

O₂ ODMG Database System Tutorial

Release 5.0 - April 1998



Information in this document is subject to change without notice and should not be construed as a commitment by O₂ Technology.

The software described in this document is delivered under a license or nondisclosure agreement.

The software can only be used or copied in accordance with the terms of the agreement. It is against the law to copy this software to magnetic tape, disk, or any other medium for any purpose other than the purchaser's own use.

Copyright 1992-1998 O₂ Technology.

All rights reserved. No part of this publication can be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopy without prior written permission of O₂ Technology.

O₂, O₂Engine API, O₂C, O₂DBAccess, O₂Engine, O₂Graph, O₂Kit, O₂Look, O₂Store, O₂Tools, and O₂Web are registered trademarks of O₂ Technology.

SQL and AIX are registered trademarks of International Business Machines Corporation.

Sun, SunOS, and SOLARIS are registered trademarks of Sun Microsystems, Inc.

X Window System is a registered trademark of the Massachusetts Institute of Technology.

Unix is a registered trademark of Unix System Laboratories, Inc.

HPUX is a registered trademark of Hewlett-Packard Company.

BOSX is a registered trademark of Bull S.A.

IRIX is a registered trademark of Siemens Nixdorf, A.G.

NeXTStep is a registered trademark of the NeXT Computer, Inc.

Purify, Quantify are registered trademarks of Pure Software Inc.

Windows is a registered trademark of Microsoft Corporation.

All other company or product names quoted are trademarks or registered trademarks of their respective trademark holders.

Information in this document is subject to change without notice and should not be construed as a commitment by O₂ Technology.

The software described in this document is delivered under a license or nondisclosure agreement.

Who should use this tutorial

This tutorial is for programmers who wish to write or adapt C++ applications to run with O₂.

It presents an example application and describes the steps required to port this application to O₂. A prior knowledge of C++ is assumed.

It also presents the object query language (OQL) which can be used for stand-alone queries or embed queries in a C++ program.

The following manuals give further details on the topics covered in this tutorial:

ODMG C++ Binding Guide

ODMG C++ Binding Reference Manual

OQL User Manual

O₂ System Administration Guide

O₂ System Administration Reference Manual

Other documents available are outlined, click below.

See [O2 Documentation set](#).



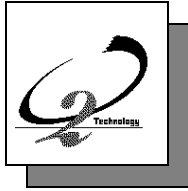


TABLE OF CONTENTS

This tutorial contains the following sections:

- 1 - Introduction
- 2 - Porting a C++ application
- 3 - OQL



TABLE OF CONTENTS

1	Introduction	9
	1.1 Introduction to O2	10
	Architecture.....	10
	1.2 Object Database Management Group (ODMG).....	11
	The O2 ODMG C++ Binding	12
	1.3 Benefits of porting to O2	13
	1.4 Example application.....	15
	Relevant files.....	17
	1.5 Setup	19
2	Porting a C++ Application	23
	2.1 Store C++ objects in an O2 database	26
	2.2 Change C++ pointers to persistent pointers	28
	2.3 Include files.....	32
	2.4 Connect to the database.....	33
	2.5 Run a transaction	34
	2.6 Create names.....	35
	2.7 Modify source code.....	36
	2.8 Create persistent objects	38
	2.9 Build and Run the Application	41
	2.10 Replace arrays by collection classes.....	42
	2.11 Build and Run the Application	49
3	OQL	51
	3.1 The O2 ODMG Object Query Language	52
	Interactive ad hoc queries	52
	Accessing data.....	53

TABLE OF CONTENTS

Selecting data.....	53
Manipulating complex values.....	54
Manipulating polymorphic collections.....	54
Composing operators.....	55
3.2 Embedding OQL in C++	56
3.3 Build and Run the Application	58



TABLE OF CONTENTS

1

Introduction

Introduction to the benefits of O₂.

This chapter briefly introduces the O₂ system and the benefits of porting a C++ application to O₂. It contains the following sections:

- [Introduction to O₂](#)
- [Object Database Management Group \(ODMG\)](#)
- [Benefits of porting to O₂](#)
- [Example application](#)
- [Setup](#)

1.1 Introduction to O₂

The O₂ ODMG database system is a fully ODMG compliant system, offering a powerful database engine, a C++ programming interface and an OQL query interface. The O₂ ODMG database system offers a data model fully compatible with the C++ type system. It provides transparent multi-user access to a database and includes a complete embedded query language for associative access to data.

The O₂ system has been designed to support large databases with a large number of users.

O₂ is a totally open system that integrates easily with other software. The open architecture of O₂ provides connections to relational database engines, GUI tools, the World Wide Web, programming languages and CASE and methodology tools.

O₂ is language and compiler independent. To provide maximum flexibility you can access the same database with any of the supported programming languages such as C, C++, Java, Eiffel, Lisp, Objective C and O₂C which is a fourth generation object language.

O₂ applications can be developed using the standard programming environments available or using O₂Tools. With O₂, you can use standard object-oriented methodologies and associated tools (Tramis, LOV, Objecteering, ROSE, etc.).

Architecture

The O₂ System supports a client server architecture. The client and server are separate processes which communicate over a network. There is usually a single server process which communicates with multiple clients.

The server (`o2server`) is responsible for reading and writing data to permanent storage, managing concurrent access to the data from multiple clients, backing up the database and recovering the database after a crash.

The client process is a user application program. For the purposes of this tutorial the client is an example C++ application. In the C++ sample application the communication with the server is transparent to the C++ developer. The C++ application automatically communicates with the `o2server` through the O₂ C++ runtime library.

1.2 Object Database Management Group (ODMG)

The Object Database Management Group (ODMG) is made up of the leading Object Database vendors plus a large number of companies that are interested in an ODBMS standard. The ODMG has produced a standard for object databases. The ODMG Standard is an interoperability standard which allows applications written to the standard to run on any compliant system.

This standard contains five categories:

- Object Model
- Object Definition Language (ODL)
- Object Query Language (OQL)
- C++ Language Binding
- Java Language Binding

The ODMG Object model is an extension of the OMG object model. The object model supports the notions of class, objects with attributes and methods, inheritance, and specialization. The extensions to the OMG object model include support for relationships between objects, support for collection classes and a set of base classes for date, time and character strings.

The ODMG-93 standard also allows for explicit names for an object or collection. From a name, an application can directly retrieve the named object and operate on it or navigate to other objects following relationship links.

A name in the schema plays the role of a variable in a program. Names are entry points to the database. From these entry points, other objects can be reached through associative queries or navigation.

The O₂ ODMG Database is fully compatible with the ODMG object model, object query language (OQL) and C++ binding.

The O₂ ODMG C++ Binding

The ODMG C++ language binding is designed to bring transparent persistence to C++ applications. Persistence is the ability for an object to outlive the process which created it. Transparency applies to both the type system and the programming language. From the C++ developer's perspective there is a single unified type system across the programming language and the database. Therefore any C++ class can be stored in the database. The database manipulation language is C++ so there is no need for a separate database programming language to access and manipulate data.

The ODMG C++ binding consists of standard C++ code. There are no new language constructs used for the binding. The binding makes use of C++ templates to provide persistence. Since the code is standard C++ code there is not a special pre-compiler for the source code, so you can continue to use any of your existing C++ tools as well as adopt new tools in the future.

1.3 Benefits of porting to O₂

A short banking application is provided for the purposes of this tutorial. It is an automated teller machine (ATM) application. The application allows users to login to their accounts to make deposits or withdrawals and to check the balance of an account or review recent transactions.

The changes required to port the application to O₂ involve less than 10% of the code and though only a small portion of code is modified the application gains substantially in terms of functionality. It gains in terms of durability, concurrency, scalability and accessibility through OQL.

