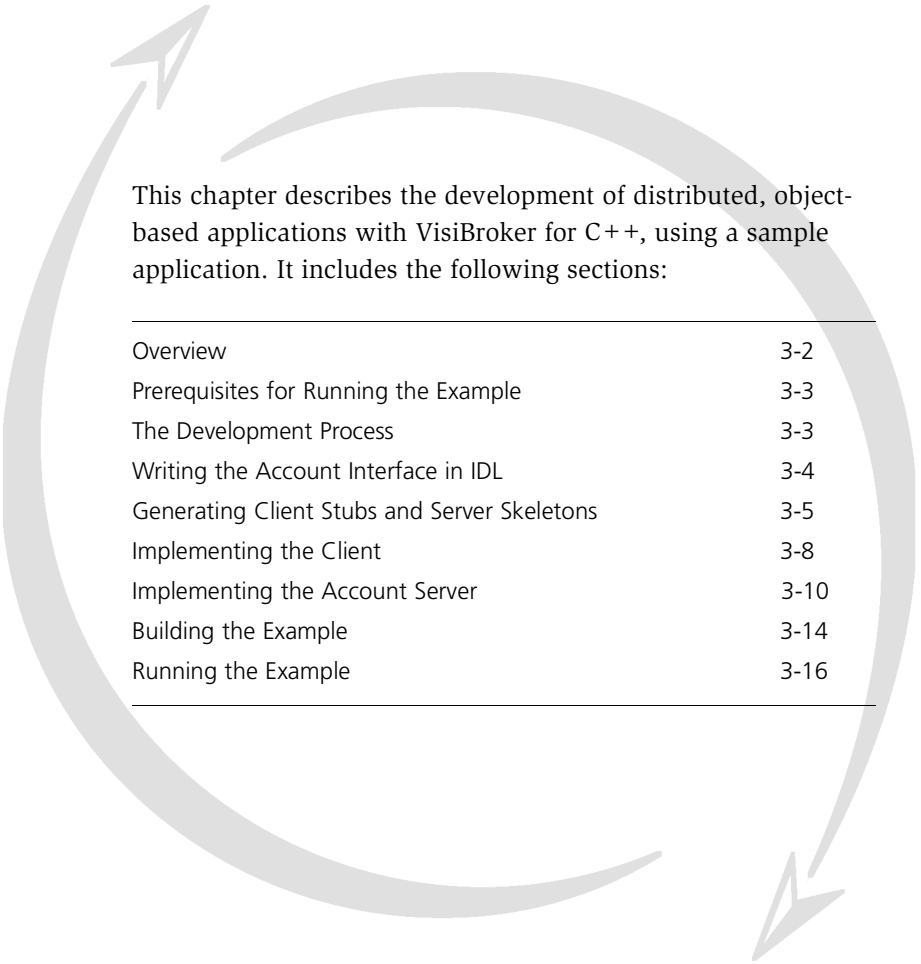
CHAPTER 3

QUICK START FOR DEVELOPMENT WITH VISIBROKER FOR C++



This chapter describes the development of distributed, object-based applications with VisiBroker for C++, using a sample application. It includes the following sections:

Overview	3-2
Prerequisites for Running the Example	3-3
The Development Process	3-3
Writing the Account Interface in IDL	3-4
Generating Client Stubs and Server Skeletons	3-5
Implementing the Client	3-8
Implementing the Account Server	3-10
Building the Example	3-14
Running the Example	3-16

OVERVIEW

In this chapter, you will build a sample client program that can query on the balance in a bank account. From this example, you will learn how to:

- Implement a simple interface in IDL for the `Account` object.
- Generate client stubs and server skeletons using the `idl2cpp` compiler.
- Implement the client program: initialize the ORB, bind to the `Account` object, obtain the balance, and handle exceptions.
- Implement the server object: initialize the ORB and BOA, instantiate and register an implementation of the server object with the BOA, and prepare to receive requests.
- Build the example.
- Run the example: start the Smart Agent, start the `Account` server, and run the client program.

The code for this example is provided in the account.html file under the directory examples/account where VisiBroker for C++ package was installed. If you do not know the location of the VisiBroker for C++ package, see your system administrator.

