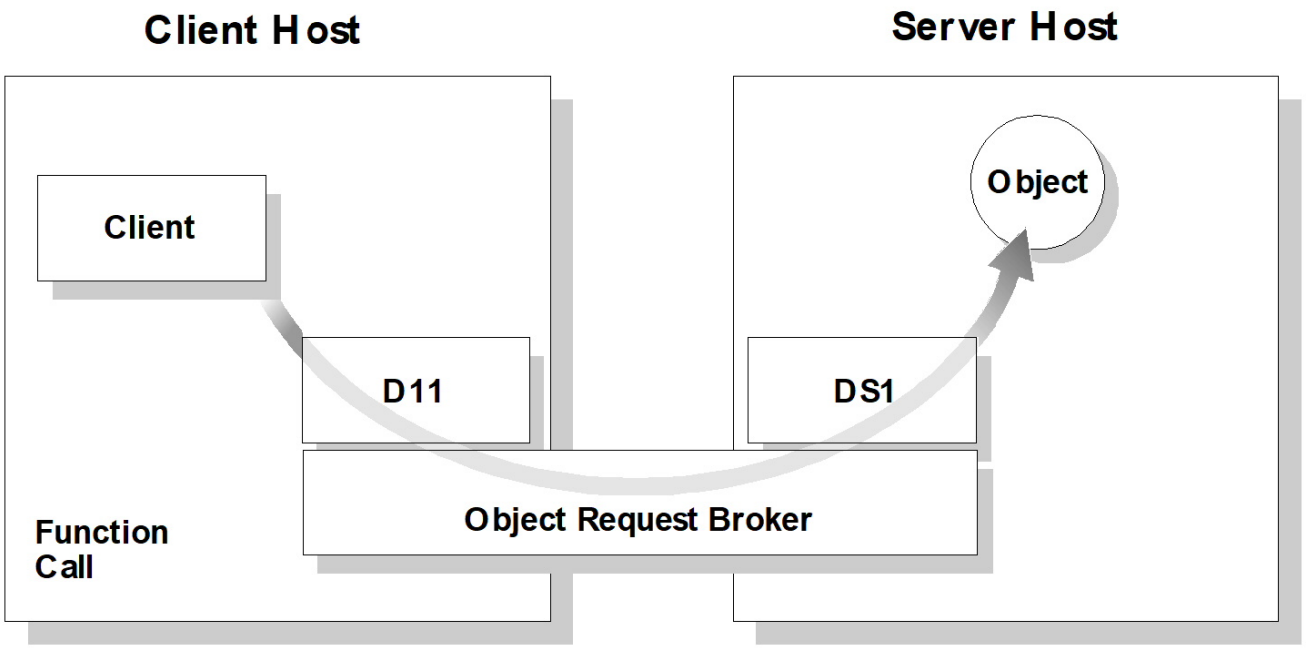


CORBA also supports dynamic server programming. A CORBA program can receive function calls through IDL interfaces for which no CORBA object exists. Using an ORB component called the *Dynamic Skeleton Interface* (DSI), the server can then examine the structure of these function calls and implement them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

Interoperability between Object Request Brokers

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

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



The Object Management Architecture

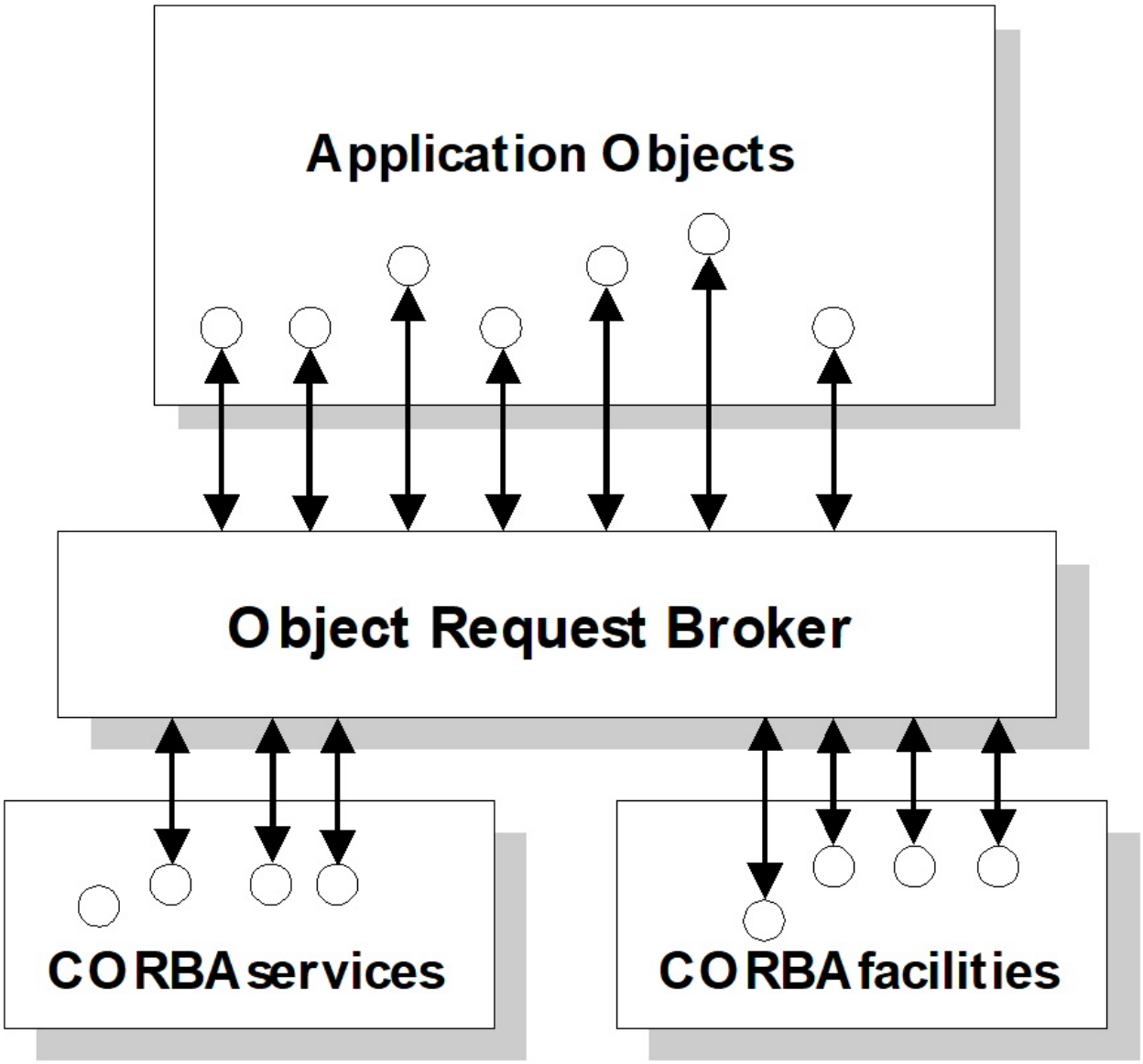
An ORB is one component of the OMG's Object Management Architecture (OMA). This architecture defines a framework for communications between distributed objects.

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

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

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

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



The

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

- The
- The *Trader Service*. This service allows a client to locate object references based on the desired properties of an object.
- The *Security Service*. This service allows CORBA programs to interact using secure communications.

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

The CORBAfacilities

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

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

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

How Orbix Implements CORBA

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

The components of Orbix are as follows:

- The
- The *Orbix*
- The *Orbix*
- The *Orbix Interface Repository server* is a process that implements the Interface Repository.

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

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

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

Orbix Components

Table 1 gives a brief synopsis of the Orbix suite.

