Using Objectivity/C++
Version 4
Using Objectivity/C++

Version 4, Release 4.0, July 14, 1997

The information in this document is subject to change without notice. Objectivity, Inc. assumes no responsibility for any errors that may appear in this document.

Copyright 1996 Objectivity, Inc. All rights reserved. This document may not be copied, photocopied, reproduced, translated or converted to any electronic or machine-readable form in whole or in part without prior written approval of Objectivity, Inc.

Objectivity and Objectivity/DB are registered trademarks of Objectivity, Inc. Objectivity/DB Fault Tolerant Option, Objectivity/FTO, Objectivity/DB Data Replication Option, Objectivity/DRO, Objectivity/C++, Objectivity/C++ Data Definition Language, Objectivity/DDL, Objectivity/C++ Tools.h++, Objectivity/Tools.h++, Objectivity/Smalltalk, Objectivity/SQL++, Objectivity/SQL++ ODBC Driver, and Objectivity/ODBC are trademarks of Objectivity, Inc. Other brands and products are registered trademarks or trademarks of their respective holders.

ODMG information in this document is based in whole or in part on material from The Object Database Standard: ODMG-93 Release 1.1, edited by R.G.G. Cattell, and is reprinted with permission of Morgan Kaufmann Publishers. Copyright 1994 by Morgan Kaufmann Publishers.

The software and information contained herein are proprietary to, and comprise valuable trade secrets of, Objectivity, Inc., which intends to preserve as trade secrets such software and information. This software is furnished pursuant to a written license agreement and may be used, copied, transmitted, and stored only in accordance with the terms of such license and with the inclusion of the above copyright notice. This software and information or any other copies thereof may not be provided or otherwise made available to any other person.

U. S. Government Restricted Rights: Use, duplication or disclosure of the software or other information by the U. S. Government or any unit or agency thereof is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and the Government is acquiring only restricted rights in the software and limited rights in any technical data provided (as such terms are defined in such clause of the DFARS). If the software or other information is supplied to any unit or agency of the U. S. other than the Department of Defense, the Government’s rights will be as defined in clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in clause 18-52.227-86 (d) of the NASA Supplement to the FAR.
Contents

About This Book  xxiii

- Audience  xxiii
- Organization  xxiii

Chapter 1  Introduction to Objectivity/DB Programming  1-1

- Basics  1-1
  - Compiling and Linking C++ Applications  1-1
  - Logical Storage Classes  1-1
  - Persistence  1-3
  - Object Identifiers (OIDs) for Accessing Objects  1-3
  - Conventions for Return Values and Naming  1-4
- Schema Development Overview  1-6
- Application Development Overview  1-6
  - Using Virtual Member Functions  1-8
  - Set Environment Variables  1-8
  - Create a Makefile  1-9
  - Build and Run an Application  1-9
  - View the Federated Database  1-10
- Working with Objectivity/DB Data  1-10
- Objectivity/DB Features to Enhance Performance  1-11
  - Variable-Size Arrays  1-11
  - Strings  1-14
  - Associations  1-17
  - Iterators  1-18
  - System Names  1-19
  - Dictionaries  1-19
  - Indexes  1-21
## Chapter 2

**Transaction Model**

- Using Multiple Transactions 2-3
- Initializing Database Services 2-4
- Specifying Memory Size for Large Objects 2-5
- Activating Hot Mode 2-6
- Starting a Transaction 2-7
- Committing a Transaction 2-9
  - `commit` 2-9
  - `commitAndHold` 2-12
- Aborting a Transaction 2-14
- Checking for an Active Transaction 2-16
- Improving Concurrency 2-17
- Concurrent Transaction Support 2-17

## Chapter 3

**Creating and Deleting Objects**

- Creating Persistent Objects 3-1
  - Declaring Persistent Classes 3-2
- Creating a Federated Database 3-2
- Creating Databases 3-3
  - Using `new` 3-3
  - Using `ooReplace` 3-6
- Creating Containers 3-7
  - Hashed Containers 3-7
  - Using `new` 3-8
  - Creating a Batch of Containers 3-11
- Creating Basic Objects 3-13
  - Using `new` 3-13
- Creating Transient Objects 3-16
- Deleting a Federated Database 3-17
- Deleting Databases, Containers, and Basic Objects 3-18
  - Deleting Persistent Objects 3-18
  - Deleting Transient Objects 3-21
- Example Using `new` and `delete` 3-22
Chapter 4  