The original version of the source code reads customer data at startup and writes information back when the program ends. If the program fails all modifications made will be lost. The possibility of losing data is unacceptable. The O₂ System protects data by ensuring that all changes committed to the database are indeed written to the database and that if a crash occurs the data is safe from corruption.

The O₂ System allows for concurrent access to data. With concurrent access it is possible to have multiple processes accessing the database at the same time. For example you could have several ATM machines accessing the same database. The database controls the access to the data ensuring that every client has a consistent view of the data.

One of the most important improvements to the sample application will be to make the application more scalable. The original application is limited in the amount of data it can store to machine memory. By using the O₂ database the application is no longer limited in this way. There is no practical limit to the size of an O₂ database.

The addition of ODMG collection classes and OQL queries reduces the physical limitations of the application. ODMG collection classes provide flexible collections which can grow as data expands and they are tightly linked to the database engine to provide efficient caching of large collections. By reusing the ODMG collection classes you will also reduce the amount of code that you need to develop and maintain.

The OQL query language is an optimized query language. All queries will be automatically optimized to improve the performance of your application. The OQL query language is a declarative query language so the physical structure of the data can be changed without having to change the query. This allows an administrator to add indexes to large collections without having to modify or recompile the application.

The O₂ System can replace existing code that implements database type features thus reducing the amount of code to be maintained. For new developments the O₂ System reduces the amount of code required. For the sample application we can remove the code for the `initializeAccounts` and `updateAccounts` methods. By removing the code which becomes useless through porting we reduce the size of the application by 200 lines or 25%.

One of the key features of the O₂ ODMG C++ Binding is that it does not extend the C++ language to add special database keywords. The binding uses standard C++ concepts such as templates and inheritance to provide connection to the database. The use of standard C++ does not force developers to change the methods or tools they use to develop code. Any tool that can work with standard C++ can be used with the O₂ ODMG C++ Binding. These include front end case tools which generate C++ code, editors, debuggers, memory management tools and third party class libraries.

This tutorial is intended to provide a quick introduction to the O₂ ODMG C++ Interface and the OQL query language. There are many important features of the O₂ System that were not covered in this tutorial. These include database features such as transaction types, schema updates, warm and cold recovery, and support for continuous operation.

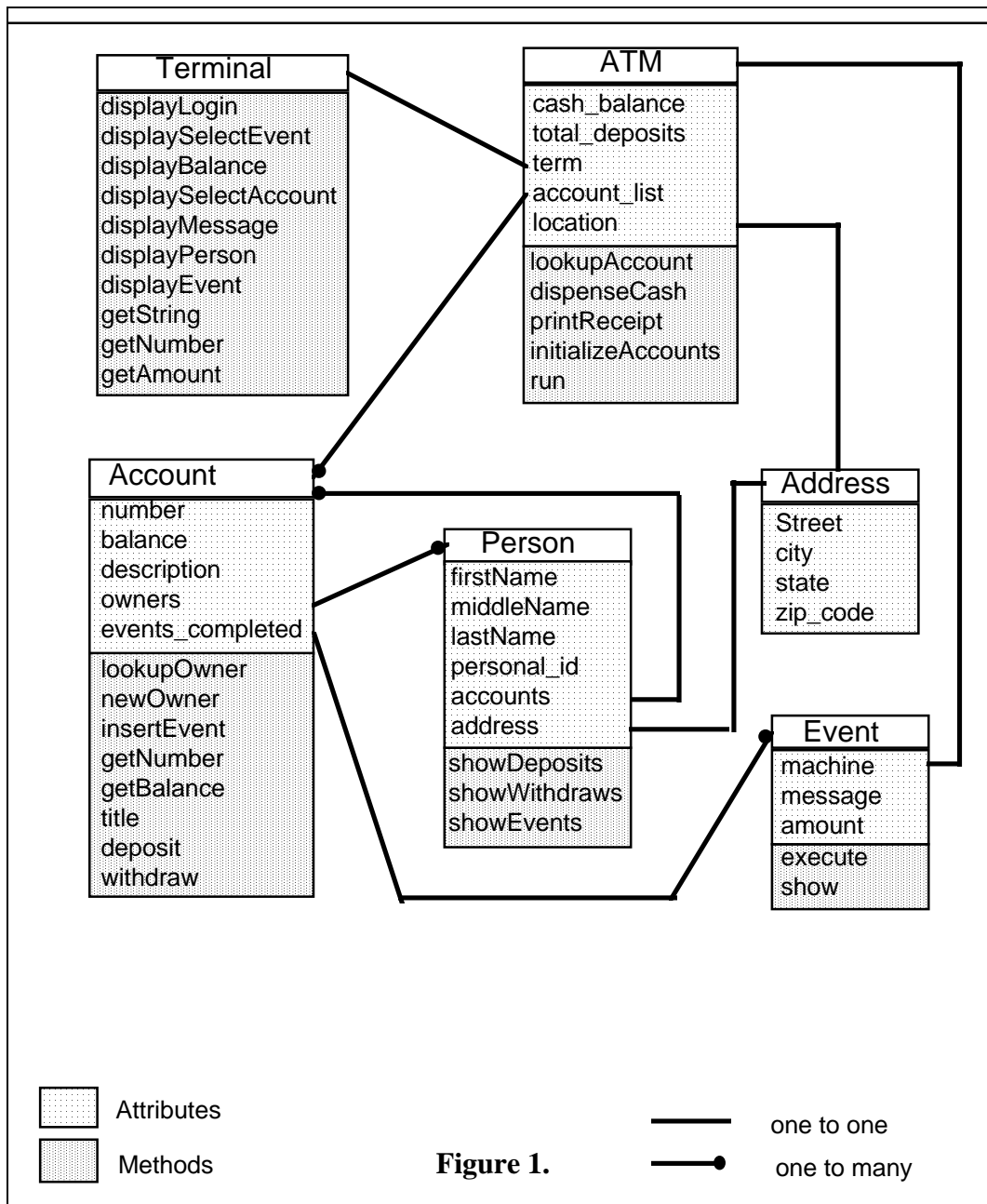
Another obvious advantage is O₂ Technology's commitment to industry standards. Using a system that complies with industry standards ensures the interoperability between your application and other tools and does not lock your source code to a specific vendor.

Example application

1.4 Example application

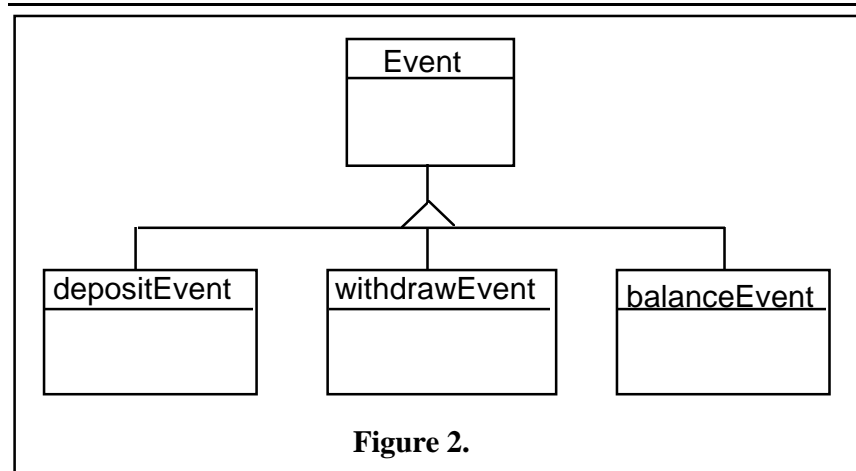
In this tutorial we are going to be working with a C++ banking application where customers use automatic teller machines (ATM) to check their accounts or carry out simple banking transactions. This is not a production application, but a simple program used for example purposes only. The application uses a simple terminal interface to communicate with customers.