Table 1 The Orbix Suite

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

Orbix Architecture

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

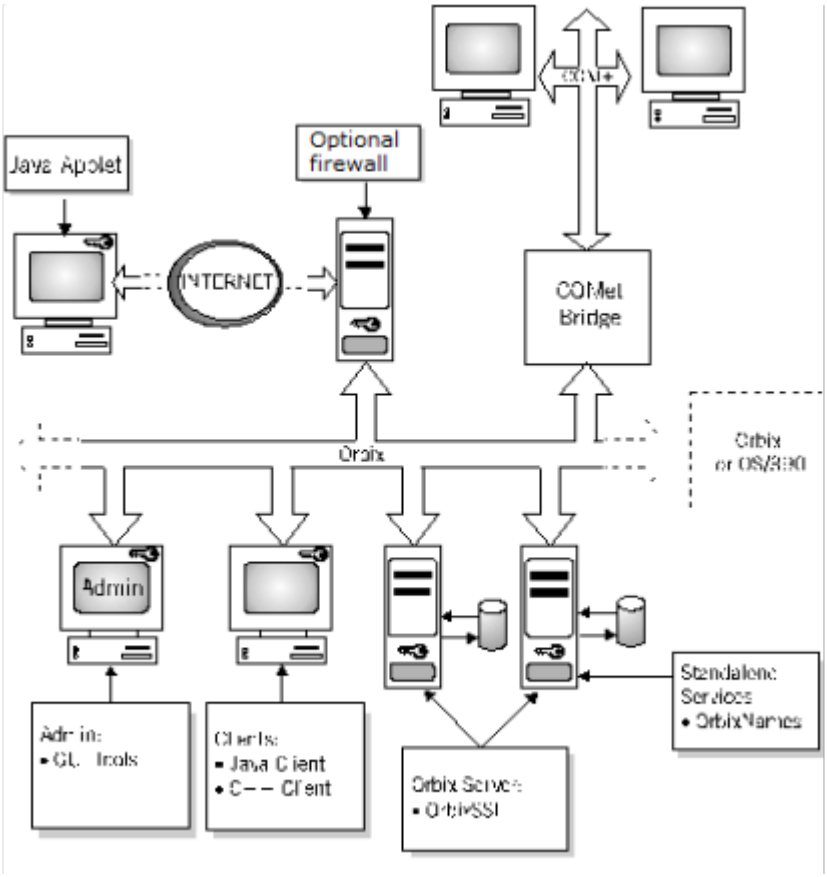
Some standard services, such as the CORBA Naming Service (OrbixNames) are implemented as clearly identifiable processes with an associated executable. There can be many instances of these processes running on one or more machines.

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

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

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

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



OrbixNames—The Naming Service

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

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

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

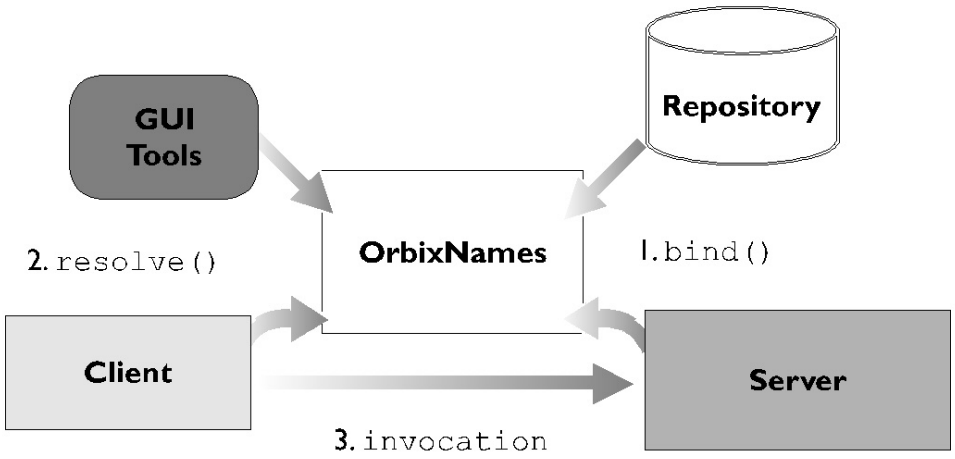


Figure 7 summarizes the functionality of OrbixNames, which is as follows:

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

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

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

Security with OrbixSSL

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

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

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

Authentication	Allows an application to verify the identity of another application with which it communicates.

Privacy	Ensures that data transmitted between applications can not be understood by a third party.
Integrity	Allows applications to detect whether data was modified during transmission.

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

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

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

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

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

Developing Applications with Orbix

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

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

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

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

Developing a Distributed Application

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

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

Defining IDL Interfaces

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

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

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

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

```
// IDL
// In file banksimple.idl
1          module BankSimple {
   typedef float CashAmount;
2          interface Account;
3             interface Bank {
         Account create_account (in string name);
         Account find_account (in string name);
      };
4             interface Account {
         readonly attribute string name;
         readonly attribute CashAmount balance;
5                void deposit (in CashAmount amount);
         void withdraw (in CashAmount amount);
      };
};
```

This code is explained as follows:

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

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

Compiling IDL Interfaces

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

Setting Up Configuration for the IDL Compiler

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

UNIX

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

1. Under `sh enter`:

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

2. or under `csh` enter:

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

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

Windows

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

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

Running the IDL Compiler

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

Windows and UNIX

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

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

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

Output from the IDL Compiler

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

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

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

By default, these files are named as follows:

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

The Client Stub Code

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

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

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

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

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

The Object Skeleton Code

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

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

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

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

Implementing the IDL Interfaces

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

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

```
// C++
// In file BankSimple\banksimple_bankimpl.h
// Implementation class for the Bank IDL interface.
...
1    class BankSimple_BankImpl : public virtual
                BankSimple::BankBOAImpl
{
   public:
      // Mapped IDL operations.
2       virtual BankSimple::Account_ptr
         create_account(const char*  name, CORBA::Environment&);
      virtual BankSimple::Account_ptr
         find_account( const char*  name, CORBA::Environment&);
      // C++ constructor and destructor.
3       BankSimple_BankImpl();
      virtual ~BankSimple_BankImpl();
   protected:
      static const int MAX_ACCOUNTS;
4       BankSimple::Account_var* m_accounts;
};
```

This code is explained as follows:

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

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

```
// C++
// In file banksimple_bankimpl.cxx
#include “banksimple_bankimpl.h”
#include “banksimple_accountimpl.h”
1    const int BankSimple_BankImpl::MAX_ACCOUNTS = 1000;
BankSimple_BankImpl::BankSimple_BankImpl() :
m_accounts(new BankSimple::Account_var[MAX_ACCOUNTS]) {
// Make sure all accounts are nil.
   for (int i = 0; i < MAX_ACCOUNTS; ++i){
      m_accounts[i] = BankSimple::Account::_nil();
   }
}
BankSimple_BankImpl::~BankSimple_BankImpl() {
   delete [] m_accounts;
}
// Add a new account.
BankSimple::Account_ptr BankSimple_BankImpl::create_account
(const char*  name, CORBA::Environment& ) {
   int i = 0;
   for ( ; i < MAX_ACCOUNTS && !CORBA::is_nil(m_accounts[i]);
             ++i)
   {}
   if (i < MAX_ACCOUNTS){
2       m_accounts[i] = new BankSimple_AccountImpl(name, 0.0);
      cout << “create_account: Created account with name: ”
           << name << endl;
3       return BankSimple::Account::_duplicate(m_accounts[i]);
   }
   else{
      cout << “create_account: failed, no space left!” << endl;
4       return BankSimple::Account::_nil();
   }
}
// Find a named account.
BankSimple::Account_ptr BankSimple_BankImpl::find_account
(const char*  name, CORBA::Environment& ) {
   int i = 0;
   for (  ; i < MAX_ACCOUNTS &&( CORBA::is_nil(m_accounts[i]) ||
         strcmp(name, m_accounts[i]->name()) != 0); ++i)
   { }
   if (i < MAX_ACCOUNTS){
      cout << “find_account: found account named” << name << endl;
      return BankSimple::Account::_duplicate(m_accounts[i]);
   }
   else{
      cout << “find_account: no account named” << name << endl;
```

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

The code is explained as follows:

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

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

Writing an Orbix Server Application

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

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

This section describes each of these programming steps in turn.