**Accessing Objects** 4-1

Object Identifiers 4-1
Access Mechanisms 4-2
Using Object References and Handles 4-2
   Using the ooRef and ooShortRef Classes 4-3
   Using the Member Function operator -> 4-3
   Using isValid 4-5
Accessing Persistent Objects 4-5
Setting an Object Reference or Handle 4-7
Class Compatibility and Casting 4-7
Member Functions and Operators 4-8
Handle Registration 4-13
Managing the Handle Stack 4-14
Using Pointers with Object References or Handles 4-14
   Assigning a Pointer to an Object Reference or Handle 4-14
   Initializing an Object Reference or Handle with a Pointer 4-15
   Using ooThis Instead of Pointer this 4-16
   Passing a Pointer as an Object Reference or Handle 4-17
   Object Reference Example 4-18
Accessing Containers 4-19
Accessing Databases 4-20

Chapter 5  

**Opening and Closing Objects** 5-1

Opening Objects 5-1
   Read and Update Access Modes 5-2
   Opening a Federated Database 5-3
   Opening a Database 5-5
   Opening a Container or Basic Object 5-6
   Determining the Current Open Mode 5-8
   Upgrading Access from Read to Update 5-9
   Using ooUpdate Within a Member Function 5-9
Closing Objects 5-10
Chapter 6  Copying and Moving Basic Objects  6-1

Copying a Basic Object  6-1
Using copy  6-1
Using ooCopyInit  6-4

Moving a Basic Object  6-8
Using move  6-8
Using ooPreMoveInit and ooPostMoveInit  6-11

Chapter 7  Using Associations  7-1

Directionality of Associations  7-1
Number of Associations per Object  7-3
Default Association Ordering  7-3
Association Cardinality  7-3
Using One-to-One and Many-to-One Associations  7-4
Getting the Handle of the Object Associated on a to-One Link  7-4
Setting an Association on a to-One Link  7-6
Deleting an Association on a to-One Link  7-7
Using One-to-Many and Many-to-Many Associations  7-7
Initializing an Iterator for Associations on a to-Many Link  7-7
Adding an Association to a to-Many Link  7-8
Deleting Specified Associations from a to-Many Link  7-8
Deleting All Associations from a to-Many Link  7-9
Checking for the Existence of an Association  7-11
Composite Objects and Propagating Operations  7-11
Adding Information to Associations  7-13

Chapter 8  Naming Objects  8-1

Using System Names  8-1
Setting System Names  8-2
Obtaining an Object’s System Name  8-2
Object Lookup Using System Names  8-2
Checking for Object Existence  8-3
Chapter 9  Using Iterators  9-1

Declaring an Iterator  9-3
Initializing an Iterator  9-3
Using scan  9-4
Using \texttt{nameOfLink}  9-6
Using contains  9-6
Using \texttt{getNameScope}  9-6
Using \texttt{getNameObj}  9-7
Getting an Iteration Handle  9-7
Accessing an Object In the Iteration  9-9
Iteration Techniques  9-10
Using the \texttt{while} Statement  9-10
Using the \texttt{for} Statement  9-10
Nesting Iterators  9-11
Iteration and Concurrency  9-12
Iterating on a Federated database  9-12
Iterating on a Database  9-13
Determining the Containment Hierarchy  9-13
Using the \texttt{contains} Member Function  9-13
Using the \texttt{containedIn} Member Function  9-16
Using Predicate Query  9-18
Using \texttt{scan} with a Conditional Predicate  9-18
Using \texttt{nameOfLink} with a Conditional Predicate  9-20
Using Indexes  9-21
Using \texttt{ooQuery}  9-24
User-Defined Operators  9-25
Chapter 10  
**Using Variable-Size Arrays**  
Using VArrays of Class ooVArray  
Declaring a VArray  
Implementing VArrays  
Creating a VArray  
VArray Element Operators and Member Functions  
Updating VArrays  
Resizing VArrays  
Extending VArrays  
VArray Assignment  
Using VArrays of Class ooTVArray  
Resizing ooTVArray VArrays  
Extending ooTVArray VArrays  
Assigning ooTVArray VArrays  
VArrays of Handles  

Chapter 11  
**Locking and Concurrency**  
Automatic Locking Rules  
Concurrency Rules  
Database and Federated Database Concurrency  
Container and Basic Object Concurrency  
Concurrency and Iterators  
Handling Lock Conflicts  
Activating Lock Waiting  
Deadlock Detection  
Multiple Readers, One Writer (MROW)  
Activating or Deactivating MROW for a Transaction  
Checking Whether an MROW Container Is Updated  
Refreshing the View of an MROW Container  
Summary of MROW and Non-MROW Concurrency
Explicit Locking  11-12
Two-Phase Locking  11-13
Lock Modes  11-13
Locking Granularity and Concurrency  11-14
Obtaining an Explicit Lock  11-15
Disabling Locking  11-17
Checking Objects Out and In
  Using checkout  11-19
  Using checkin  11-20