Before being ported to O₂, the application uses standard operating system files to store information about customers and accounts. When the ATM application is started information about existing accounts is read from a file and when the application stops information is written back to the file. The class composition is shown in figure 1.



Example application

The inheritance hierarchy of the event classes is shown in figure 2.



Relevant files

The application source code is stored in various files:

main.cxx - contains the main program.

atm.hxx and **atm.cxx** - contain the definition of the ATM class and method definitions.

account.hxx and **account.cxx** - contain the definitions of the Address, Person and Account classes and method definitions.

event.hxx and **event.cxx** - contain the definitions of the Event, depositEvent, withdrawEvent and balanceEvent classes and their method definitions.

terminal.hxx and **terminal.cxx** - contain the Terminal class definition and method definitions.

MakefileUnix.original or **MakefileNt.original** - contain the makefile for compiling the original version of the application.

The main routine of the program creates the ATM, Person and Account objects, from data in a file and then passes control to the ATM::run method. The data file is located in data/ accounts.

Finally, **README** files describe the step by step process to run the tutorial. Please have a look at them.

The ATM::run method controls the interaction between customers and accounts. The method first displays a login prompt to the user. The login process produces an account number and a personal identification number (PIN). The ATM:lookupAccount method is then used to find an account corresponding to the account number. Then the Account::lookupOwner method is used to lookup a Person based on the PIN. The terminal then displays a list of actions that the customer can take.

The actions are represented as event objects which represent the interface between customers and their accounts. Event objects are saved in the database to maintain an account history. Event objects are the only new objects created in the database while the ATM:Run method is running. The other objects in the database are only modified when the ATM is running, for instance the Account object is updated to reflect any change in the balance of the account and also to add events to the event history.

Setup

1.5 Setup

It is assumed that the O₂ software is installed. If this is not the case, refer to the O₂ Installation sheet for further details.

The O₂ system must be set up before you can begin porting the example application.

Set an environment variable called **o2HOME** specifying the directory path where the O₂ software is installed.

Position yourself at **o2HOME**:

```
> cd $O2HOME
```

Add the **o2HOME/bin** directory to the **path** in each working window:

```
> set path = ($O2HOME/bin $path)
```

In order to run the O₂ System you need to have a **.o2KEYS** license file for the machine you are running on. The **.o2KEYS** file should be stored in the O₂ installation directory.

The **o2server** program calls the O₂ database server. Each **o2server** runs a single named system. The logical definition for each system is given in the **.o2serverrc** file located in the **o2HOME** directory. The **.o2serverrc** file contains the system name, name of the server machine, the location of the catalog, log and shadow directory. For more information about the **.o2serverrc** file refer to the O₂ System Administration Guide. From the **o2HOME** directory use the following commands to create a **.o2serverrc** file for the tutorial.

```
> mkdir o2vol
```

Edit the **.o2serverrc** file and add the following lines:

```
demo.server = <hostname>
demo.cataldir = $O2HOME/o2vol
demo.logdir = $O2HOME/o2vol
demo.shadowdir = $O2HOME/o2vol
```

where **<hostname>** is the name of the machine where the O₂ server will run.

The default system name and server name can be specified by the environment variable **o2OPTIONS**. Use the following commands to set **o2OPTIONS**:

On Unix,

```
(in sh) set O2OPTIONS "-system demo -server  
<hostname> ; export O2SYSTEM  
(in csh) setenv O2OPTIONS "-system demo -server  
<hostname>"
```

On Windows NT, use the **Environment** tab of the system properties panel to set the O2OPTIONS variable.

The O2OPTIONS environment variable is used by all O₂ utilities programs.

You must initialize a system before you can run a database server. The initialization process creates the files used by the database server. To initialize the system use the following command.

```
> o2dba_init -system demo -server <hostname>
```

Now start the database server for the system with the following command:

```
> o2server -system demo
```

Each system can contain multiple schemas and bases. A **schema** is a collection of class definitions. A **base** is a collection of instances of classes. Schemas allow for the reuse of code by letting you organize a group of classes that work together into a single schema. For instance, you could create a customer schema that could contain several classes to deal with customers. Then any other application that needed to deal with customers has only to import the relevant classes from the customer schema.

You must create the **schema** and **base** for the tutorial. This can be done from a C++ program or interactively using the O₂ administration tools. We will create the **schema** and **base** interactively. First start the O₂ administration tool.

```
> o2dsa_shell
```

Now create the schema and base. Commands are confirmed by typing a new line followed by ctrl D.

```
schema bankSchema;
```

```
base bankBase;
```

```
quit;
```

```
^D
```

Setup

Now copy the tutorial source code to a working directory. To complete the tutorial you will need approximately ten megabytes of disk space in the working directory. To copy the source code to a directory called `working_directory` execute the following:

On Unix,

```
> cd $O2HOME/samples/o2cplusplus/odmg_tutorial/  
> cp -r * working_directory
```

On Windows NT, simply make a `copy/paste` operation to the folder using the NT Explorer.

The directory called `working_directory` must already exist. Now move to the working directory

On Unix,

```
> cd working_directory/original
```

On Windows NT,

```
> cd working_directory\original
```

You are now ready to port the example application to O₂.



2

Porting a C++ Application

This chapter describes how to port an example C++ application to O₂.

It contains the following sections:

- [Store C++ objects in an O₂ database](#): importing class definitions to O₂
- [Change C++ pointers to persistent pointers](#)
- [Include files](#)
- [Connect to the database](#)
- [Run a transaction](#)
- [Create names](#)
- [Modify source code](#)
- [Create persistent objects](#)
- [Replace arrays by collection classes](#)

The example application works in its current state. Before proceeding we suggest you browse through the files to familiarize yourself with the application, then compile, link and execute it. To compile and link the application enter the following command:

On Unix
> `make -f MakefileUnix.original`

On Windows NT
> `nmake -F MakefileNt.original`

Once it has been compiled run the program through the following commands:

Start the application

> `bank`

Enter the Account number 11111 at the prompt and press enter. The following will be displayed.

```
Welcome to the Bank of O2
```

```
Enter Account Number, 0 to exit
```

```
> 11111
```

At the personal ID prompt enter the number 9999 and press enter.

```
Enter Personal ID Number, 0 to exit
```

```
> 9999
```

You are now logged into the system so you can carry out transactions with the accounts available.

Now exit the program.

We will now go through the steps necessary to port the existing application to the O2 System. Figure 3 shows the import, compile and link steps that are necessary to generate an O2 C++ database application.