Initializing the ORB

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

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

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

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

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

Creating an Implementation Object

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

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

```
// C++
// In file server.cxx
#include “banksimple_bankimpl.h”
int main ( ... ) {
...
// Create a bank implementation object.
BankSimple::Bank_var my_bank = new BankSimple_BankImpl;
   ...
}
```

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

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

Receiving Client Requests

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

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

```
// C++
// In file server.cxx
#include “banksimple_bankImpl.h”
#include “banksimple_accountImpl.h”
#include <it_demo_nsw.h>
// Server mainline.
int main (int argc, char*  argv[]) {
   try {
      // Use standard demo server options.
1       IT_Demo_ServerOptions
                  serveropt(“IT_Demo/BankSimple/Bank”);
      ...
2       CORBA::ORB_var orb = CORBA::ORB_init(argc, argv,
       “Orbix”);
      CORBA::BOA_var boa = orb->BOA_init(argc, argv, “Orbix_BOA”);
      // Set diagnostics.
      orb->setDiagnostics(serveropt.diagnostics());
      // Set server name.
3       orb->setServerName(serveropt.server_name());
4       // Indicate server should not quit while clients
      // are connected.
      boa->setNoHangup(1);
      // Set up Naming Service Wrappers (NSW).
5       IT_Demo_NSW ns_wrapper;
6       ns_wrapper.setNamePrefix(serveropt.context());
7       const char* bank_name = “BankSimple.Bank”;
      ...
      // Create a bank implementation object.
8       BankSimple::Bank_var my_bank = new BankSimple_BankImpl;
9       // Register server object with the Naming Service.
      if (serveropt.bindns()) {
         cout << “Binding objects in the Naming Service”
          << endl;
         ns_wrapper.registerObject(bank_name, my_bank);
      }
      // Server has completed initialization, wait for
      // incoming requests.
10       boa->impl_is_ready( (char*)serveropt.server_name(),
           serveropt.timeout());
      // impl_is_ready() returns only when Orbix times-out
      // an idle server.
      cout << “server exiting” << endl;
   }
   catch (const CORBA::Exception& e) {
      cerr << “Unexpected exception” << e << endl;
      return 1;
```

```
    }
    return 0;
};
```

This code is explained as follows:

1. Create the standard server options for use throughout the demonstration and set the server name to IT_Demo/BankSimple/Bank. The Orbix `demos\demolib` directory contains the standard server and client options used by the Bank series examples in this book.
2. Initialize the ORB and BOA. The ORB object provides functionality common to both clients and servers. The BOA (Basic Object Adapter) object is derived from the ORB and provides additional server-side functionality.
3. The ORB and the BOA are different views of the same ORB API—this object is also available via the global variable `CORBA::Orbix`. However, use of this variable is not CORBA-defined and is discouraged.
4. Set the server name using `setServerName(serveropt.server_name())`. This is required by Orbix before exporting object references.
5. Create a *Naming Service Wrapper* (NSW) object. To simplify the use of the Naming Service, a Naming Service Wrapper is provided. This hides the low-level detail of the CORBA Naming Service. Refer to Using the Naming Service in Orbix Example Applications for details of the Naming Service wrapper functions.
6. Define a name prefix that is used for subsequent operations.
7. `BankSimple.Bank` is the name that the bank object is known by in the Naming Service.
8. The created `BankSimple` instance is `my_bank`. This object implements an instance of the IDL interface `Bank`. This is called directly from client applications using the CORBA standard Internet Inter-ORB Protocol (IIOP).
9. The server now registers its objects in the Naming Service using the Naming Service wrapper function `registerObject()`.
10. The `CORBA::BOA::impl_is_ready()` operation is called to complete server initialization. This takes a server name and a timeout value as parameters. You can specify any name for your server; however, the name should match the name used to register the server in the Implementation Repository, and the argument used to call `setServerName()`.
11. The timeout value indicates the period of time, in milliseconds, that the `impl_is_ready()` call should block for while waiting for an operation call to arrive from a client. If no call arrives in this period, `impl_is_ready()` returns. If a call arrives, Orbix calls the appropriate member function on the implementation object and the timeout counter starts again from zero.

Writing an Orbix Client Application

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

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

This section describes each of these steps in turn.

Initializing the ORB

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

CORBA Object References

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

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

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

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

Getting a Reference to an Object

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

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

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

```
   // C++
   // In file client.cxx
   ...
   // Naming Service Setup.
   // Create a Naming Service Wrapper object.
      IT_Demo_NSW ns_wrapper;
1       ns_wrapper.setNamePrefix(clientopt.context());
   // Get CORBA object.
   // Specify the object name in the Naming Service.
2       const char* object_name = "BankSimple.Bank";
   // Get a reference to the required object from the NSW.
3     CORBA::Object_var obj = ns_wrapper.resolveName(object_name);
   // Narrow the object reference.
4     BankSimple::Bank_var bank = BankSimple::Bank::_narrow(obj);
       if (CORBA::is_nil(bnk)) {
       cerr << "Object \"" << object_name
            << "\"in the Naming Service" << endl
            << "\tis not of the expected type."<< endl;
      return 1;
   }
   // Start client menu loop
5       BankMenu main_menu(bank);
         main_menu.start);
   }
   ...
}
```

This code is described as follows:

1. Define a name prefix used by the Naming Service wrapper object for subsequent operations.
2. `BankSimple.Bank` is the name by which the bank object is known in the Naming Service.

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

Invoking IDL Attributes and Operations

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

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

```
// C++
// In file bankmenu.cxx
void BankMenu::do_create() throw(CORBA::SystemException) {
   cout << “Enter account name: ” << flush;
   CORBA::String_var name = IT_Demo_Menu::get_string();
1       BankSimple::Account_var account =
m_bank->create_account(name);
   // Start a sub-menu with the returned account ref.
   AccountMenu sub_menu(account);
   sub_menu.start();
}
// do_find -- calls find account and runs account menu.
void BankMenu::do_find throw (CORBA::SystemException) {
   cout << “Enter account name: “ << flush;
2    CORBA::String_var name = IT_Demo_Menu::get_string();
   BankSimple::Account_var account = m_bank->find_account(name);
      AccountMenu sub_menu(account)
   sub_menu.start();
}
```

This code is explained as follows:

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

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

Compiling the Client and Server

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

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

Compiling the Client

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

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

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

Compiling the Server

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

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

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

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

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

Windows	`>nmake`
UNIX	`%make`

Running the Application

To run the application, do the following:

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

Running the Orbix Daemon

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

Windows and UNIX

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

Registering the Server

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

Windows and UNIX and OpenVMS

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

```
putit server_name server_executable
```

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

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

To register the server, type one of the following:

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

Running the Client

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

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

Windows and UNIX

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

Summary of Programming Steps

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

1. Identify the objects required in your system and define the public interfaces to those objects using the CORBA Interface Definition Language (IDL).
2. Compile the IDL interfaces.
3. Implement the IDL interfaces with C++ classes.
4. Write a server program that creates instances of the implementation classes. This involves:

   Initializing the ORB.

Creating initial implementation objects.

Allowing Orbix to receive and process incoming requests from clients.

5. Write a client program that accesses the server objects. This involves:

   Initializing the ORB.

   Getting a reference to an object.

   Invoking object attributes and operations.

6. Compile the client and server.
7. Run the application. This involves:

   Running the Orbix daemon process.

   Registering the server in the Implementation Repository.

   Running the client.

Orbix C++ Programming

Introduction to CORBA IDL

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

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

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

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

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

IDL Modules and Scoping

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

The following example illustrates the use of modules in IDL:

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

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

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

Defining IDL Interfaces

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

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

```
//IDL
module BankSimple {
   // Define a named type to represent money.
   typedef float CashAmount;
         // Forward declaration of interface Account.
   interface Account;
   interface Bank {
      ...
   };
   interface Account {
      // The account owner and balance.
      readonly attribute string name;
      readonly attribute CashAmount balance;
      // Operations available on the account.
      void deposit (in CashAmount amount);
      void withdraw (in CashAmount amount);
   };
};
```

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

Attributes in IDL Interface Definitions

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

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

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

Operations in IDL Interface Definitions

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

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

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

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

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

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

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

Raising Exceptions in IDL Operations

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

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

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

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

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

```
// IDL
module BankExceptions {
   typedef float CashAmount;
   ...
   interface Account {
      exception InsufficientFunds {
         string reason;
      };
      void withdraw(in CashAmount amount)
         raises(InsufficientFunds);
      ...
   };
};
```

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

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

Invocation Semantics for IDL Operations

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

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

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

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

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

Passing Context Information to IDL Operations

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

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

```
// IDL
module BankSimple {
   typedef float CashAmount;
   ...
   interface Account {
      void deposit(in CashAmount amount)
         context("sys_time", "sys_location");
      ...
   };
};
```

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

Note that IDL contexts are rarely used in practice.