Chapter 12  Using the String Classes  12-1

Choosing a String Class  12-1
Class ooVString  12-2
  ooVString Constructors  12-2
  ooVString Member Functions  12-2
  ooVString Operators  12-3
Class ooString(N)  12-8
  ooString(N) Constructors  12-9
  ooString(N) Member Functions  12-10
  ooString(N) Operators  12-11

Chapter 13  Using the Map Dictionary Classes  13-1

Map Dictionary Classes  13-1
Including the Map Dictionary Header File  13-2
Class ooMap  13-2
  ooMap Constructor  13-2
  ooMap Member Functions  13-4
Advanced Techniques  13-11
  Class ooMapElem  13-14
  Class ooMapItr  13-15
  Referential Integrity  13-20
<table>
<thead>
<tr>
<th>Chapter 14</th>
<th>Using Indexes</th>
<th>14-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td></td>
<td>14-1</td>
</tr>
<tr>
<td>Building Applications That Use Indexes</td>
<td>14-2</td>
<td></td>
</tr>
<tr>
<td>Creating an Index</td>
<td>14-3</td>
<td></td>
</tr>
<tr>
<td>Creating a Key Description</td>
<td>14-3</td>
<td></td>
</tr>
<tr>
<td>Creating Key Fields</td>
<td>14-5</td>
<td></td>
</tr>
<tr>
<td>Adding a Key Field to the Key Description</td>
<td>14-9</td>
<td></td>
</tr>
<tr>
<td>Creating an Index from the Key Description</td>
<td>14-10</td>
<td></td>
</tr>
<tr>
<td>Verifying an Index Is Unique</td>
<td>14-11</td>
<td></td>
</tr>
<tr>
<td>Dropping an Index</td>
<td>14-12</td>
<td></td>
</tr>
<tr>
<td>Dropping an Index from the Key Description</td>
<td>14-12</td>
<td></td>
</tr>
<tr>
<td>Deleting All Indexes from a Key Description</td>
<td>14-13</td>
<td></td>
</tr>
<tr>
<td>Using the Indexes You Create</td>
<td>14-14</td>
<td></td>
</tr>
<tr>
<td>Concurrency of Index Operations</td>
<td>14-14</td>
<td></td>
</tr>
<tr>
<td>Consistency of Index Operations</td>
<td>14-14</td>
<td></td>
</tr>
<tr>
<td>Setting Index Modes</td>
<td>14-15</td>
<td></td>
</tr>
<tr>
<td>Explicitly Updating Indexes</td>
<td>14-16</td>
<td></td>
</tr>
<tr>
<td>Lookup Keys</td>
<td>14-17</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 15</th>
<th>Using Keyed Objects</th>
<th>15-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyed Object Procedure Summary</td>
<td>15-2</td>
<td></td>
</tr>
<tr>
<td>Initializing a Key Structure</td>
<td>15-3</td>
<td></td>
</tr>
<tr>
<td>Creating a Keyed Object</td>
<td>15-5</td>
<td></td>
</tr>
<tr>
<td>Locating a Keyed Object</td>
<td>15-6</td>
<td></td>
</tr>
<tr>
<td>Manipulating a Keyed Object</td>
<td>15-8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 16</th>
<th>Versioning</th>
<th>16-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Versioning Techniques</td>
<td>16-2</td>
<td></td>
</tr>
<tr>
<td>Version Genealogy</td>
<td>16-3</td>
<td></td>
</tr>
<tr>
<td>Enabling and Disabling Basic Versioning</td>
<td>16-5</td>
<td></td>
</tr>
<tr>
<td>Creating a Version</td>
<td>16-6</td>
<td></td>
</tr>
<tr>
<td>Specifying Copy Semantics</td>
<td>16-10</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 17  **Using Debug Mode**  17-1

Activating Debug Mode  17-1
Debug Options  17-2
  Data Verification  17-2
  Event Trace  17-3

Chapter 18  **Signal Handling**  18-1

System Signal Handling Issues  18-1
Signal Handling Example  18-2

Chapter 19  **Error Handling**  19-1

Using the Error Handling Facility  19-1
  The `ooStatus` Type  19-2
  Error Identifier Structure  19-2
  Error Message File  19-3
  Error Levels  19-4
  Error Flags  19-5
  Signaling an Error  19-7
Detecting an Error Condition  19-9
The Error Handler  19-9
The Message Handler  19-13