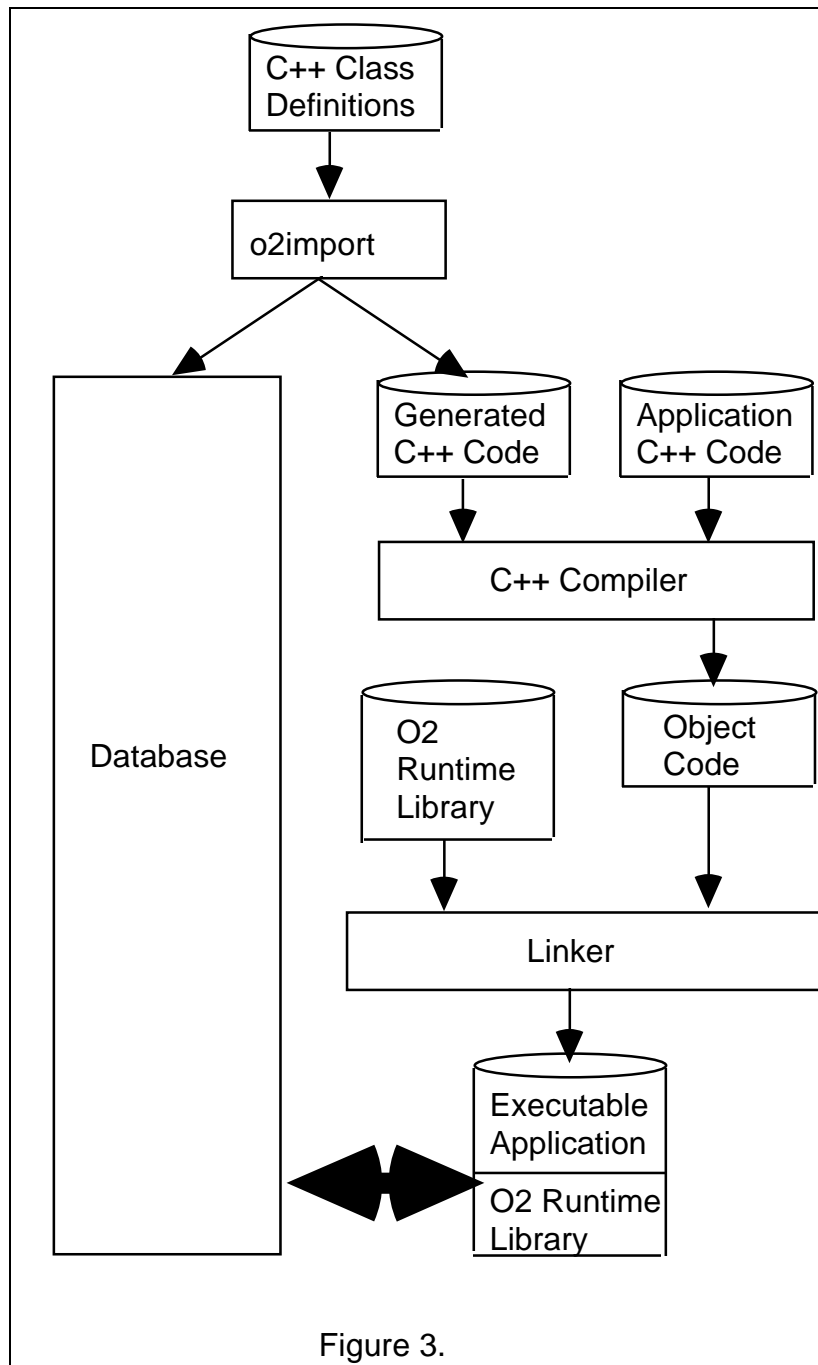


Figure 3.

2.1 Store C++ objects in an O₂ database

To store a C++ object in an O₂ database, the database needs information on the structure of the object. You can supply this information by importing C++ class definitions. The import process parses class definitions to generate meta information for the database and it enhances the existing class definitions to add persistence. An enhanced class definition is a subclass of a root persistent class (`o2_root`) and several methods are generated for the class which manage interaction with the database.

Once the class has been imported into O₂, instances of the class can be stored in the database or can continue to be used as normal transient C++ objects. Such classes are known as persistent capable classes.

The `o2cpp_import` tool imports class definitions into O₂. To simplify the process of importing multiple classes into O₂ a utility is provided. This is `o2makegen` and it generates a Makefile to import all your classes. The generated Makefile can be used directly or can be integrated into your existing make process.

The `o2makegen` program takes a configuration file as input. The configuration file for this tutorial is already written. The file is `bank.config`. This configuration file includes comments which describe the syntax. You should review this file to better understand the information needed to import classes into O₂.

The files imported must be compilable in C++ or self-contained. Header files are usually parsed during the compilation of a source file. The source file may provide context information that is needed to parse the header file. Command line options may also provide vital information. Most header files are self contained, however.

Some header files do require additional context information to be imported into O₂. The most common additional information needed is other class definitions. The best way to solve this is to order the import process so that any class definition needed by a class has already been imported.

Sometimes a closed circle of pointers can exist in class definitions, for example:

```
class A { B *pointer };
class B { A *pointer };
```

To break this cycle you can use the `ImpForwardClasses` option with `o2makegen` to declare classes which are needed to parse a file, but which have not yet been defined.

Store C++ objects in an O2 database

Once imported, the enhanced persistent capable class definitions must be used instead of the original transient class definitions. You can either have the `o2cpp_import` overwrite the original version of the header file or write it to another file. In this example we do not overwrite the header files. The line "[FILENAME]ImpOutputDir: output" in `bank.config` specifies that the generated files should be stored in the `output` directory.

To generate the Makefile for the application execute the following command:

```
> o2makegen bank.config
```

2.2 Change C++ pointers to persistent pointers

For each class T imported into O₂ an ancillary class `d_Ref<T>` is available. An instance of the `d_Ref` class is called a persistent pointer. The persistent pointer is then used to reference an instance of a persistent capable class. Persistent pointers are manipulated in the same way as standard C++ pointers. Persistent pointers maintain a relationship between two objects permanently in the database. Standard C++ pointers refer to an address which is valid only while a program is executing.

Before importing class definitions into O₂ you must change standard C++ pointers in persistent capable classes to persistent pointers. For instance in the example application the `Person` class has a pointer to an `Address` object.

```
class Person {
...
Address *address;
...
};
```

So that the database can maintain the relationship between the `Person` and the `Address` you need to change the pointer to a persistent pointer.

```
class Person {
...
d_Ref<Address> address;
...
};
```

The persistent pointer mechanism offers a number of advantages to the C++ programmer. Persistent pointers check referential integrity so your data is safe from corruption by dangling pointers. A persistent pointer can address more data than the standard 32 bit C++ pointer so your database can be scaled to a larger size.

`o2cpp_import` helps you by automatically modifying the class definition to use persistent pointers instead of standard C++ pointers. If an attribute is a pointer to another known persistent class the pointer will automatically be changed to a persistent pointer. The method signatures may also contain references to persistent capable classes. However, method signatures may or may not need to be changed depending on whether or not the data referenced is persistent. For this reason method signatures are not modified by `o2cpp_import` and it is left to you to decide if method signatures should be changed.

In the `account.hxx` file you only need to change two method signatures of the `Account` class. The `Person` and `Address` class can be imported without change. You must make the changes outlined below to the `Account` class.

Change C++ pointers to persistent pointers

When making changes you can delete lines or convert them to comments. Both the original version and the modified version are in the file. The modified version is in comments. You need only remove the comments and delete the original version or convert it to comments. The best way to find the lines that need to be changed is to search for the class name or method name where the change takes place. The line "// ODMG Persistent Version" should remain in comments.

- ***account.hxx file***

In the Account class definition in the account.hxx file delete or comment out the following lines:

```
// Original Version
Person *lookupOwner(int pin_number) const;
void newOwner(Person *NewOwner);
void insertEvent(Event* event);
```

and add or un-comment the following lines in their place:

```
// ODMG Persistent Version
d_Ref<Person> lookupOwner(int pin_number) const;
void newOwner(d_Ref<Person> &NewOwner);
void insertEvent(const d_Ref<Event> &event);
```

You must also change the corresponding signature of the method definition.

- ***account.cxx file***

In the file account.cxx delete or comment out the following line:

```
// Original Version
Person * Account::lookupOwner(int pin_number) const {
```

and add or un-comment the following line in its place:

```
// ODMG Persistent Version
d_Ref<Person> Account::lookupOwner(int pin_number)
const{
```

In the file account.cxx delete or comment out the following line:

```
// Original Version
void Account::newOwner(Person *NewOwner) {
```

and add or un-comment the following line in its place:

```
// ODMG Persistent Version
void Account::newOwner(d_Ref<Person> &NewOwner) {
```

In the file `account.cxx` delete or comment out the following line:

```
// Original Version
void Account::insertEvent(Event* event) {
```

and add or un-comment the following line in its place:

```
// ODMG Persistent Version
void Account::insertEvent(const d_Ref<Event> &event) {
```

- ***atm.hxx file***

In the file `atm.hxx` again you only need to change the method signatures. Make the changes outlined below to `atm.hxx`.