The following files are included for the example:

FILE	DESCRIPTION
account.idl	Contains the IDL interface for the `Account` object.
account_srvr.cpp **account_srvr.C**	`Account` server. Creates an instance of the `Account` object and calls `CORBA::BOA::impl_is_ready()` to make this object available to client programs. The `Account` server implementation implements the `balance()` method which returns the balance in a person's account whose name is provided as input (by generating a random number).
account_clnt.cpp **account_clnt.C**	This is the `account` client. It binds to an `Account` object and invokes `balance()` on the object reference obtained.
account_srvrtie.cpp **acount_srvtie.C**	Implements the `account_server` using the TIE mechanism instead of using the inheritance approach. This chapter does not discuss this feature—see Chapter 11, "Using the Tie Mechanism: An Alternative to Inheritance" for details.
Makefile	Used to build all the test targets.

Table 3-1 Files included with the quick start example.

PREREQUISITES FOR RUNNING THE EXAMPLE

You must configure your runtime environment before you attempt to execute the sample application presented in this chapter. See Chapter 2, "Setting Up Your Environment," for details on configuring your environment.

THE DEVELOPMENT PROCESS

When you develop distributed applications with VisiBroker, you must first identify the objects required by the application. You will then usually follow these steps:

1 Write a specification for each object using the Interface Definition Language (IDL).

IDL is the language that an implementer uses to specify the operations that an object will provide and how they should be invoked. In this example, we define the `Account` object in IDL with one method—`balance()`.

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

Using the idl2cpp compiler, we'll produce a client stub (which provides the interface to the `Account` object's `balance()` method) and a server skeleton (which provides an interface to the `balance()` method).

3 Write the client program code.

To complete the implementation of the client program, we must initialize the ORB, bind to the `Account` object, invoke the `balance()` method on the `Account` object, and print out the balance.

4 Write the server object code.

To complete the implementation of the server object code, we must derive from the `Account` class, provide an implementation of the `balance()` method, and implement the server's `main` routine.

5 Compile the client and server code.

To create the client program, we must compile and link the client program code with the client stub. To create the `Account` server, we must compile and link the server object code with the server skeleton.