Chapter 20  Using Informational Functions  20-1
Printing an Object’s OID  20-1
Locating the Default Container  20-1
Obtaining a Class Type Number  20-2
Determining an Object’s Class  20-4
Getting the Number of Containers in a Database  20-6
Checking Virtual Table Pointers  20-6

Chapter 21  Monitoring and Tuning Performance  21-1
Tuning Guidelines  21-1
Tuning Concepts  21-2
Clustering  21-2
Variable-Size Arrays  21-2
Object References  21-3
String Classes  21-3
Dictionary Classes  21-4
Using Indexes  21-4
Tuning for Database Size  21-4
Minimize Use of Scope Naming  21-4
Use Dictionary Classes When Naming Many Objects  21-5
Minimize Number of Hash Pages  21-5
Minimize Number of Containers  21-5
Minimize Size of Containers  21-5
Minimize Growth Factor of Containers  21-6
Create Single Containers Using new Operator  21-6
Set Associations on a Basic Object at the Same Time  21-6
Use Unidirectional Instead of Bidirectional Associations  21-6
Use Object References Instead of Unidirectional Associations  21-6
Use Short Object References Instead of Object References  21-7
Use Fixed Arrays or String Classes Instead of VArrays 21-7
Delete Basic Objects Efficiently 21-8
Be Judicious in Creating Indexes 21-8
Tuning for Runtime Speed 21-9
Avoid Hash Overflows 21-9
Avoid Container (OC) Extensions 21-9
Find the Best Cache Size 21-10
Minimize the Size of Your Database 21-10
Cluster Objects Efficiently 21-11
Minimize Use of Scope Names and Keyed Objects 21-11
Use Indexes Instead of Scope Names or Keyed Objects 21-12
Explicitly Updating Indexes 21-12
Avoid Unnecessary Opening of Containers 21-12
Use ooNewConts When Creating Multiple Containers 21-12
Find the Best Page Size 21-13
Set Associations as Soon as Possible 21-13
Pass Object References or Handles by Reference Instead of Value 21-14
Avoid Constantly Resizing VArrays 21-14
Specify the Open Mode for Iterators 21-14
Use contains Instead of scan 21-14
Assign Explicit Handles When Using Object References 21-14
Minimize the Number of Indexed Objects 21-15
Use Dictionary Classes to Name and Locate Objects 21-15
Use the String Classes to Store String Data 21-16
Tuning for Concurrency 21-16
Do Not Explicitly Lock Objects 21-16
Consider Access Granularity 21-17
Use Unidirectional Associations 21-17
Make Container-Level Indexes Sharable 21-17
Obtaining and Interpreting Runtime Statistics 21-18
Obtaining Runtime Statistics Using ooRunStatus 21-18
Interpreting Objectivity/DB Internal Statistics 21-18
Chapter 22  **Object Conversion**  22-1

Basics  22-1
Using Deferred Mode  22-2
Using On-Demand Mode  22-3
  Procedure  22-3
Using Immediate Mode with an Upgrade Application  22-4
  Creating an Upgrade Application  22-4
  Upgrade Application Procedure  22-5
Using Conversion Functions  22-5
  Registering a Conversion Function  22-6
  Getting Primitive Data Member Values  22-6
  Setting Primitive Data Member Values  22-7
  Getting the Objects that Use a Specified Data Member  22-8
  Getting the Objects that Have a Specified Base Class  22-8
Example Upgrade Application with Conversion Functions  22-8
Performance Considerations  22-13
  Redirected Objects  22-13
  Read-Only Transactions  22-14
Deploying to End Users  22-14
Effect of Schema Changes on Indexes  22-16

Chapter 23  **Using the ODMG Interface**  23-1

Basics  23-1
  Terminology  23-3
  Template Requirements  23-4
  Enabling the ODMG Interface  23-4
Developing an ODMG Application  23-4
Example  23-5
Chapter 24  Administration Programming Tasks  24-1

Using the C++ Administration Interface  24-1
Federated Database Tasks  24-2
  Changing Federated Database Attributes  24-2
  Encoding and Decoding Schema  24-4
  Getting Federated Database Information  24-5
  Getting Transaction Information  24-6
  Tidying a Federated Database  24-7
Database Tasks  24-8
  Changing Database Attributes  24-8
  Getting Database Information  24-9
  Moving a Database File  24-10
  Tidying a Database  24-11
AMS Tasks  24-12
  Setting AMS Usage in an Objectivity/C++ Application  24-12
Automatic and Manual Recovery Tasks  24-13
  Enabling Automatic Recovery from Application Failures  24-13
  Creating a Recovery Application  24-13