In the ATM class definition in the file `atm.hxx` delete or comment out the following lines:

```
// Original Version
Account *newAccount(int account_number, ...);
Account *lookupAccount(int account_number) const;
```

and add or un-comment the following lines in their place:

```
// ODMG Persistent Version
d_Ref<Account> newAccount(int account_number, ...);
d_Ref<Account> lookupAccount(int account_number) const;
```

Again change the corresponding signatures in the method definitions.

- ***atm.cxx file***

In the file `atm.cxx` delete or comment out the following line:

```
// Original Version
Account * ATM::lookupAccount(int account_number) const
```

and add or un-comment the following line in its place

```
// ODMG Persistent Version
d_Ref<Account> ATM::lookupAccount(int
    account_number) const
```

In file `atm.cxx` delete or comment out the following line:

```
// Original Version
Account * ATM::newAccount(int account_number, ...) {
```

and add or un-comment the following line in its place

```
// ODMG Persistent Version
d_Ref<Account> ATM::newAccount(int account_number,
    ...);
```

Change C++ pointers to persistent pointers

- ***terminal.hxx file***

In file `terminal.hxx` in the class `Terminal` delete or comment out the following lines:

```
// Original Version
Account *displaySelectAccount(const Person* customer)
    const;
void displayPerson(const Person *customer) const;
```

and add or un-comment the following line in their place

```
// ODMG Persistent Version
d_Ref<Account> displaySelectAccount(const d_Ref<Person>
    &customer) const;
void displayPerson(const d_Ref<Person> &customer)
    const;
```

Once again change the corresponding method definitions.

- ***terminal.cxx file***

In file `terminal.cxx` delete or comment out the following line:

```
// Original Version
Account* Terminal::displaySelectAccount(const Person*
    customer) const{
```

and add or un-comment the following line in its place

```
// ODMG Persistent Version
d_Ref<Account> Terminal::displaySelectAccount(const
    d_Ref<Person>& customer) const {
```

In file `terminal.cxx` delete or comment out the following line:

```
// Original Version
void Terminal::displayPerson(const Person *customer)
    const {
```

and add or un-comment the following line in its place

```
// ODMG Persistent Version
void Terminal::displayPerson(const d_Ref<Person>&
    customer) const{
```

2.3 Include files

You must add the following include files to the application. These include files contain the objects, classes and functions which provide the interface between the application and the database.

The atm.hxx file is included by every source file so we will add the include lines to that file.

- ***atm.hxx file***

In the file atm.hxx add or un-comment the following lines

```
// ODMG Persistent Version
// Contains d_Session, d_Database and d_Transaction
  classes
#include "o2lib_CC.hxx
// Contains templates for d_Ref, d_Set, d_List, ...
#include "o2template_CC.hxx"
// Contains OQL interface
#include "OQL_CC.hxx"
```

Connect to the database

2.4 Connect to the database

The ODMG C++ binding contains a set of classes for database management. The `d_Session` class sets up communications between a C++ application and a database server. The `d_Database` class is used to create, destroy, open and close a database. The `d_Database` class can also be used to create and destroy names in the database as well as associating C++ variables to names. The `d_Transaction` class starts and ends transactions with the database. You need to add instances of all of these classes to the example application.

The `d_Session` class has a `begin` method to establish communication with the database server. The following line in the source code shows the call to `begin`:

```
session.set_default_env();
if (session.begin(argc,argv) {
```

The parameters passed to `begin` are the `argc` and `argv` parameters passed to the main routine. The `set_default_env` method sets the `O2 System` name, the name of the machine that the `O2 Server` is running on and the location of the `O2 software`.

The `d_Session` class also has an `end` method. This method closes the connection with the database. The following line from the source code shows the call to `end`:

```
session.end();
```

After connecting to the database server the application must specify which database to open. The following line shows the call need to open the `bankBase` database.

```
database_g->open("bankBase");
```

It is possible to open and close multiple databases in a single session. The following line show the call needed to close the database

```
database_g->close();
```

2.5 Run a transaction

In order to modify data in the database you need to be in a transaction.

The transaction mechanism protects an application against failure and ensures that data is consistent in a multi-user environment.

Inside a transaction, O₂ provides an isolation property by locking the objects. Consequently, in certain circumstances a transaction waits until another transaction commits.

To minimize the wait time, we declare a transaction only in the methods that modify the data.

- ***account.cxx file***

In the method `Account::deposit` add or un-comment the following lines:

```
d_Transaction transaction;
transaction.begin();
...
transaction.validate();
```

In the method `Account::withdraw` add or un-comment the following lines:

```
d_Transaction transaction;
transaction.begin();
...
transaction.validate();
```

- ***atm.cxx file***

In the method `ATM::initializeAccounts()` add or un-comment the following lines:

```
d_Transaction transaction;
transaction.begin();
...
transaction.validate();
```

Create names

2.6 Create names

You will need to create names in the database. A name is the entry point for the application to the database. You can create a single name for the ATM object, ATM_MACHINE and then access all persistent data by traversing from the named ATM object. Since the name needs to be created only once you can create it during initialization. The `set_object_name` method of the `d_Database` class is used to create the name. The following line shows the call needed to create the ATM_MACHINE name

```
database_g-> set_object_name(atm_machine,  
    "ATM_MACHINE");
```

Once the name is created you access the name through a persistent pointer. You can associate a persistent pointer with a named object by transmitting the name as an argument to the constructor. For example:

```
d_Ref<ATM> machine("ATM_MACHINE");
```

The relationship can also be established by using the `d_Database::lookup_object` method. We will use the `lookup_object` method for the ATM application since the persistent pointer is declared before the name has been created. The following line shows the call needed to lookup the name ATM_MACHINE

```
atm_machine->lookup_object("ATM_MACHINE");
```

2.7 Modify source code

We need to create an instance of `d_Database` in global scope so that we can create new objects in the database in any function or method.

Near line 80 in `main.cxx` add or un-comment the following global declaration.

- ***main.cxx file***

In the file `main.cxx` near line 80 add or un-comment the following global declaration.

```
d_Database* database_g = new d_Database;
```

We also need to add a declaration so that other files can access the global variable. We will add the declaration to the file `atm.hxx` since it is included by every source file.

- ***atm.hxx file***

In file `atm.hxx` add or un-comment the following lines:

```
// ODMG Persistent Version
// Global database pointer for creating objects in the
  database
extern d_Database* database_g;
```

You can then add the code to establish a connection with the database, open the bankBase database, start the transaction, create the names in the database, end the transaction, close the bankBase and finally close the connection with the database.

In the class `ATM` add or un-comment the following line:

```
void d_activate {term = new Terminal;}
```

You must initialize the temporary attribute `term` when the persistent object of the class `ATM` is read from the database for the first time.