6 Start the server.

7 Run the client program.

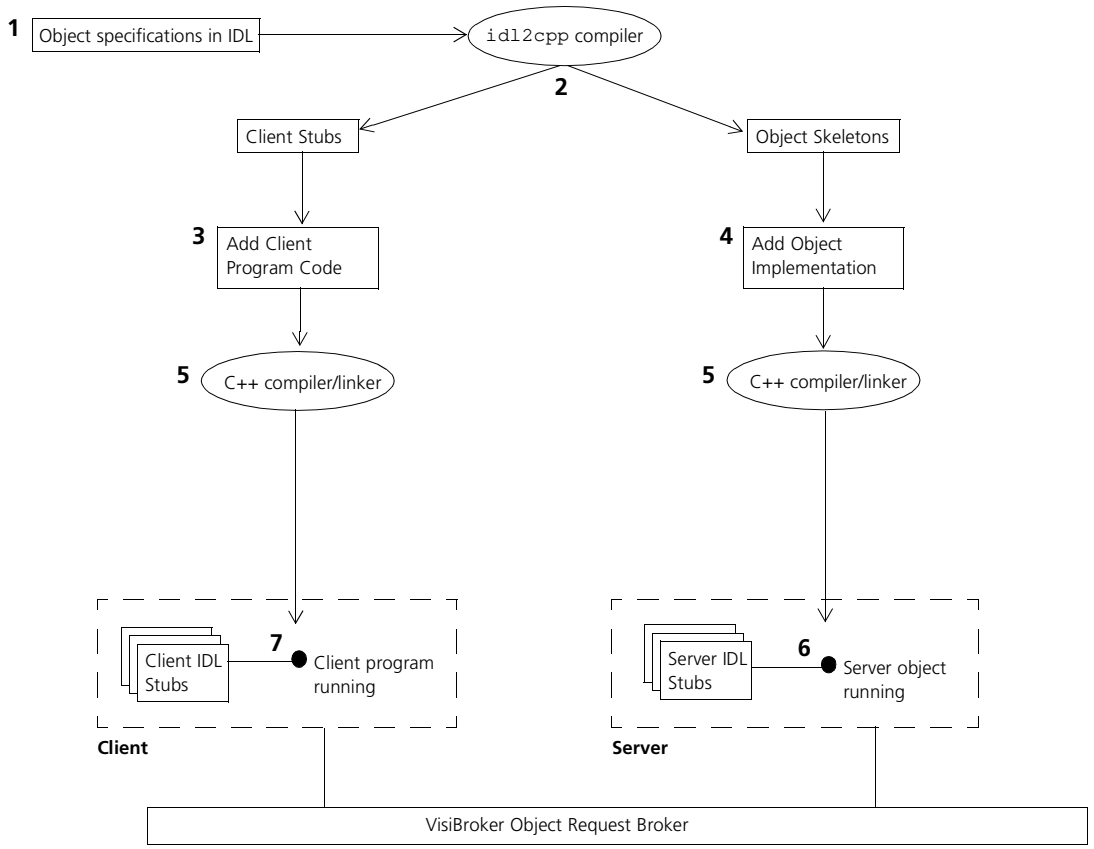


Figure 3-1 Process for developing the sample bank account application.

WRITING THE ACCOUNT INTERFACE IN IDL

The first step to creating an application with VisiBroker is to specify all of your objects and their interfaces using the CORBA Interface Definition language (IDL). IDL has a syntax similar to C++, but can be mapped to a variety of programming languages. The IDL mappings for the C++ language are covered in the *VisiBroker for C++ Reference Manual*.

IDL Sample 3-1 shows the contents of the account.idl file that defines the `account` interface. The `Account` interface provides a single method for obtaining the current balance.

IDL
```
interface Account {
        float balance();
};
```

IDL Sample 3-1 The account.idl file provides the `Account` interface definition.

GENERATING CLIENT STUBS AND SERVER SKELETONS

The interface specification you create in IDL is used by VisiBroker's `idl2cpp` compiler to generate C++ stub routines for the client program, and skeleton code for the object implementation. The stub routines are used by the client program for all method invocations. You use the skeleton code, along with code you write, to create the server that implements the objects.

The code for the client program and server object, once completed, is used as input to your C++ compiler and linker to produce the executable client program and server object. These steps are shown in Figure 3-1 on page 3- 4.

Because the account.idl file requires no special handling, it can be complied with the following command.

```
prompt> idl2cpp account.idl
```

For more information on the command line options for the `idl2cpp` compiler, see the *VisiBroker for C++ Reference Manual*.

Files Produced by the idl2cpp Compiler

The `idl2cpp` compiler generates these four files from the account.idl file:

- account_c.hh - Contains the definitions for the `Account` classes.
- account_c.cc (UNIX), account_c.cpp (Windows)- Contains internal stub routines used by the client.
- account_s.hh - Contains the definitions for the `_sk_Account` skeleton class.
- account_s.cc (UNIX), account_s.cpp (Windows)- Contains the internal routines used by the server.

You will use the account_c.hh and account_c.cc (UNIX) or account_c.cpp (Windows) files to build the client application. The account_s.hh and account_s.cc (UNIX) or account_s.cpp (Windows) files are for building the server object. All generated files have either a "cc " (UNIX only) or "hh" suffix to help you distinguish them from source files.

CAUTION *You should never modify the contents of files generated by the `idl2cpp` compiler.*

W The files that you create yourself traditionally use the “cpp” and “h” extensions.

U The files that you create yourself traditionally use the “C” and “h” extensions.

Looking at the Client Account Class

Code Sample 3-1 highlights the important aspects of the `account` class, defined in the account_c.hh file. The `Account` class is derived from the `CORBA::Object` class.

CODE
```
...
class Account: public virtual CORBA::Object {
    ...
    public:
    static Account_ptr _duplicate(Account_ptr obj) {
    if (obj) obj->_ref();
      return obj;
    }
    static Account_ptr _nil() { return (account_ptr)NULL;}
    static Account_ptr _narrow(CORBA::Object *obj);
    static Account_ptr _clone(Account_ptr obj) {
      CORBA::Object_var obj_var(__clone(obj));
      return _narrow(obj_var);
    }
    static Account_ptr _bind(const char *object_name = NULL,
           const char *host_name = NULL,
           const CORBA::BindOptions* opt = NULL,
           CORBA::ORB_ptr orb=NULL);
    virtual CORBA::Float balance();
    ...};
```

Code Sample 3-1 A portion of the `Account` class generated by the `idl2cpp` compiler.