Inheritance of IDL Interfaces

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

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

```
// IDL
module BankSimple{
   interface Account {
      ...
   };
   interface CheckingAccount : Account {
      readonly attribute overdraftLimit;
      boolean orderChequeBook ();
   };
   interface SavingsAccount : Account {
      float calculateInterest ();
   };
};
```

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

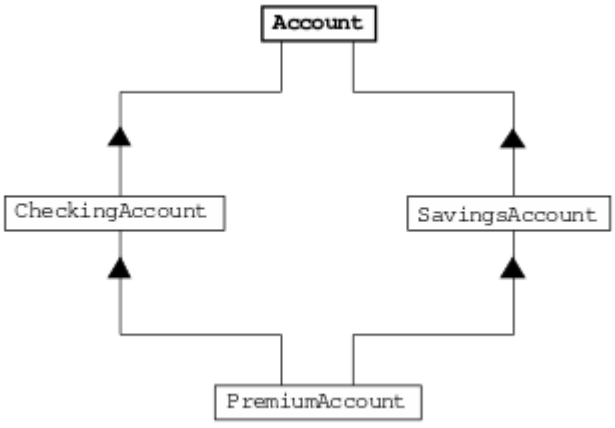
An object that implements `CheckingAccount` can accept invocations on any of the attributes and operations of this interface, and on any of the elements of interface `Account`. However, a `CheckingAccount` object may provide different implementations of the elements of interface `Account` to an object that implements `Account` only.

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

```
// IDL
module BankSimple {
   interface Account {
      ...
   };
   interface CheckingAccount : Account {
      ...
   };
   interface SavingsAccount : Account {
      ...
   };
   interface PremiumAccount :
      CheckingAccount, SavingsAccount {
      ...
   };
};
```

Interface `PremiumAccount` is an example of multiple inheritance in IDL. Figure 8 on page 36 illustrates the inheritance hierarchy for this interface.

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



The Object Interface Type

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

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

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

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

Forward Declaration of IDL Interfaces

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

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

```
// IDL
module BankSimple {
   // Forward declaration of Account.
   interface Account;
   interface Bank {
      Account create_account (in string name);
      Account find_account (in string name);
   };
   // Full definition of Account.
   interface Account {
      ...
   };
};
```

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

Overview of the IDL Data Types

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

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

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

IDL Basic Types

The following table lists the basic types supported in IDL.

IDL Type	Range of Values
`short`	`-2``15``...2``15``-1 (16-bit)`
`unsigned short`	`0...2``16``-1 (16-bit)`
`long`	`-2``31``...2``31``-1 (32-bit)`
`unsigned long`	`0...2``32``-1 (32-bit)`
`long long`	`-2``63``...2``63``-1 (64-bit)`
`unsigned long long`	`0...-2``64 (64-bit)`
`float`	IEEE single-precision floating point numbers.
`double`	IEEE double-precision floating point numbers.
`char`	An 8-bit value.
`boolean`	`TRUE` or `FALSE`.
`octet`	An 8-bit value that is guaranteed not to undergo any conversion during transmission.

`any`	The `any` type allows the specification of values that can express an arbitrary IDL type.

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

IDL Complex Types

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

Enum

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

```
// IDL
module BankSimple {
   enum Currency {pound, dollar, yen, franc};
   interface Account {
      readonly attribute CashAmount balance;
      readonly attribute Currency
          balanceCurrency;
      ...
   };
};
```

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

Struct

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

```
// IDL
module BankSimple{
      struct CustomerDetails {
         string name;
         short age;
      };
      interface Bank {
         CustomerDetails getCustomerDetails
            (in string name);
         ...
      };
};
```

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

Union

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

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

For example, consider the following IDL union definition:

```
// IDL
struct DateStructure {
   short Day;
   short Month;
   short Year;
};
union Date switch (short) {
   case 1: string stringFormat;
   case 2: long digitalFormat;
   default: DateStructure structFormat;
};
```

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

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

String

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

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

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

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

Sequence

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

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

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

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

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

```
// IDL
module BankSimple {
   typedef sequence<string> CustomerSeq;
   interface Account {
      void getCustomerList(out CustomerSeq names);
      ...
   };
};
```

Arrays

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

```
// IDL
module BankSimple {
   ...
   interface Account {
      ...
   };
   struct CustomerAccountInfo {
      string name;
      Account accounts[3];
   };
   interface Bank {
      getCustomerAccountInfo (in string name,
         out CustomerAccountInfo accounts);
      ...
   };
};
```

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

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

The following code illustrates this:

```
// IDL
module BankSimple {
   interface Account {
      ...
   };
   typedef Account AccountArray[100];
   interface Bank {
      readonly attribute AccountArray accounts;
      ...
   };
};
```

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

Fixed

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

```
module BankSimple {
         typedef fixed<10,4> ExchangeRate;
         struct Rates {
                  ExchangeRate USRate;
                  ExchangeRate UKRate;
                  ExchangeRate IRRate;
         };
};
```

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

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

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

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

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

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

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

IDL Pseudo Object Types

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

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

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

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

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

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

Defining Data Type Names and Constants

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

Data Type Names

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

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

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

Constants

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

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

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

The CORBA IDL to C++ Mapping

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

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

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

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

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

Overview of the Mapping

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

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

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

Mapping for Modules and Scoping

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

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

This maps to:

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

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

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

Alternative Mappings for Modules

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

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

This maps to:

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

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

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

Mapping for Interfaces

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

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

Consider a simple interface to describe a bank account:

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

This maps to the following IDL C++ class:

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

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

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

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

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

A `_ptr` type is more primitive and has similar semantics to a C++ pointer. In fact, `_ptr` types in Orbix are currently implemented as C++ pointers. However, it is important that you do not use this knowledge because this implementation may change. For example, you should not attempt conversion to `void*`, arithmetic operations and relational operations, including test for equality on `_ptr` types.

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

Mapping for Attributes

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

Consider the following IDL interfaces:

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

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

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

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

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

```
// C++
Account_var aVar;
CORBA::Float bal = 0;
CORBA::Long number = 99;
// Code to bind aVar to an Account object omitted.
...
try {
   // Get value of balance.
   bal = aVar->balance();
   // Set and get value of accountNumber.
   aVar->accountnumber(number);
   number = aVar->accountnumber();
}
catch (const CORBA::SystemException& se) {
   ...
}
```

Mapping for Operations

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

Consider the following IDL interfaces:

```
// IDL
typedef float CashAmount;
....
interface Account {
   void deposit(in CashAmount amount);
   void withdraw(in CashAmount amount);
   ...
};
interface Bank {
   Account create_account(in string name);
};
```

The following code illustrates the mapping for IDL operations:

```
// C++
class Account : public virtual CORBA::Object {
public:
   virtual void deposit(CashAmount amount);
   virtual void withdraw(CashAmount amount);
   ...
};
class Bank : public virtual CORBA::Object {
public:
   virtual Account_ptr create_account
      (const char* name);
};
```

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

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

```
// C++
Account_var aVar;
Bank_var bVar;
// Code to bind bVar to a Bank object omitted.
...
try {
   aVar = bVar->create_account("Chris");
   aVar->deposit(100.00);
}
catch (const CORBA::SystemException& se) {
   ...
}
```

Memory Management for Parameters provides more information about the mapping for operation parameters.