Chapter 25  Using the Cursor Class  25-1

Architecture  25-1
  Usage  25-3
    Including the Header File  25-3
Class ooCursor  25-4
Declaring a Cursor  25-4
Implementing Cursors  25-4
ooCursor Constructors  25-4
Initializing the Cursor  25-5
Retrieving the Cursor Size  25-6
Checking for Query Completeness  25-6
Retrieving Objects from a Cursor  25-6
Resizing the Cursor  25-7
Appending Objects  25-7
Inserting Objects  25-8
Deleting Objects 25-8
Moving the Visible Area 25-9
Sorting the Cursor 25-10
Registering a Callback Function 25-10
Cursor Example 25-11

Appendix A Types A-1

- ooAccessMode A-1
- ooAMSUsage A-1
- ooBoolean A-2
- ooDataType A-2
- ooDowngradeMode A-2
- ooError A-3
- ooErrorHandlerPtr A-3
- ooErrorLevel A-3
- ooFileNameFormat A-4
- ooIndexMode A-4
- ooKey A-4
- ooKeyType A-5
- ooLockMode A-5
- ooMode A-5
- ooMsgHandlerPtr A-6
- ooNameHashFuncPtr A-6
- ooStatus A-6
- ooThreadId A-6
- ooVoidFuncPtr A-7
- ooTypeNumber A-7
- ooVersMode A-7
- ooVoidPtr A-7

Parameterized Types and Classes A-8
- ooHandle(className) A-8
- ooItr(className) A-8
- ooRef(className) A-8
- ooShortRef(className) A-9
Appendix B  Macros and Functions  B-1

About Macros
free  B-1
ooAbort  B-1
ooCheckVTablePointer  B-2
ooCleanup  B-2
ooCommit  B-4
ooCommitAndHold  B-5
ooCurrentThread  B-5
ooDelete  B-5
ooDeleteNoProp  B-5
ooGetActiveTrans  B-6
ooGetErrorHandler  B-7
ooGetMemberOffset  B-7
ooGetMemberSize  B-7
ooGetMsgHandler  B-7
ooGetResourceOwners  B-8
ooInit  B-9
ooInitAllThreads  B-9
ooInitThread  B-10
ooNewConts  B-11
ooNewKey  B-12
ooNoLock  B-12
ooRegAtStartFunc  B-13
ooRegErrorHandler  B-13
ooRegMsgHandler  B-13
ooRegOnExitFunc  B-14
ooReplace  B-14
ooResetError  B-15
ooRunStatus  B-15
Appendix C  Global Variables and Constants  C-1

Global Variables
oovLastError  C-1
oovLastErrorLevel  C-1
oovNError  C-1
oovTopDB  C-2
oovTopFD  C-2
oovTopTrans  C-2

Appendix D  Base Classes  D-1

ooObj (d_Persistent_Object)  D-1
ooContObj  D-20
ooDefaultContObj  D-25
ooDBObj  D-27
ooFDObj  D-28
ooGeneObj  D-29
Appendix E  Dictionary Classes
       ooMap  E-1
       ooMapElem  E-7
       ooMapItr  E-8

Appendix F  Index Classes
       Persistent Index Classes  F-1
       ooKeyDesc  F-1
       ooKeyField  F-4
       Non-Persistent Index Classes  F-6
       ooLookupFieldBase  F-6
       ooEqualLookupField  F-6
       ooGreaterThanLookupField  F-7
       ooLessThanLookupField  F-7
       ooGreaterThanEqualLookupField  F-8
       ooLessThanEqualLookupField  F-8
       ooLookupKey  F-9

Appendix G  String Classes
       ooVString, d_String  G-1
       ooString(\texttt{N})  G-8

Appendix H  Transaction Class
       ooTrans, d_Transaction  H-1

Appendix I  VArray Classes
       ooVArray(\texttt{type}), d_Varray<\texttt{type}>  I-1
       ooTVArray(\texttt{type})  I-6
Appendix J  
**Iterator Classes**

- `ooItr(ooObj)`  
- `ooItr(ooContObj)`  
- `ooItr(ooDBObj)`  
- `d_Iterator`  
- `ooQuery`

Appendix K  
**Object Reference and Handle Classes**

- `ooRef` and `ooHandle` Inheritance Hierarchy
- **Relationship Between ooRef and ooHandle Classes**
- **Basic Objects** (`ooObj`)
  - `ooRef(ooObj)`  
  - `ooHandle(ooObj)`
- **Containers** (`ooContObj`)
  - `ooRef(ooContObj)`  
  - `ooHandle(ooContObj)`
- **Databases** (`ooDBObj`)
  - `ooRef(ooDBObj)`  
  - `ooHandle(ooDBObj)`
- **Federated Databases** (`ooFDObj`)
  - `ooRef(ooFDObj)`  
  - `ooHandle(ooFDObj)`
- `d_Database`  
- `d_Ref_Any`