THE _BIND() METHOD

When your client program invokes the `_bind()` method, the ORB uses the Smart Agent to locate the appropriate server object and return a handle to the `Account` object. If the ORB cannot locate or connect to an `Account` object, the `_bind()` method will raise a system exception. The binding process is covered in detail in Chapter 4, "Accessing Distributed Objects with Object References." Additional aspects of the Smart Agent are covered in Chapter 10, "The Smart Agent Architecture."

THE BALANCE() METHOD

The `balance()` method generated by the `idl2cpp` compiler for your client program is actually a stub method. When your client program invokes the `balance()` method, a request is sent to the ORB with all the necessary parameters. The ORB ensures that the request is sent to the `Account` server object. Once the method is executed on the server, the ORB returns the results to your client program.

OTHER METHODS

Several other methods are provided that allow your client program to duplicate, initialize, and narrow an `Account` object reference. These methods are discussed in Chapter 4, "Accessing Distributed Objects with Object References."

Looking at the Server _sk_Account Class

The account_s.hh include file contains the C++ definitions for the `_sk_Account` class from which you derive your implementation of the `AccountImpl` server object. This class contains a skeleton method `_balance()`. This skeleton method is used by the ORB on the server side to unpack the parameters from your client program's `balance()` request and invoke the actual `balance()` method on the server object.

The `balance()` method is a pure virtual function. You create the actual implementation of this method for the `AccountImpl` server object. The following excerpt shows the `_sk_Account` class definition contained in the account_s.hh file.

CODE
```
class _sk_Account : public Account {
    protected:...
    public:...
    // The following operations need to be implemented by the server.
    virtual CORBA::Float balance() = 0;

    // Skeleton Operations implemented automatically
    ...};
```

Code Sample 3-2 Portions of the `_sk_Account` class.

IMPLEMENTING THE CLIENT

The file named account_clnt.cpp (Windows) or account_clnt.C (UNIX), part of the example, contains the implementation of the client program. Because your program uses the `Account` class, it must include the account_c.hh file.

The bank client program performs these steps:

1. Initializes the ORB.
2. Binds to an `Account` object.
3. Invokes the `balance()` method on the `Account` object, using the object reference returned by the `_bind()` method.
4. Prints out the balance.

Initializing the ORB

The first task that your client program needs to do is initialize the ORB object, as shown in Code Sample 3-3.

CODE
```
#include <iostream.h>
#include "account_c.hh"

int main(int argc, char* const* argv) {
       try {
         // Initialize the ORB.
         CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
         ...
```

Code Sample 3-3 Initializing the ORB.

Binding to the Account Object

Before your client program can invoke the `balance()` method, it must first use the `_bind()` method to establish a connection to the server that implements the `Account` object. The implementation of the `_bind()` method is generated automatically by the `idl2cpp` compiler. The `_bind()` method requests the ORB to locate and establish a connection to the server. If the server is successfully located and a connection is established, a proxy object is created to represent the server's `AccountImpl` object. A reference to the proxy object is returned to your client program.

When multiple instances of an object are offered by the same or different servers, an object name argument may be specified to indicate the specific object the client wishes to use.

In this simple example, the object name is omitted so any available object that offers the `Account` interface will be used. Object names are described in detail in Chapter 4, "Accessing Distributed Objects with Object References."

CODE
```
...
     // Locate an account.
     Account_var account = Account::_bind();
...
```

Code Sample 3-4 Binding to the `Account` object.

Obtaining the Balance

Once your client program has established a connection with an `Account` object, the `balance()` method can be used to obtain the balance. The `balance()` method on the client side is actually a stub generated by the `idl2cpp` compiler that gathers all the data required for the request and sends it to the server object.

CODE
```
...
     // Get the balance of the account.
     CORBA::Float acct_balance = account->balance();
     // Print out the balance.
...
```

Code Sample 3-5 Invoking the `balance()` method.

Handling Exceptions

Code Sample 3-6 shows the complete account_clnt.cpp (Windows) or account_clnt.C (UNIX) file. Notice how the `try` and `catch` statements are used to detect any failures, print a message, and exit the client program.