- ***main.cxx file***

In the file `main.cxx` delete or comment out the following lines:

```
// Original Version
ATM* atm_machine;
atm_machine->initializeAccounts();
atm_machine = new ATM;
atm_machine->run();
```

and add or un-comment the following lines in their place, note that the commented out portion begins with `/* BEGIN PERSISTENT VERSION`

Modify source code

and ends with "END PERSISTENT VERSION */" instead of the "/*" comments:

```
// ODMG Persistent Version
/* BEGIN PERSISTENT VERSION */
d_Session session;
d_Transaction transaction;
// ODMG Persistent Version
d_Ref<ATM> atm_machine;
session.set_default_env();
if (session.begin(argc,argv)) {
    cerr<<"ERROR: Unable to connect to System "<<
    getenv("O2SYSTEM")<<endl;
    cerr << "Check O2SYSTEM, O2SERVER & O2HOME
    Environment Vars"<<endl;
    exit(1);
}

if ( argc > 1 && !strcmp(argv[1],"-initialize")) {
    transaction.begin();
    database_g->create("bankBase","bankSchema");
    database_g->open("bankBase");
if ( database_g->look_up_object("ATM_MACHINE")) {
    database_g->rename_object("ATM_MACHINE",0);
    database_g->garbage("bankBase");
}
    atm_machine = new(database_g) ATM;
    database_g->set_object_name
    (atm_machine,"ATM_MACHINE");
    transaction.validate();
    machine->initializeAccounts();
} else {
    database_g->open("bankBase");
    atm_machine =
    database_g->lookup_object("ATM_MACHINE");
}

    if (! atm_machine) {
    cerr << "There is no ATM in the database" << endl;
    exit(0);
}

atm_machine->run();

database_g->close();
session.end();
/* END PERSISTENT VERSION */
```

Now you are ready to create objects in the database.

2.8 Create persistent objects

In the ODMG binding, persistence is specified at object creation time. If you want to create a persistent object you transmit a `d_Database` pointer to the new operator of the object. Since all persistent data is accessed from names, each newly created persistent object should be attainable from a name. An object can be associated directly with a name or an object can be indirectly attainable from a name through persistent pointers.

The allocation statements in the example application must be modified to create persistent objects instead of transient ones. Some local variable declarations which are pointers to persistent objects need to be changed to persistent pointers. This ensures that persistent objects are kept in the C++ cache. The database will swap persistent objects out of the cache that are no longer pointed to by a persistent pointer.

- ***account.cxx file***

In file `account.cxx` in the method `Account::deposit` delete or comment out the following lines:

```
// Original Version
depositEvent* event = new depositEvent(machine_g);
```

and add or un-comment the following lines in their place

```
// ODMG Persistent Version
d_Ref<depositEvent> event = new(database_g)
    depositEvent(machine_g);
```

In file `account.cxx` in the method `Account::withdraw` delete or comment out the following lines:

```
// Original Version
withdrawEvent* event = new withdrawEvent(machine_g);
```

and add or un-comment the following lines in their place

```
d_Ref<withdrawEvent> event = new(database_g)
    withdrawEvent(machine_g);
```

Create persistent objects

- *atm.cxx file*

In file atm.cxx in the method ATM::run delete or comment out the following lines:

```
// Original Version
Account * current_account;
Person * current_customer
```

and add or un-comment the following lines in their place

```
// ODMG Persistent Version
d_Ref<Account> current_account;
d_Ref<Person> current_customer;
```

In file atm.cxx in the constructor ATM::ATM() delete or comment out the following lines:

```
// Original Version
location = new Address("3600_BAYSHORE", "PALO
ALTO", "CA", "94043");
```

and add or un-comment the following lines in their place

```
// ODMG Persistent Version
location = new(database_g) Address("3600_BAYSHORE",
"PALO_ALTO", "CA", "94303");
```

In file atm.cxx in the method ATM::lookupAccount(int account_number) delete or comment out the following lines:

```
// Original Version
Account *target_account = (Account*) NULL;
```

and add or un-comment the following lines in their place

```
// ODMG Persistent Version
d_Ref<Account> target_account = (Account*)NULL;
```

In file atm.cxx in the method ATM::initializeAccounts() delete or comment out the following lines:

```
// Original Version
Person* current_person;
...
Account* new_account = new Account;
inFile >> * new_account;
...
// Original Version
current_person = new Person;
inFile >> * current_person;
```

and add or un-comment the following lines in their place

```
// ODMG Persistent Version
d_Ref<Person> current_person;
...
d_Ref<Account> new_account = new(database_g) Account;
inFile >> * (new_account.ptr());
...
// ODMG Persistent Version
current_person = new(database_g) Person;
inFile >> * (current_person.ptr());
```

In file atm.cxx in the method ATM:newAccount(int account_number, ...) delete or comment out the following lines:

```
// Original Version
Account* new_account = new Account(account_number,
    description, balance);
```

and add or un-comment the following lines in their place

```
// ODMG Persistent Version
d_Ref<Account> new_account = new(database_g) Account
    (account_number, description, balance);
```

Since the ATM object is named and all persistent objects are associated with the ATM by persistent pointers no additional names need to be created.

Just as with a typical C++ program you can explicitly delete objects from the database when they are no longer needed. However, O₂ provides a garbage function to deal with data that is not deleted by the application.

In file account.cxx in the operator >> (ifstream&, Person&) delete or comment out the following lines:

```
p.address = new Address;
ifs >> *p.address
...
Account* current_account;
```

and add or un-comment the following lines in their place

```
p.address = new(database_g) Address;
ifs >> *(p.address.ptr());
...
d_Ref<Account> current_account;
```

Build and Run the Application

2.9 Build and Run the Application

The sample application is now ported to the O2 System.

To compile, link and run the application enter the following command:

```
On Unix,  
> make clean  
> make
```

```
On Windows NT,  
> nmake clean  
> nmake
```

If necessary, correct any typing errors you might have made that cause syntax errors and repeat the make.

Once it has been compiled and linked run the program through the following commands:

Start the application

```
> bank -initialize
```

Enter the Account number 11111 at the prompt and press enter. The following will be displayed.

```
Welcome to the Bank of O2
```

```
Enter Account Number, 0 to exit
```

```
> 11111
```

At the personal ID prompt enter the number 9999 and press enter.

```
Enter PIN Number
```

```
> 9999
```

You are now logged into the system so you can carry out transactions with the accounts available.

Now exit the program

2.10 Replace arrays by collection classes

ODMG-93 introduces predefined generic collection classes. An O₂ ODMG database supports all ODMG collection classes. The collections supported are:

d_Set - unordered collection of elements with no duplicates.

d_Bag - unordered collection of elements that allows for duplicates.

d_List - an ordered collection of elements that allows for duplicates.

d_Varray - a one dimensional array of varying length.

Collection classes use standard C++ template classes to provide support for elements of an arbitrary type. These classes have methods for various operations, such as insert, delete, append, union, intersection, etc. There is also an iterator template class for defining iterators for any type of collection. The iterators are used to scan a collection sequentially returning each matching element from the collection.

The use of collection classes in C++ programs makes the programs more scalable. Our example application uses arrays to store collections. The size of the arrays is set at the time the application is compiled. The ODMG C++ collection classes are not of a fixed size and can expand as the application evolves.

Collection classes are tightly coupled to the O₂ database to provide transparent caching of large collections. When an application accesses a large ODMG collection only a portion of the collection is brought into application memory. As the application traverses the collection the rest of the collection is brought in. Accessing data in collections can be speeded up by creating indexes on a collection.

You can now replace the arrays in the example application with collection classes. The ATM class contains the list of accounts. This is known as the `accounts_list`. The list of accounts is implemented using a standard C++ array. Since there is no requirement to maintain any ordering on the list of accounts we will use a `d_Set` to replace the array. First we need to modify the declaration of the account list.

- ***atm.hxx file***

In the ATM class declaration in the file `atm.hxx` comment out or delete the following lines:

```
// Original Version
Account *accounts_list[MAX_ACCOUNTS];
```

and add or un-comment the following lines in their place:

Replace arrays by collection classes

```
// ODMG Collection Class Replacement
d_Set< d_Ref<Account> > accounts_list;
```

Note the use of a space between > > to differentiate between a nested template and the shift operator.

Now modify the code where the `accounts_list` is being used. Only the `lookupAccount` method accesses the `accounts_list`. Remove the initialization code that is no longer needed then replace the for loop used in the method with the use of an iterator. First create the iterator from the `accounts_list` collection and then traverse the collection using the iterator.