Appendix L  
**Operators on User-Defined Classes**

- `userClass`

Appendix M  
**User-Defined Operators**

- `ooOperatorSet`
Appendix N  User-Defined Functions  N-1
  userRelOperFunction  N-1
  userConversionFunction  N-2

Appendix O  User-Defined Iterator Classes  O-1
  ooItr(userClass)  O-1

Appendix P  User-Defined Object Reference and Handle Classes  P-1
  ooRef(userClass), (d_Ref<userClass>)  P-1
  ooHandle(userClass)  P-1
  ooShortRef(userClass)  P-6

Appendix Q  Association Member Functions for User Classes  Q-1
  userClass  Q-1

Appendix R  Object Conversion Classes  R-1
  ooConvertInObject  R-1
  ooConvertInOutObject  R-5

Appendix S  ODMG Date and Time Classes  S-1
  d_Date  S-1
  ::overlaps  S-10
  ::overlaps  S-11
  ::overlaps  S-11
  d_Time  S-11
  ::overlaps  S-19
  ::overlaps  S-19
  ::overlaps  S-19
Appendix T  Predicate Query Language  T-1

Overview  T-1
Operators and Literals  T-2
Regular Expressions  T-3

Index  i-1
About This Book

This manual describes how to develop C++ applications to create and manipulate persistent objects.

Audience

This document assumes that you are familiar with programming in C++.

Organization

- Chapter 1 provides a brief overview of the process of developing and using an application in Objectivity/C++.
- Other chapters in this book provide more in-depth information on how to develop C++ applications to create and manipulate persistent objects.
- The appendixes contain reference descriptions of the constructs provided in the Objectivity/C++ programming interface.
Introduction to Objectivity/DB Programming

This chapter provides a brief overview of the process of creating and using an application in Objectivity/C++.

Basics

To work with Objectivity/DB, you use C++ and a set of Objectivity/DB-specific member functions to interact with Objectivity/DB persistent objects and perform database operations. You can use standard C++ compilers and tools for all your Objectivity/DB application development work.

Compiling and Linking C++ Applications

Installation and Platform Notes describes the static and shared libraries you will use with Objectivity/DB.

Logical Storage Classes

To exist in an Objectivity/DB database, an object must be an instance of one of the four Objectivity/DB storage classes: basic object, container, database, or federated database. As you become more familiar with Objectivity/DB, you will learn how to use multiple containers, multiple databases, and how to manage several applications concurrently accessing the same federated database.
**Basic Object**

A basic object is the fundamental unit of storage. It may contain any of the following:

- Scalar types (int, float, char)
- Structures and class instances
- Non-persistent C++ objects such as strings
- Multiple aggregates (fixed-size and variable-size arrays)
- Associations to other objects
- Object references to persistent-capable classes

Each basic object is contained within a container.

**Container**

A container is a collection of basic objects. Basic objects within a container are physically clustered together in memory pages and on disk, so access to all basic objects in a single container is very efficient.

Your application can create and access multiple containers. Since locking takes place at the container level, you can optimize concurrent access among multiple applications by placing objects that will be accessed together in the same container and objects that will not be accessed together in different containers.

Each container is contained within a database.

**Database**

A database is a collection of containers. It contains a system-created default container and user-defined containers. The default container holds basic objects that you have not explicitly placed in user-defined containers.

Your application can create and access one or more databases. Since databases are maintained as files on the host file system, they provide a convenient way to administer related containers and basic objects at a particular physical location.

Each database belongs to a single federated database.


**Federated database**

A federated database logically contains user-defined databases and the data model (or schema) that describes all publicly visible class definitions. Most administrative control is at the federated database level, including configuration information (where Objectivity/DB files physically reside) and concurrent access control.

**Persistence**

Unlike C++ objects, objects of the four Objectivity/DB classes are persistent. That is, instances of each class continue to exist after your application terminates. Persistent objects can be shared among applications, with locking managed by Objectivity/DB. Except for a federated database, all types of persistent objects can be created and deleted dynamically by an application program.