CODE
```
#include <iostream.h>
#include "account_c.hh"

int main(int argc, char* const* argv) {
    try {
          // Initialize the ORB.
          CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

          // Bind to an account.
          Account_var account = Account::_bind();

          // Get the balance of the account.
          CORBA::Float acct_balance = account->balance();
          // Print out the balance.
          cout  << "The balance in the account is $"
                << acct_balance << endl;
    }
    catch(const CORBA::Exception& e) {
          cerr << e << endl;
    }
}
```

Code Sample 3-6 The client program for the quick start example.

IMPLEMENTING THE ACCOUNT SERVER

There are a few tasks you must complete to implement the account server:

- Derive the `AccountImpl` class from the `Account` class.
- Provide an implementation for the `balance()` method.
- Implement the server's main routine.

Understanding the Account Class Hierarchy

The `Account` class that you implement is derived from the `_sk_Account` class that was generated by the `idl2cpp` compiler. Look closely at the `_sk_Account` class definition and notice that it is derived from the `Account` class that is defined in the account_c.hh file. Figure 3-2 shows the class hierarchy.

You can choose any name you wish for the server's implementation class, as long as it is distinct from the client's class name.

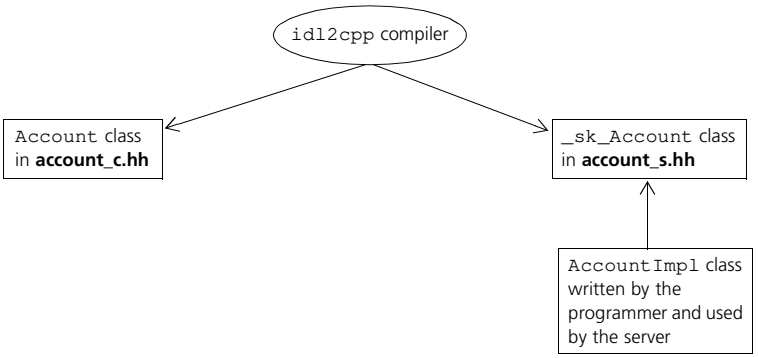


Figure 3-2 Class hierarchy for the `AccountImpl` interface.

Creating the AccountImpl Class

The `AccountImpl` class defines the constructor and the `balance()` method. The constructor sets the account balance to a random amount between 0 and 10,000 dollars.

CODE

```
#include "account_s.hh"
#include <math.h>

class AccountImpl: public _sk_Account {
    public:
        AccountImpl(const char *object_name) : _sk_Account(object_name) {
            // Set a random balance between 0 and 10000 dollars.
            _balance = abs(rand()) % 100000 / 100.0;
        }
        CORBA::Float balance() {
            return _balance;
        }
    private:
        CORBA::Float _balance;
};
```

Code Sample 3-7 The `AccountImpl` class.

Looking at the Account Server's main Routine

The bank server performs these steps in the `main` routine:

1. Initializes the ORB.
2. Instantiates an `AccountImpl` object.
3. Registers the `AccountImpl` object with the ORB.
4. Enters a loop waiting for client requests.

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

The `argc` and `argv` parameters passed to the `ORB_init()` and `BOA_init()` methods are the same parameters that are passed to the `main` routine. These parameters can be used to specify options for the ORB and BOA. They are described in the *VisiBroker for C++ Reference Manual*.

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

Code Sample 3-8 Initializing the ORB and BOA in the server.

After instantiating the `AccountImpl` object, the server tells the BOA that the object is ready by invoking the `obj_is_ready()` method. Lastly, the server calls the `impl_is_ready()` method to start the event loop that receives client requests. The details of the event loop are discussed in Chapter 19, "Event Loop Integration."

CODE

```
    ...
        // Create a new account object.
        AccountImpl account("Jack B. Quick");

        // Export the newly created object.
        boa->obj_is_ready(&account);
        cout << account << " is ready." << endl;
        // Wait for incoming requests
        boa->impl_is_ready();
    ...
```

Code Sample 3-9 Instantiating an object, registering the object, and waiting for client requests.

Code Sample 3-10 shows the complete account_srvr.cpp (Windows) or account_srvr.C (UNIX) file.