- ***atm.cxx file***

In the constructor `ATM::ATM` in the file `atm.cxx` delete or comment out the following lines:

```
// Original Version
//for(int i=0;i<MAX_ACCOUNTS;i++) accounts_list[i] =
NULL;
```

In the method `ATM::lookupAccount` in the file `atm.cxx` delete or comment out the following lines:

```
// Original Version
for(int i=0; i < MAX_ACCOUNTS; i++) {
    if ( accounts_list[i] != NULL &&
        accounts_list[i]->getNumber() == account_number)
    {
        target_account = accounts_list[i];
        break;
    }
}
```

and add or un-comment the following lines in their place:

```
// ODMG Collection Class Replacement
d_Ref<Account> search_account;
d_Iterator< d_Ref<Account> > iterator =
accounts_list.create_iterator();
while ( iterator.next(search_account) ) {
    if (search_account->getNumber() ==
        account_number)
        target_account = search_account;
    break;
}
```

Modify the account creation routine to append the new account onto the list by making the following modification.

In the method `ATM::initializeAccounts` in the file `atm.cxx` delete or comment out the following lines:

```
// Original Version
int i = 0;
while (accounts_list[i] != NULL)
    i++;
accounts_list[i] = new_account;
```

and add or un-comment the following lines in their place:

```
// ODMG Collection Class Version
accounts_list.insert_element( new_account );
```

In the method `ATM::newAccount` in the file `atm.cxx` delete or comment out the following lines:

```
// Original Version
int i = 0;
while (accounts_list[i] != NULL)
    i++;
accounts_list[i] = new_account;
```

and add or un-comment the following lines in their place:

```
// ODMG Collection Class Version
accounts_list.insert_element( new_account );
```

Replace the array of pointers to events in the `Account` class to use an ODMG collection class. The order of the events is significant so use a `d_List` to maintain the order. Collection classes support polymorphism through class hierarchy therefore the `d_List<d_Ref<Event> >` may contain elements of the subclasses of `Event`, namely `depositEvent` and `withdrawEvent`. Replace the array of `Person` by a relationship of type `d_Set<d_Ref<Person> >` to maintain referential integrity with `Person` and change its inverse link, `accounts` in the `Person` class, to a relationship of the type `d_List<d_Ref<Account> >`. Make the following changes to the declaration of the `events_completed`, `owners` in the `Account` class and `accounts` in the `Person` class.

Replace arrays by collection classes

- **account.hxx file**

In the account class declaration in the file account.hxx delete or comment out the following lines:

```
// Original Version
    Person* owners[CUSTOMERS_PER_ACCOUNT];
    ...
    Event *events_completed[MAX_EVENTS];
```

Add or un-comment the following lines in their place:

```
// ODMG Collection Class Replacement
    d_Set<d_Ref<Person> > owners inverse accounts;
    ...
    d_List< d_Ref<Event> > events_completed;
```

In the Person class declaration delete or comment out the following lines:

```
// Original Version
    Account* accounts[ACCOUNTS_PER_CUST]
```

Add or un-comment the following lines in their place:

```
// ODMG Collection Class Replacement
    d_List<d_Ref<Account> > accounts inverse owners
```

You also need to change the way the events_completed and the owners are used. Stop the events_completed and owners being initialized to NULL in both the Account constructors and add the initializer for the relationship.

- **account.cxx file**

In the constructor Account::Account in the file account.cxx delete or comment out the following lines

```
// Original Version
int i;
for (i=0;i<MAX_EVENTS;i++) events_completed[i] = NULL;
for (i=0;i<CUSTOMERS_PER_ACCOUNT;i++) owners=NULL;
```

and add the initializer for the relationship:

```
Account::Account() : owners(this)
```

In the constructor Account::Account(int num, char* desc, float bal) in the file account.cxx delete or comment out the following lines

```
// Original Version
    int i;
    for (i=0;i<MAX_EVENTS;i++) events_completed[i] =
    NULL;
```

```
for (i=0;i<CUSTOMERS_PER_ACCOUNT;i++) owners=NULL;
```

and add the initializer for the relationship:

```
Account::Account(int num, char* desc, float bal) :
owners(this)
```

Now you need to change the showEvents method which traverses the events_list. Instead of using an iterator you can traverse the collection using a C++ construct.

In the method Person::showEvents in the file account.cxx delete or un-comment the following lines:

```
// Original Version
for (int i=0; i < ACCOUNTS_PER_CUST; i++){
...
for (int j=0; j<MAX_EVENTS; j++) {
```

and add or un-comment the following lines:

```
// ODMG Collection Class Version
int nb = accounts.cardinality();
for (int i=0; i < nb; i++) {
...
for (int j=0; j < accounts[i]-
>events_completed.cardinality();
j++) {
```

Modify the code which inserts objects into the collection.

In the method Account::insertEvent in the file account.cxx remove or comment out the following lines:

```
// Original Version
static int i = 0;
events_completed[i%MAX_EVENTS] = event;
i++;
```

and un-comment or add the following lines:

```
// ODMG Collection Class Version
events_completed.insert_element_last(event);
```

In the method Account::lookupOwner in the file account.cxx remove or comment out the following lines:

```
// Original Version
int j;
for (int j=0; j < CUSTOMERS_PER_ACCOUNT; j++) {
if owners[j] && owners[j]->personal_id ==
pin_number) {
return owners[j];
```

Replace arrays by collection classes

and un-comment or add the following lines:

```
d_Ref<Person> search_person;
d_Iterator< d_Ref<Person> > iterator =
    owners.create_iterator();
while ( iterator.next(search_person) ) {
    if (search_person->personal_id == pin_number) {
        return search_person;
    }
}
```

In the method Account::newOwner in the file account.cxx remove or comment out the following lines:

```
// Original Version
int i = 0;
while (owners[i]) i++;
owners[i] = NewOwner;
```

and un-comment or add the following lines:

```
// ODMG Collection Class Version
owners.insert_element(NewOwner);
```

In the constructor Person::Person() in the file account.cxx remove or comment out the following lines:

```
// Original Version
Person::Person() {
    for (int i = 0; i<ACCOUNTS_PER_CUST; i++)
        accounts[i] = NULL;
```

and un-comment or add the following lines:

```
// ODMG Collection Class Version
Person::Person() : accounts(this) {
```

In the input file stream operator in the file account.cxx remove or comment out the following lines:

```
// Original Version
p.accounts[account_count] = current_account;
p.accounts[account_count]->newOwner(&p);
```

and un-comment or add the following lines:

```
// ODMG Collection Class Version
p.accounts.insert_element (current_account);
```

In the method `Terminal::displaySelectAccount` in the file `terminal.cxx` remove or comment out the following lines:

```
// Original Version
while (i < ACCOUNTS_PER_CUST && customer->accounts[i])
{
```

and un-comment or add the following lines:

```
// ODMG Collection Class Version
int nb = customer->accounts.cardinality();
while (i < nb && customer->accounts[i]) {
```

Build and Run the Application

2.11 Build and Run the Application

Now you have ported the automated teller machine application to use ODMG collection classes. The next step is to compile and link the modified application.

Run the following command to compile and link the application:

On Unix,
> **make**

On Windows NT,
> **nmake**

If necessary, correct any typing errors you might have made that cause syntax errors and repeat the make.

Now run the application, using the initialize option to reread the data from the files. You need to reinitialize the database because you modified the schema when you added the collection classes.

> **bank -initialize**

Enter the Account number 11111 at the prompt and press enter. The following will be displayed.

Welcome to the Bank of O2

Enter Account Number, 0 to exit

> 11111

At the personal ID prompt enter the number 9999 and press enter.

Enter PIN Number

> 9999

You are now logged into the system so you can carry out transactions with the accounts available.

To exit the program enter 0 at the account number prompt.

3

OQL

This chapter describes OQL.