**Object Identifiers (OIDs) for Accessing Objects**

To access a persistent object, Objectivity/DB uses an object identifier (OID) to locate the object. OIDs allow Objectivity/DB to locate and manage objects with more flexibility and safety than direct memory access. In addition, Objectivity/DB uses OIDs to provide:

- Object independence for an application through transparent access at runtime to objects located anywhere on the network
- Full interoperability across all platforms
- Access to more objects and databases than a direct memory address permits
- Integrity constraints and runtime type checking that are not possible through direct memory addresses

An OID is 64 bits in length and is composed of four 16-bit fields in the following format:

\[ D-C-P-S \]

where

\( D \) Database identifier
\( C \) Container identifier
\( P \) Logical page number
\( S \) Logical slot number on the page

For additional information about OIDs and memory usage, see the “Federated Database Tasks” chapter in Objectivity/DB Administration.
Object References and Handles

Objectivity/DB provides two special classes that use OIDs to access persistent objects: object references (ooRef) and handles (ooHandle). Both ooRef and ooHandle are type-safe classes that provide many of the operations (such as -> and +) available on C++ pointers, in addition to Objectivity/DB-specific database operations.

The ooRef class is optimized for space. The ooHandle class is optimized for speed and contains additional information that provides more efficient access to multiple fields in the same object. For more information about object references and handles, see Chapter 4, "Accessing Objects".

Conventions for Return Values and Naming

Return Values

The majority of Objectivity/DB member functions and functions return a status code of type ooStatus, which is compatible with type int32.

By convention, when successful, a function returns the constant oocSuccess, which is defined as non-zero. When unsuccessful, a function returns the constant oocError, which has a zero value. If a zero value is returned, one or more Objectivity/DB error messages are printed. For information about error handling, see “Using the Error Handling Facility” on page 19-1.

Naming

For reasons of consistency, and also to provide you with maximum flexibility, Objectivity/DB names follow these conventions:

<table>
<thead>
<tr>
<th>Item</th>
<th>Prefix</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>oo</td>
<td>Except for the primitive types (uint8, int16, uint16, int32, uint32, float32, float64), all type and class names are prefixed with oo.</td>
</tr>
<tr>
<td>Constants</td>
<td>ooc</td>
<td>All system constant names are prefixed with ooc.</td>
</tr>
<tr>
<td>Variables</td>
<td>oov</td>
<td>All system variable names are prefixed with oov.</td>
</tr>
</tbody>
</table>
### Macros and functions

All macro and function names are prefixed with `oo` followed by a name with its first character in upper case, for example, `ooInit`.

### Object member functions

Most of the member functions on persistent objects are prefixed with `oo` just like global macros and functions.

### Object reference and handle member functions

There are no specific prefixes used for member functions on object reference or handle objects. If there is an underscore character `_` in an object reference or handle member function name, then the member function gets or sets the contents of the object reference or handle itself.

<table>
<thead>
<tr>
<th>Item</th>
<th>Prefix</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros and functions</td>
<td>ooX</td>
<td>All macro and function names are prefixed with <code>oo</code> followed by a name with its first character in upper case, for example, <code>ooInit</code></td>
</tr>
<tr>
<td>Object member functions</td>
<td>oo</td>
<td>Most of the member functions on persistent objects are prefixed with <code>oo</code> just like global macros and functions.</td>
</tr>
<tr>
<td>Object reference and handle member functions</td>
<td></td>
<td>There are no specific prefixes used for member functions on object reference or handle objects. If there is an underscore character <code>_</code> in an object reference or handle member function name, then the member function gets or sets the contents of the object reference or handle itself.</td>
</tr>
</tbody>
</table>
Schema Development Overview

A schema, or data model, is the physical representation of the structure of your database. You or your team’s schema developer must create a schema before you can build and run your Objectivity/DB application. Schema development is covered in depth in Using Objectivity/ C++ Data Definition Language. Briefly, to develop a schema, you:

1. Prepare your class definitions using the Data Definition Language (DDL) and identify the classes you want to be persistent.
2. Create a federated database, if necessary.
   Before your application can interact with Objectivity/DB, you must create a federated database. If the federated database has already been created, ask your database administrator for the boot file path. Otherwise, follow the instructions in Objectivity/ DB Administration to use oonewfd to create the federated database.
3. Process your DDL files using the DDL processor.
   You may want to automate this step by creating a makefile. See “Create a Makefile” on page 1-9.

Application Development Overview

Once you have created a schema, you can write application code to create and manipulate objects in the federated database.

The basic activities in application development are shown in Figure 1-1. Specifically, these steps are:

1. Write application code based on the schema.
   ◆ Include the DDL source files, including the Objectivity/DB-generated header file in your application source code.
   ◆ Specify necessary preprocessor directives.
   See “Working with Objectivity/DB Data” on page 1-10.
2. Set up the environment to start database operations.
3. Create or modify a makefile (or project file for Windows or MacOS platforms) to do the following:
   ◆ Run the DDL processor (optional; do this once the schema is complete and stable).
- Compile your C++ application.
- Link the application with the Objectivity/DB libraries and object files.