Mapping for Exceptions

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

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

This maps to the following C++:

```
// C++
class CannotCreate : public CORBA::UserException {
public:
   CORBA::String_mgr reason;
   CORBA::Short s;
   CannotCreate(const char* _reason,
            const CORBA::Short& _s);
   CannotCreate();
   CannotCreate(const CannotCreate&);
   ~CannotCreate();
   CannotCreate()& operator = (const
         CannotCreate&);
      static CannotCreate*
         _narrow(CORBA::Exception* e);
};
```

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

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

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

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

Mapping for Contexts

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

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

This interface maps to:

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

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

Mapping for Inheritance of IDL Interfaces

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

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

The corresponding C++ is:

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

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

```
// C++
CheckingAccount_var checkingAc;
// Code for binding checkingAc omitted.
...
checkingAc->deposit(90.97);
```

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

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

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

Widening Object References

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

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

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

Some widening examples are shown in the code below:

```
// C++
CheckingAccount_ptr cPtr = ....;
// Implicit widening:
Account_ptr aPtr = cPtr;
// Implicit widening:
Object_ptr objPtr = cPtr;
// Implicit widening:
objPtr = aPtr;
CheckingAccount_var cVar = cPtr;
// cVar assumes ownership of cPtr.
aPtr = cVar;
      // Implicit widening, cVar retains ownership of cPtr.
objPtr = cVar;
      // Implicit widening, cVar retains ownership of cPtr.
Account_var av = cVar;
      // Illegal, compile-time error, cannot assign
      // between _var variables of different types.
      Account_var aVar = CheckingAccount::_duplicate(cVar);
      // aVar and cVar both refer to cPtr.
      // The reference count of cPtr is incremented.
```

Narrowing Object References

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

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

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

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

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

```
// C++
Account_ptr aPtr;
CheckingAccount_ptr caPtr;
// Code to bind aPtr to an object that implements
// CheckingAccount omitted.
...
// Narrow aPtr to be a CheckingAccount.
if (caPtr = CheckingAccount::_narrow(aPtr))
   ...
else
   // Deal with failure of _narrow().
```

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

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

Object Reference Counts and Nil Object References

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

Object Reference Counts

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

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

For example, consider the following server code:

```
// C++
// Create a new Bank object:
Bank_ptr bPtr = new Bank_i;
// The reference count of the new object is 1.
Bank::_duplicate(bPtr);
// The reference count of the object is 2.
CORBA::release(bPtr);
// The reference count of the object is 1.
```

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

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

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

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

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

You can call this function as follows:

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

Nil Object References

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

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

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

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

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

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

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

The following call is guaranteed to be true:

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

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

Mapping for IDL Data Types

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

Mapping for Basic Types

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

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

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

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

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

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

Mapping for Complex Types

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

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

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

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

Mapping for Enum

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

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

This maps to:

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

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

Mapping for Struct

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

Fixed Length Structs

Consider the following IDL fixed length struct:

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

This maps to:

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

Variable Length structs

Consider the following IDL variable length struct:

```
// IDL
interface A {
   ...
};
struct VariableLengthStruct {
   short i;
   float f;
   string str;
   A a;
};
```

This maps to a C++ struct as follows:

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

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

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

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

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

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

Mapping for Union

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

```
// IDL
typedef long vector[100];
struct S { ... };
interface A;
union U switch(long) {
   case 1: float f;
   case 2: vector v;
   case 3: string s;
   case 4: S st;
   default: A obj;
};
```

This maps to the following C++ struct:

```
// C++
struct U {
public:
   // The discriminant.
   CORBA::Long _d() const;                                          (1)
   // Constructors, Destructor, and Assignment.
   U();                                          (2)
   U(const CORBA::Long);                                      (2a)
   U(const U&);                                        (3)
   ~U();                                        (4)
   U& operator = (const U&);                                          (5)
   // Accessor and modifier functions for members.
   // Basic type member:
   CORBA::Float f() const;                                          (6)
   void f(CORBA::Float IT_member);                                          (7)
   // Array member:
   vector_slice* v() const;                                          (8)
   void v(vector_slice* IT_member);
(9)
   // String member:
   const char* s() const;                                       (10)
   void s(char* IT_member);                                       (11)
   void s(CORBA::String_var IT_member);
(12)
   void s(const char* IT_member);                                          (13)
   // Struct member:
   S& st();                                       (14)
   const S& st() const;                                      (15)
   void st(const S& IT_member);                                          (16)
   // Object reference member:
   A_ptr obj() const;                                     (17)
   void obj(A_ptr IT_member);                                          (18)
   ...
};
```

The Discriminant

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

Constructors, Destructor and Assignment

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

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

Accessors and Modifiers

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

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

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

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

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

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

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

Note

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

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

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

Example Program

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

```
// C++
U* u;
u = new U;
u->f(19.2);
// And later:
switch (u->_d()) {
   case 1 : cout << “f = ” << u->f()
                 << endl; break;
   case 2 : cout << “v = ” << u->v()
                 << endl; break;
   case 3 : cout << “s = ” << u->s()
                 << endl; break;
   // Do not free the returned string.
   case 4 : cout << “st = ” << “x = ” << u->st().x
                  << “ ” << “y = ” << u->st().y
                 << endl; break;
   default: cout << “A = ” << u->obj() << endl; break;
   // Do not release the returned object
   // reference.
}
```

Mapping for String

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

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

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

Consider the following IDL:

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

The corresponding C++ is:

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

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

```
// C++
stringTen s1 = 0;
stringInf s2 = 0;
// Or using the _var type:
CORBA::stringTen_var sv1;
CORBA::stringInf_var sv2;
```

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

Dynamic Allocation of Strings

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

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

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

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

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

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

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

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

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

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

Bounds Checking of String Parameters

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

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

General Mapping for Sequences

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

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

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

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

Mapping for Unbounded Sequences

Consider the following IDL declaration:

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

The IDL compiler generates the following class definition:

```
// C++
class unbounded {
public:
   unbounded();                                          (1)
   unbounded(const unbounded&);                                    (2)
   // This constructor uses existing space.
   unbounded(                                          (3)
      CORBA::ULong max,
      CORBA::ULong length,
      CORBA::Long* data,
      CORBA::Boolean release = 0);
   // This constructor allocates space.
   unbounded(CORBA::ULong max);                                    (4)
   ~unbounded();                                         (5)
   unbounded& operator = (const
unbounded&);                                          (6)
   static CORBA::Long* allocbuf(                                    (7)
      CORBA::ULong nelems);
   static void freebuf(CORBA::Long*
data);                                          (8)
   CORBA::ULong maximum() const;                                    (9)
   CORBA::ULong length() const;                                  (10)
   void length(CORBA::ULong len);                                  (11)
   CORBA::Long& operator[](                                   (12)
      CORBA::ULong IT_i);
   const CORBA::Long& operator[](                                  (13)
      CORBA::ULong IT_i) const;
};
```

Constructors, Destructor and Assignment

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

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

Constructor (3) allows the buffer space for a sequence to be allocated externally to the definition of the sequence itself. Normally sequences manage their own memory. However, this constructor allows ownership of the buffer to be determined by the `release` parameter: `0` (false) means the caller owns the storage, while `1` (true) means that the sequence assumes ownership of the storage. If `release` is `true`, the buffer must have been allocated using the sequence `allocbuf()` function, and the sequence passes it to `freebuf()` when finished with it. In general, constructor (3) particularly with the `release` parameter set to `0`, should be used with caution and only when absolutely necessary.

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

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

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

Sequence Buffer Management: allocbuf() and freebuf()

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

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

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

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

Other Functions

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

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

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

Unbounded Sequences Example

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

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

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

- Using the default constructor:

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

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

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

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

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

- Using the copy constructor:

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

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

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

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

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

```
// C++
unbounded_var uVar = new unbounded;
uVar->length(10);
CORBA::Long i;
for (i = 0; i<10; i++)
   uVar[i] = i;
...
// Do not call ‘delete uVar’.
```

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

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

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

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

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

Mapping for Bounded Sequences

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

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

The corresponding C++ code is as follows:

```
// C++
class bounded {
public:
   bounded();                                          (1)
   bounded(const bounded&);                                    (2)
   bounded(CORBA::ULong length,                                    (3)
      CORBA::Long* data,
      CORBA::Boolean release = 0);
   ~bounded();                                          (4)
   bounded& operator = (const
bounded&);                                        (5)
   static CORBA::Long* allocbuf(                                    (6)
      CORBA::ULong nelems);
   static void freebuf(CORBA::Long*
data);                                      (7)
   CORBA::ULong maximum() const;                                    (8)
   CORBA::ULong length() const;                                    (9)
   void length(CORBA::ULong len);                                    (10)
   CORBA::Long& operator[](                                  (11)
      CORBA::ULong IT_i);
   const CORBA::Long& operator[](                                    (12)
      CORBA::ULong IT_i) const;
};
```

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

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

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

Bounded Sequence Examples

Consider the following IDL declaration:

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

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

- Using the default constructor:

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

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

- Using the copy constructor:

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

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

- By dynamically allocating the sequence:

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

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

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

- Using constructor (3) as follows:

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

As for unbounded sequences, avoid this form of sequence construction whenever possible. In this example, the `release` parameter is set to `1` (true) to indicate that sequence `z` is to responsible for releasing the buffer, `data`.