It contains the following sections:

- [The O2 ODMG Object Query Language](#)
- [Embedding OQL in C++](#)
- [Build and Run the Application](#)

3.1 The O₂ ODMG Object Query Language

ODMG-93 introduces an object query language (OQL). OQL is an SQL-style language that allows for easy access to objects. Similar in use to SQL, OQL is an essential tool for developing database applications.

- **Query optimization**

The system optimizes your queries. Tried and tested optimization techniques are used to evaluate an OQL query. For example, OQL uses appropriate indexes to reduce the amount of data to be filtered. It factorizes common sub-expressions, discovers which expressions can be computed once outside an iteration, and makes selections before starting an inner iteration.

- **Logical and physical independence**

OQL differs from standard programming languages in that the execution of an OQL query can be improved dramatically without modifying the query itself. This is done by declaring new physical data structures or new indexing or clustering strategies. This reduces response time.

This sort of modification in a purely imperative language like C++ requires that an algorithm be completely rewritten, as it makes use of physical structures.

Interactive ad hoc queries

A database user cannot be asked to write, compile, link, edit and debug a C++ program to make simple queries. OQL can be used directly as a stand-alone query interpreter. Its syntax is simple and flexible. For someone familiar with SQL, OQL can be learned in a short time.

Some examples are given to demonstrate the most useful features of OQL. You can execute them as ad hoc queries with o2dsa. First start the o2dsa interpreter:

```
> o2dsa_shell
```

Now set the base and put the interpreter into query mode

```
set base bankBase
```

```
^D
```

```
query
```

```
^D
```

You can type the queries into the Input window and then execute them by typing a new line followed by ctrl D.

Accessing data

Data is accessed in the database through a named object. All queries start from a named object and navigate through pointers to reach relevant data. To do this in OQL use “.” or “->”. These allow you to enter complex objects, as well as to follow simple relationship paths. For instance, if you need to know the name of the street that an ATM machine is on, the query is as follows:

```
ATM_MACHINE.location.street
^D
```

This query starts from an ATM object, traverses an Address object and enters the address object to get the name of the street.

This example treated a one-to-one relationship. Let us look at multiple relationships. Assume we want to retrieve the account numbers. We cannot write `ATM_MACHINE.accounts_list.number` because `accounts_list` is a set of references. The result should be a collection of numbers but we need an unambiguous form of notation to traverse this multiple relationship. We can use the select-from-where clause to handle collections just as in SQL.

```
select a.number
from a in ATM_MACHINE.accounts_list
^D
```

Now you can navigate from one object to any other object following any relationship path and enter their complex subvalues.

Selecting data

OQL supports a where clause to select data which matches the predicate of the clause. For instance, let us suppose that we want to restrict the previous search to those accounts with a balance greater than zero.

```
select a.number
from a in ATM_MACHINE.accounts_list
where a.balance > 0
^D
```

- *Join*

In the from clause, collections that are not directly related can also be declared. As in SQL, this allows you to compute joins between these collections. For instance, to identify customers whose PIN number is the same as their account number execute the following query.

```
select a
from a in ATM_MACHINE.accounts_list,
     p in (select distinct b
           from c in ATM_MACHINE.accounts_list,
                b in c->owners)
where a.number = 10000 + p.personal_id / 9
^D
```

Manipulating complex values

A major difference between OQL and SQL is that an object query language may manipulate complex values. OQL can create complex values as a final result, or during the query as an intermediate result. To build a complex value, OQL uses the constructors struct, set, bag, list, and array. For example, to obtain the account number and balance for every account, execute the following query:

```
select struct(num: a.number, bal: a.balance)
from a in ATM_MACHINE.accounts_list
^D
```

OQL can also create complex objects. For this purpose, it uses the name of a class as a constructor. Attributes of the object of this class can be initialized explicitly by any valid expression.

Manipulating polymorphic collections

A major contribution of object orientation is that it allows you to manipulate polymorphic collections. Another is the ability to carry out generic actions on the elements of these collections. This is made possible by a late binding mechanism,

For instance, the events_completed list is a polymorphic list since it contains objects of the depositEvent and withdrawEvent classes. When a polymorphic collection is filtered its elements are known to be of that class. This means that properties of a subclass (attribute or method) cannot be applied to such an element, except in two important cases: late binding to a method, or explicit class indication (cast). The OQL language supports both late binding of methods and explicit casting.

- **High level constructs**

OQL, like SQL, provides high-level operators that allow you to sort, group, or aggregate objects or to gather statistics, all of which would require considerably more programming in C++.

Composing operators

OQL is a purely functional language: all operators can be composed freely as long as the type system is respected. The operators offered in OQL include the set operators (union, intersect, except), the universal operator (for all) and the existential quantifiers (exists), the sort and group by operators and the aggregate operators (count, sum, min, max and average).

Now exit the query interpreter by executing the following commands:

```
quit
```

```
^D
```

Now exit the o2dsa_shell program by executing the following commands:

```
quit
```

```
^D
```

3.2 Embedding OQL in C++

Embedded in a C++ program, OQL reduces program writing. OQL is powerful enough to contain a long C++ program in one statement. It can also improve the performance of an application because the query language is optimized.

As OQL respects the ODMG model, it has the same type system as C++ and can query objects and collections computed by C++ and passed on to OQL as parameters. OQL then delivers a result, which is put directly into C++ variables with no conversion. This solves impedance mismatch problems that make embedded SQL more difficult to use.

To invoke a query in C++ use the following function:

```
int d_oql_execute(d_OQL_Query &q, T &result)
```

`d_oql_execute` takes a query in the `d_OQL_Query` parameter, parses and evaluates the query at runtime and returns the result.

To construct a query you create an instance of the `d_OQL_Query` class and initialize it with a character string containing the query. In the query construction, you can give parameters in the form of \$1, \$2, etc. After creating the query you associate arguments to these parameters using the `<<` operators.

In the sample application you can add queries to replace some of the C++ code.

You can change the `lookupAccount` method to use a query instead of while loops. The `lookupAccount` method returns a pointer to an account which matches the account number given as a parameter. The query which replaces the while loop is

```
select x
from x in accounts_list
where a->getNumber = account_number
```

Embedding OQL in C++

To replace the existing C++ code with a query make the following changes.

In the method `ATM::lookupAccount` in the file `atm.cxx` remove or comment out the following lines:

```
// ODMG Collection Class Replacement
d_Ref<Account> search_account;
d_Iterator< d_Ref<Account> > iterator =
    accounts_list.create_iterator();
while ( iterator.next(search_account) ) {
    if (search_account->getNumber() == account_number) {
        target_account = search_account;
        break;
    }
}
```

Add or un-comment the following lines in their place:

```
// OQL Query Replacement
d_OQL_Query query ("element(select x from x in $1 where
    x->getNumber = $2)");
query << account_list << account_number;
d_oql_execute (query, target_account);
```

3.3 Build and Run the Application

Now you have ported the automated teller machine application to use an OQL query. The `lookupAccount` method is executed as part of the login process. The next step is to compile and link the modified application.

Run the following command to compile and link the application:

On Unix,
> `make`

On Windows NT,
> `nmake`

If necessary, correct any typing errors you might have made that cause syntax errors and repeat the make.

Now run the application.

> `odmg -tutorial`

Enter the Account number 11111 at the prompt and press enter. The following will be displayed.

```
Welcome to the Bank of O2
```

```
Enter Account Number, 0 to exit
```

```
> 11111
```

At the personal ID prompt enter the number 9999 and press enter.

```
Enter PIN Number
```

```
> 9999
```

You are now logged into the system so you can carry out transactions with the accounts available.

To exit the program enter 0 at the account number prompt.

This is the end of the O₂ODMG C++ tutorial.

You now know how to port applications to O₂ and can integrate your C++ program in the O₂ database system, for better and more efficient data management.