CODE

```
#include "account_s.hh"
#include <math.h>
class AccountImpl: public _sk_Account {
    public:
      AccountImpl(const char *object_name) : _sk_Account(object_name) {
        // Set a random balance between 0 and 10000 dollars.
        _balance = abs(rand()) % 100000 / 100.0;
      }
      CORBA::Float balance() {
        return _balance;
      }
    private:
      CORBA::Float _balance;
};
int main(int argc, char* const* argv) {
    try {
      // Initialize the ORB and BOA.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
      // Create a new account object.
      AccountImpl account("Jack B. Quick");
      // Export the newly created object.
      boa->obj_is_ready(&account);
      cout << account << " is ready." << endl;
      // Wait for incoming requests
      boa->impl_is_ready();}
    catch(const CORBA::Exception& e) {
      cerr << e << endl;
    }
}
```

Code Sample 3-10 The complete `Account` server implementation.

BUILDING THE EXAMPLE

W The account_clnt.cpp file that you created and the generated account_c.cc file are compiled and linked together to create the client program. The account_srvr.cpp file that you created, along with the generated account_s.cc and the account_c.cc files, are compiled and linked to create the bank account server. Both the client program and the server must be linked with the VisiBroker ORB library.

Use the -DSTRICT preprocessor option when linking the implementations. Otherwise, the linker may display an error message suggesting that a constructor is missing from orb.lib. To set the STRICT option using Microsoft Visual C++ 5.0:

1. From the Project menu, choose Settings, and then Category.
2. From Category, choose General.
3. Add STRICT to the preprocessor definitions.

   This resolves the link errors generated for orb.lib.

U The account_clnt.C file that you created and the generated account_c.cc file are compiled and linked together to create the client program. The account_srvr.C file that you created, along with the generated account_s.cc and the account_c.cc files, are compiled and linked to create the bank account server. Both the client program and the server must be linked with the VisiBroker ORB library.

The examples/account directory of your VisiBroker release contains a Makefile for this example. The examples directory also contains a file named stdmk, which is included by the Makefile and defines all site-specific settings. You may need to customize the examples/stdmk file.

Compiling the Example with make or nmake

W Assuming the VisiBroker distribution was installed in C:\vbroker, use the following commands to compile the example:

```
prompt> C:
prompt> cd \vbroker\examples\account
prompt> nmake
```

The Visual C++ `nmake` command runs the `idl2cpp` compiler and then compiles each file.

- Assuming the VisiBroker distribution was installed in /usr/local, issue these commands:

```
prompt> cd /usr/local/vbroker/examples/account
prompt> make
```

In this example, `make` is the standard UNIX facility. If you do not have it in your `PATH`, see your system administrator. Code Sample 3-11 shows a sample Makefile for UNIX.

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

ORBCC = $(ORBDIR)/bin/idl2cpp

SRCS = account_c.cc account_s.cc account_clnt.C account_srvr.C

.SUFFIXES: .o .cc .hh

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

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

#
# Account specific build parameters
#
all: account_client account_server

account_c.cc: account.idl
    $(ORBCC) account.idl

account_s.cc: account.idl
    $(ORBCC) account.idl

account_client: account_c.o account_clnt.o
    $(CC) -o account_client account_clnt.o \
    account_c.o $(ORBLIB) $(LDFLAGS)

account_server: account_s.o account_c.o account_srvr.o
    $(CC) -o account_server account_srvr.o \
    account_s.o account_c.o $(ORBLIB) $(LDFLAGS)

clean:
    rm -f *.o *.hh *.cc core account_client account_server
```

Code Sample 3-11 Sample `Makefile` for Solaris.

RUNNING THE EXAMPLE

Now that you have compiled your client program and server implementation, you are ready to run your first VisiBroker application.

Starting the Smart Agent

Before you attempt to run VisiBroker client programs or server implementations, you must first start the Smart Agent on at least one host in your local network. See the *VisiBroker for C++ Installation and Administration Guide* for details on starting the Smart Agent.

The Smart Agent is described in detail in Chapter 10, "The Smart Agent Architecture."

Starting the Account Server

W Open a DOS prompt window and start your Account server by using the following DOS command:

```
prompt> account_server
```

U Start your `Account` server by typing:

```
prompt> account_server &
```

Running the Client

W Open a separate DOS prompt window and start your client by using the following DOS command:

```
prompt> account_client
```

U To start your client program, type:

```
prompt> account_client
```

You should see output similar to that shown below.

```
The balance in the account in $168.38.
```