Mapping for Fixed

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

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

The fixed template class is defined as follows:

```
   template<unsigned short d, short s> class CORBA_Fixed
{
   public:
      CORBA_Fixed(const int val = 0);
      CORBA_Fixed(const long double val);
      CORBA_Fixed(const CORBA_Fixed<d, s>& val);
      ~CORBA_Fixed();
   operator CORBA_Fixed<d, s> () const;
   operator double() const;
   CORBA_Fixed<d, s>& operator= (const CORBA_Fixed<d, s>& val);
   CORBA_Fixed<d, s>& operator++();
   const CORBA_Fixed<d, s> operator++(int);
   CORBA_Fixed<d, s>& operator--();
   const CORBA_Fixed<d, s> operator--(int);
   CORBA_Fixed<d, s>& operator+() const;
   CORBA_Fixed<d, s>& operator-() const;
   int operator!() const;
   CORBA_Fixed<d, s>& operator+= (const CORBA_Fixed<d, s>& val1);
   CORBA_Fixed<d, s>& operator-= (const CORBA_Fixed<d, s>& val1);
   CORBA_Fixed<d, s>& operator*= (const CORBA_Fixed<d, s>& val1);
   CORBA_Fixed<d, s>& operator/= (const CORBA_Fixed<d, s>& val1);
   const unsigned short Fixed_Digits() const;
   const short Fixed_Scale() const;
```

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

```
// C++
double rate = 1.4234;
ExchangeRate USRate(rate);
USRate + = 0.1;
cout << “US Exchange Rate = ” << USRate << endl;
   // outputs 0001.523400
```

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

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

Streaming Operators

The streaming operators for fixed are as follows:

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

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

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

Arithmetic Operators

The arithmethic operators for fixed are as follows:

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

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

```
   // C++
   ExchangeRate USRate(1.453);
   ExchangeRate UKRate(0.84);
   ExchangeRate diff;
   diff = USRate - UKRate;
   cout << “difference between US rate and UK rate is ”
         << diff << endl;
         // outputs 0000.613000;
```

Logical Operators

The logical operators for fixed are as follows:

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

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

```
// C++
ExchangeRate USRate(1.453);
ExchangeRate UKRate(0.84);
if (USRate<= UKRate)
   {
      // Do stuff...
   };
```

Mapping for Array

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

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

Arrays as Out Parameters and Return Values

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

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

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

This generates the following array and array slice typedefs:

```
// C++
typedef long arrayLong[10];
typedef long arrayLong_slice;
typedef float arrayFloat[5][3];
typedef float arrayFloat_slice[3];
```

Dynamic Allocation of Arrays

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

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

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

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

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

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

Mapping for Typedef

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

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

This generates the following C++ typedef:

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

Mapping for Constants

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

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

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

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

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

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

This maps to the following C++:

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

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

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

Mapping for Pseudo-Object Types

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

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

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

Memory Management and _var Types

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

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

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

  For example, consider the following interface definition:

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

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

```
// C++
{
   // Set aPtr to refer to an object:
   A_ptr aPtr = ...
   A_var aVar = aPtr;
   // Here, aVar assumes ownership of aPtr.
   // The object reference is not duplicated.
   aVar->op();
   ...
}
   // Here, aVar is released (its
   // reference count decremented).
```

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

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

Constructors and Destructor

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

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

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

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

The following code illustrates some of these points:

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

Assignment Operators

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

```
// C++
// Set aVar to refer to an object reference.
A_var aVar = ...
// Set aPtr to refer to an object reference.
A_ptr aPtr = ...
// The following assignment causes the _ptr
// owned by aVar to be released before aVar
// assumes ownership of aPtr.
aVar = aPtr;
```

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

```
// C++
{
   T_var t1Var = ...
   T_var t2Var = ...
   // The following assignment frees t1Var and
   // deep copies t2Var, duplicating its
   // object reference.
   t1Var = t2Var;
}
// Here, t1Var and t2Var are released. They both /// refer to the same object
so the reference count
// of the object is decremented twice.
```

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

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

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

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

operator->()

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

For example:

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

The following are some examples of illegal code:

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

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

Memory Management for Parameters

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

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

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

in Parameters

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

For example, consider the following IDL operation:

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

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

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

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

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

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

Note

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

inout Parameters

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

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

Object Reference inout Parameters

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

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

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

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

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

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

String inout Parameters

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

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

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

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

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

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

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

Sequence inout Parameters

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

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

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

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

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

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

Type any inout Parameters

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

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

Other inout Parameters

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

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

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

out Parameters

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

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

For example, consider the following IDL:

```
// IDL
struct VariableLengthStruct {
   string aString;
};
struct FixedLengthStruct {
   float aFloat;
};
interface A {
   void opOut(out float f,
      out FixedLengthStruct fs,
      out VariableLengthStruct vs);
};
```

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

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

A client calls this operation as follows:

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

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

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

```
// C++
{
   //You must allocate memory for a fixed
   //length struct
   FixedLengthStruct fs;
   //No need to initialize memory for a variable
   //length struct
   VariableLengthStruct* vs_p;
   aVar->opOut(fs, vs_p)
   // Use fs and vs_p.
   ...
   // Free pointer vs_p before passing it to
   // A_i::opOut() again.
   delete vs_p;
   aVar->opOut(*fs, vs_p);
   // Use fs and vs_p.
   ...
// Delete memory pointed to by vs_p
   delete vs_p;
}
```

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

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

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

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

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

Return Values

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

An Example of Applying the Rules for Object References

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

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

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

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

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

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

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

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

ImplementingIDL

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

Overview of an Example Application

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

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

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

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

Overview of the Programming Steps

1. Define IDL interfaces to your application objects.
2. Compile the IDL interfaces.
3. Implement the IDL interfaces with C++ classes.
4. Write a server program that creates instances of the implementation classes. This involves:

   Initializing the ORB.

Creating initial implementation objects.

Allowing Orbix to receive and process incoming requests from clients.

5. Write a client program that accesses the server objects. This involves:

   Initializing the ORB.

   Getting a reference to an object.

   Invoking object attributes and operations.

6. Compile the client and server.
7. Run the application. This involves:

   Running the Orbix daemon process.

   Registering the server in the Implementation Repository.

   Running the client.

Defining IDL Interfaces

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

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

```
// IDL
// In banksimple.idl
module BankSimple {
         typedef float CashAmount;
         interface Account;
            // A factory for bank accounts.
      interface Bank {
         // Create new account with specified name.
         Account create_account(in string name);
         // Find the specified named account.
         Account find_account(in string name);
      };
      interface Account {
         readonly attribute string name;
         readonly attribute CashAmount balance;
         void deposit(in CashAmount amount);
         void withdraw(in CashAmount amount);
      };
   };
```

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

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

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

Implementing IDL Interfaces

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

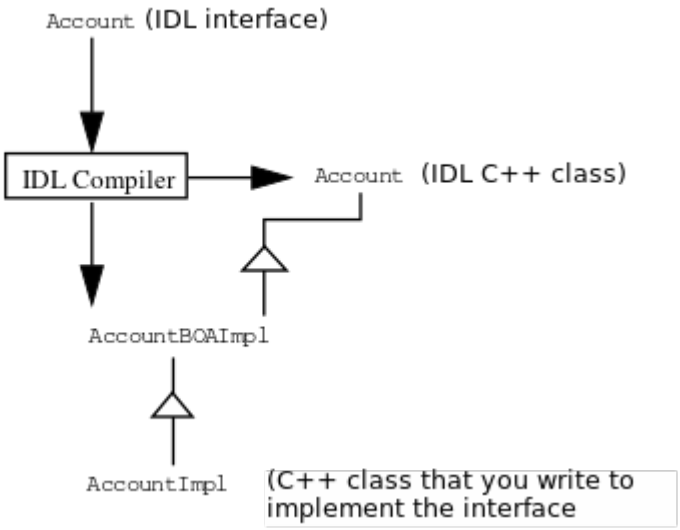
- The *BOAImpl* approach.
- The *TIE* approach.

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

The BOAImpl Approach to Implementing Interfaces

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

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



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

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

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

The TIE Approach to Implementing Interfaces

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

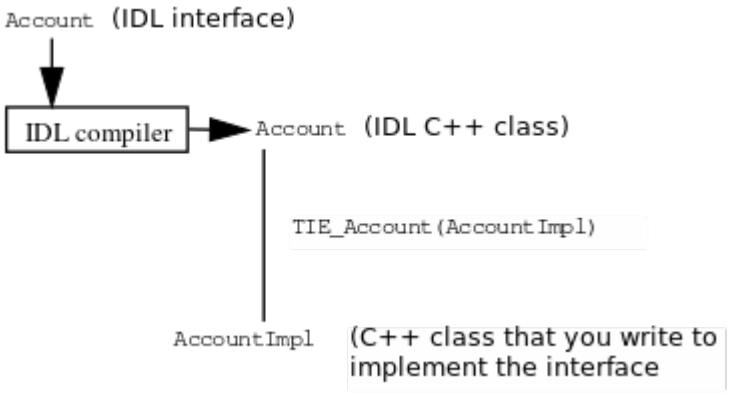
To use the `TIE` mechanism, the server programmer indicates that a particular class implements a given IDL C++ class by calling a `DEF_TIE` macro, which has the general form:

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

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

```
DEF_TIE_Account(AccountImpl)
```

This generates a class named `TIE_Account(AccountImpl)`. Figure 10 shows the TIE approach. For simplicity, the fully scoped name, `BankSimple::Account`, is not used.



`DEF_TIE` macros also work when interfaces are defined in IDL modules. For example, if interface `I` is defined in module `M`, the macros take the following form:

```
DEF_TIE_M_Impl (*implementation class name*)
TIE_M_Impl (*implementation class name*)
```

For example, interface `Account` is defined in module `BankSimple` and implemented by C++ class `AccountImpl`. The macros thus take the following form:

- `DEF_TIE_BankSimple_Account(BankSimple_AccountImpl)`

  This macro is called in the implementation header file (in this case, `banksimple_accountimpl.h`).

- `TIE_BankSimple_Account(BankSimple_AccountImpl)`

  This macro is called in the implementation file (in this case, `banksimple_bankimpl.cxx`).

Refer to Using the TIE Approach for more details.

Defining Implementation Classes for IDL Interfaces

This section illustrates both the BOAImpl and TIE approaches. Two implementation classes are required:

`BankSimple_BankImpl`	Implements the `Bank` interface.
`BankSimple_AccountImpl`	Implements the `Account` interface.

> **Note**
>
> You can automatically generate a skeleton version of the class and function definitions for `BankSimple::BankImpl` and `BankSimple::AccountImpl` by specifying the `-S` switch to the IDL compiler.

The `-S` switch produces two files. If the IDL definitions are in the file `banksimple.idl`, the skeleton definitions are placed in the following files:

`banksimple_ih`	This is the class header file that defines the class. This file declares the member functions that you must implement. It can be renamed to `banksimple_bankimpl.h.`
`banksimple.ic`	This is the code file. It gives an empty body for each member function and can be renamed to `banksimple_bankimpl.cxx`.

You can edit both files to provide a full implementation class. You must add member variables, constructors, and destructors. Other member functions can be added if required. You can use either the BOAImpl or the TIE approach to relate the implementation classes to your IDL C++ classes.

Using the BOAImpl Approach

Using this approach, you should indicate that a class implements a specific IDL interface by inheriting from the corresponding BOAImpl-class generated by the IDL compiler:

```
// C++
// In file banksimple_accountimpl.h
#define BANKSIMPLE_ACCOUNTIMPL_H_
#include “banksimple.hh”
// The Account implementation class.
class BankSimple_AccountImpl :
         public virtual BankSimple::AccountBOAImpl {
   public:
      // IDL operations
      virtual void deposit
         (BankSimple::CashAmount amount,
            CORBA::Environment&);
      virtual void withdraw
         (BankSimple::CashAmount amount,
            CORBA::Environment&);
      // IDL attributes
      virtual char* name(CORBA::Environment&);
      virtual void name
         (const char*  _new_value, CORBA::Environment&);
      virtual BankSimple::CashAmount balance
         (CORBA::Environment&);
      // C++ operations
      BankSimple_AccountImpl
         (const char* name, BankSimple::CashAmount
            balance);
      virtual ~BankSimple_AccountImpl();
   protected:
      CORBA::String_var m_name;
      BankSimple::CashAmount m_balance;
      ...
};
// C++
// In file banksimple_bankimpl.h.
#define BANKSIMPLE_BANKIMPL_H_
#include <banksimple.hh>
// The Bank implementation class.
class BankSimple_BankImpl : public virtual BankSimple::BankBOAImpl {
   public:
      // IDL operations.
      virtual BankSimple::Account_ptr
      create_account(const char* name,
            CORBA::Environment&);
      virtual BankSimple::Account_ptr
      find_account(const char* name,
            CORBA::Environment&);
      // C++ operations.
```

```
        BankSimple_BankImpl();
        virtual ~BankSimple_BankImpl();
        protected:
        // This bank stores account in an array in memory.
        static const int MAX_ACCOUNTS;
        BankSimple::Account_var* m_accounts;
        ....
};
```

Note

The BOAImpl class is produced only if the `-B` switch is specified to the IDL compiler.

Classes `BankSimple_BankImpl` and `BankSimple_AccountImpl` redefine each of the functions inherited from their respective BOAImpl classes. They can also add constructors, destructors, member functions and member variables. Virtual inheritance is not strictly necessary in the code shown; it is used in case C++ multiple inheritance is required later. Any function inherited from the BOAImpl class is virtual because it is defined as virtual in the BOAImpl class. Therefore, it is not strictly necessary to explicitly mark them as virtual in an implementation class (for example, `BankSimple_AccountImpl`).

The accounts managed by a bank are stored in a array with members of type `BankSimple::Account_var`.

Outline of the Bank Implementation (BOAImpl Approach)

First, in `BankSimple_BankImpl::create_account()`, you should construct a new `BankSimple::Bank` object. The function `create_account()` corresponds to an IDL operation, and its return value is of type `BankSimple::Account_ptr`:

```
// C++
// In file banksimple_bankimpl.cxx.
// Add a new account.
BaBankSimple::Account_ptr
BankSimple_AccountImpl::create_account
   (const char* name, CORBA::Environment&) {
   int i = 0;
   for ( ; i < MAX_ACCOUNTS&  !CORBA::is_nil(m_accounts[i]); ++i)
   { }
   if (i < MAX_ACCOUNTS){
      // Create an account with zero balance.
      m_accounts[i]=new BankSimple_AccountImpl(name, 0.0);
      cout << “create_account: Created with name:” << name << endl;
      return BankSimple::Account::_duplicate(m_accounts[i]);
   }
   else {
      // Cannot create an account, return nil.
      cout << “create_account: failed, no space left!” << endl;
      return BankSimple::Account::_nil();
   }
}
```

You must call `BankSimple::Account::_duplicate()` because Orbix calls `CORBA::release()` on any object returned as an `out` / `inout` parameter or as a return value. The reference count on the new object is initially one, and subsequently calling `CORBA::release()` without first calling `BankSimple::Account::_duplicate()` results in deletion of the object.

Using the BOAImpl approach, the `Bank` implementation code is as follows:

```
// C++
// In file bankSimple_bankImpl.cxx.
// Implementation of the BankSimple::Bank interface.
#include “banksimple_bankimpl.h”
#include “banksimple_accountimpl.h”
// Maximum number of accounts handled by the bank.
const int BankSimple_BankImpl::MAX_ACCOUNTS = 1000;
// BankSimple_BankImpl constructor.
BankSimple_BankImpl::BankSimple_BankImpl():
   m_accounts(new BankSimple::Account_var[MAX_ACCOUNTS]) {
   // Make sure all accounts are nil.
   for (int i = 0; i < MAX_ACCOUNTS; ++i) {
      m_accounts[i] = BankSimple::Account::_nil();
   }
   // BankSimple_BankImpl destructor.
   BankSimple_BankImpl::~BankSimple_BankImpl() {
      delete [] m_accounts;
   }
   // Add a new account.
   BankSimple::Account_ptr
   BankSimple_AccountImpl::create_account
      (const char* name, CORBA::Environment&) {
   ...
   }
   // Find a named account
   BankSimple::Account_ptr
   BankSimple_BankImpl::find_account
      (const char*  name, CORBA::Environment&) {
   ...
   }
}
```

In this example, the possibility of making the server objects persistent is ignored. You can do this by storing the account and bank data in files or in a database. Refer to the section Loading Objects at Runtime for more details.

Using the TIE Approach

Using the TIE Approach, an implementation class does not have to inherit from any particular base class. Instead, a class implements a specific IDL interface by using the `DEF_TIE` macro.

The DEF_TIE Macro

A version of the `DEF_TIE` macro is available for each IDL C++ class. The macro takes one parameter—the name of a C++ class implementing this interface:

```
// C++
// In file banksimple_accountimpl.h
class BankSimple_AccountImpl {
... // As before.
};
// DEF_TIE Macro call.
DEF_TIE_BankSimple_Account(BankSimple_AccountImpl)
```

This macro call defines a `TIE` class that indicates that class `BankSimple_AccountImpl` implements interface `BankSimple::Account.`

```
// C++
// In file banksimple_bankimpl.h
class BankSimple_BankImpl {
   . . .  // As before.
};
// DEF_TIE Macro call.
DEF_TIE_BankSimple_Bank(BankSimple_BankImpl)
```

This macro call defines a `TIE` class which indicates that class `BankSimple_BankImp` l implements interface `BankSimple::Bank.`

The TIE Class

The `TIE_BankSimple_Account(BankSimple_AccountImpl)` construct is a preprocessor macro call that expands to the name of a C++ class representing the relationship between the `BankSimple::Account` and `BankSimple_AccountImpl` classes. This class is defined by the macro call `DEF_TIE_BankSimple_Account(BankSimple_AccountImpl)`. Its constructor takes a pointer to a `BankSimple_AccountImpl` object as a parameter.

The C++ class generated by calling the macro `TIE_BankSimple_Account(BankSimple_AccountImpl)` has a name that is a legal C++ identifier, but you do not need to use its actual name. You should use the macro call `TIE_BankSimple_Account(BankSimple_AccountImpl)` when you wish to use this class.

The TIE approach gives a *complete* separation of the class hierarchies for the IDL Compiler-generated C++ classes and the class hierarchies of the C++ classes used to implement the IDL interfaces.

Consider an IDL operation that returns a reference to an `Account` object; for example, `BankSimple::Bank::create_account()`. In the IDL C++ class, this is translated into a function returning an `BankSimple::Account_ptr`.

However, using the TIE approach, the actual object to which a reference is returned is of type `BankSimple_AccountImpl`. This is not a derived class of `BankSimple::Account`. Therefore, the server should create an object of type `TIE_BankSimple_Account(BankSimple_AccountImpl)`. This `TIE` object references the `BankSimple_AccountImpl` object, and a reference to the `TIE` object should be returned by the function. This is because the class `TIE_BankSimple_Account(BankSimple_AccountImpl)` is a derived class of class `BankSimple::Account`. All invocations on the TIE object are automatically forwarded by it to the associated `BankSimple_AccountImpl` object.

When you code the server you create the `BankSimple_AccountImpl` object and a `TIE` object. The server should then use the `TIE` object, rather than the `BankSimple_AccountImpl` object directly. A bank's linked list of accounts, for example, should then point to TIE objects, rather than directly pointing to the `BankSimple_AccountImpl` objects.

A `TIE` object automatically delegates all incoming operation calls to its corresponding implementation object. For example, all invocations on a `TIE_Account(BankSimple_AccountImpl)` object are automatically passed to the `BankSimple_AccountImpl` object to which the `TIE` object holds a pointer.

> **Note**
>
> By default, calling `CORBA::release()` on a `TIE` object with a reference count of one also deletes the referenced object. The `TIE` object's destructor calls the `delete` operator on the implementation object pointer it holds. This is usually the desired behavior; however, you can use `CORBA::BOA::propagateTIEdelete()` to specify whether the `TIE` object should be deleted. Refer to the **Orbix Programmer's Reference C++ Edition** for more details.

Using the TIE approach, the bank service header file might look as follows:

```
// C++
// In file banksimple_bankimpl.h
#define BANKSIMPLE_BANKIMPL_H_
#include <banksimple.hh>
class BankSimple_BankImpl {
public:
   // IDL-defined operations.
   virtual BankSimple::Account_ptr
   create_account(const char*  name, CORBA::Environment&);
   virtual BankSimple::Account_ptr
   find_account(const char*  name, CORBA::Environment&);
   // C++ operations.
   BankSimple_BankImpl();
   virtual ~BankSimple_BankImpl();
protected:
   // This bank steored accounts in an array of Account_var.
   static const int MAX_ACCOUNTS;
   BankSimple::Account_var* m_accounts;
};
// Indicate that BankSimple_BankImp implements
// IDL interface BankSimple::Account.
DEF_TIE_BankSimple_Account(BankSimple_BankImpl)
```

Outline of the Bank Implementation (TIE Approach)

An outline of the code for `BankSimple_BankImpl::create_account()` is shown below:

```
// C++
// In file banksimple_bankimpl.cxx
BankSimple::Account_ptr BankSimple_BankImpl::create_account
   (const char*  name, CORBA::Environment&) {
   // Ensure that a valid account name is found.
   int i = 0;
   for ( ; i < MAX_ACCOUNTS& & CORBA::is_nil(m_accounts[i])
         ++i ) {
      ...
   }
   if (i < MAX_ACCOUNTS) {
      // Create an account with zero balance.
      m_accounts[i] =
          new TIE_BankSimple_Account(BankSimple_AccountImpl)
                  (new BankSimple_AccountImpl(name, 0.0));
   ...
   }
   else {
      ... // Cannot create account, return nil.
   }
};
```

The `BankSimple::Account_ptr` is initialized to reference a `TIE` object that points in turn to the new `BankSimple_AccountImpl` object.

Note

The object that a TIE object points to *must* be dynamically allocated using C++ operator `new`. By default, when a TIE object is destroyed, it deletes the object that it points to. The object must therefore be dynamically allocated.

Using the TIE approach, the Bank implementation class code is:

```
// C++
// In file banksimple_bankimpl.cxx
// Implementation of the BankSimple::Bank interface.
#include “banksimple_bankimpl.h”
#include “banksimple_bccountimpl.h”
const int BankSimple_BankImpl::MAX_ACCOUNTS = 1000;
// Constructor.
BankSimple_BankImpl::BankSimple_BankImpl():
   m_accounts(new BankSimple::Account_var[MAX_ACCOUNTS]) {
   // Make sure all accounts are nil.
   for (int i = 0; i < MAX_ACCOUNTS; ++i) {
         m_accounts[i] = BankSimple::Account::_nil();
   }
}
// Destructor.
BankSimple_BankImpl::~BankSimple_BankImpl() {
   delete [] m_accounts;
}
// Add a new account.
BankSimple::Account_ptr
   BankSimple_AccountImpl::create_account(const char*name,
                           CORBA::Enviroment& ) {
   int i = 0;
   for ( ; i < MAX_ACCOUNTS&  !CORBA::is_nil(m_accounts[i]);
                       ++i)
   { }
   if (i < MAX_ACCOUNTS) {
      m_accounts[i]=
         new TIE_BankSimple_Account(BankSimple_AccountImpl)
                  (new BankSimple_AccountImpl(name, 0.0));
      cout << “create_account: Created with name:” << name
          << endl;
      return BankSimple::Account::_duplicate(m_accounts[i]);
   }
   else {
      cout << “create_account: failed, no space left!” << endl;
      return BankSimple::Account::_nil();
   }
}
// Find a named account
BankSimple::Account_ptr
   BankSimple_BankImpl::find_account (const char*  name,
                           CORBA::Enviroment& ) {
   ... // Same as for BOAImpl approach.
   }
};
```