If you use virtual member functions in your DDL schema, you need to explicitly link your application with a file that references all derived classes that implement the virtual functions. For more information, see “Using Virtual Member Functions” on page 1-8.

4. Build and run the application.
5. View the results in the federated database.
Using Virtual Member Functions

C++ compilers and linkers are not designed to inherently support persistent objects. Because of this, a linker does not link in the implementation code for virtual member functions defined in derived classes, unless the linker encounters a file with a reference to the derived class.

As a result of this behavior, you need to force the linker to bring in the necessary implementation code if both of the following are true for your application:

- Your application in any way accesses an object of a derived class that implements a virtual function
- None of the files linked with your application contain code that references the derived class that implements a virtual function

Under this circumstance, you need to explicitly link your application with a file that references all derived classes that implement virtual member functions. You need to do this to prevent a runtime error when an object of the derived class is accessed.

To achieve this, perform the following steps:

1. Create a dummy file that in some way references the derived class that implements the virtual function. One way of doing this is writing dummy functions with local variables of the derived class.
2. Link this dummy file with your application.

Set Environment Variables

When you work with data stored in Objectivity/DB, you will work with a specific federated database. You can use environment variables to specify which federated database to use as a default when you run the application. You can also specify a default database, if desired.

1. Set the OO_FD_BOOT environment variable to the name of the boot file, which was created when the federated database was created. See “Setting Boot File Environment Variables” on page 2-9 in Objectivity/DB Administration.
2. (Optional) Set the OO_DB_NAME environment variable to the name of the default database you want to work with. This variable is not available on VMS platforms.
Create a Makefile

You can create or modify a makefile to automate:

- Running the DDL processor to read the boot file and process your .ddl file
- Compiling your C++ source code
- Linking your application’s object files, with the Objectivity/DB library

If you already have a makefile that you use to build your C++ applications, you can make a copy and modify it as follows:

1. Add .ddl to other suffix definitions, so that nmake (Windows) or make (UNIX) can handle .ddl files.
   To do so, include a statement to .SUFFIXES such as:
   ```
   otherSuffixes .ddl
   ```
2. Add a rule that runs the DDL processor (ooddlx).
   To do so, include a statement to .ddl.h such as:
   ```
   ooddlx $(CPP_FLAGS) yourSchema.ddl bootFilePath
   ```
3. Modify the command that links your application to include the Objectivity/DB library. See Installation and Platform Notes for library file names.

Build and Run an Application

Perform the following steps to build and run your C++ application:

1. To make sure that the lock server is running, use the oocheckls tool. For more information, see “Checking to See if the Lock Server is Running” on page 7-3 in Objectivity/DB Administration.
   If the lock server is not running, check with your database administrator or start the lock server by following the procedure in “Starting a Lock Server” on page 7-4 in Objectivity/DB Administration.
2. Build the application.
   Objectivity/C++ automatically generates an Objectivity/DB database application from your source C++ application.
3. Type the name of the application and any required input parameters to run it.
View the Federated Database

Objectivity/DB provides a Data Browser and a Type Browser for viewing and modifying your federated database. To browse data or types, see Chapter 4, "Browsing Objects and Types," in Objectivity/DB Administration.

Working with Objectivity/DB Data

All Objectivity/DB database operations occur within transactions. Thus, your application must handle the basic steps outlined here.

1. Initialize Objectivity/DB.
   To do this, you call *ooInit*. For more information see “Initializing Database Services” on page 2-26.

2. Begin a transaction.
   For information on starting a transaction, see “Starting a Transaction” on page 2-29.

3. Open the federated database.
   By default, the federated database that is opened is the one specified in the boot file named by the *OO_FD_BOOT* environment variable. To set or reset this variable, see “Set Environment Variables” on page 1-8.

4. Work with the data.
   You can create, access, delete, move and copy basic objects, containers, and databases. Several chapters in this book describe these activities. You might start by looking at:
   - Chapter 3, "Creating and Deleting Objects"
   - Chapter 4, "Accessing Objects"

   The effect of all operations during a transaction are visible only within the application performing them. When the transaction completes, all of the changes also become visible to other applications. This allows an application to perform a set of coordinated operations that appear as a consistent unit to other applications.

5. Save changes (commit) and end the transaction.
   Generally at the end of a transaction, you save changes made during the execution of your application by committing. If you do not want to save the changes, you can abort the transaction.
For information on ending a transaction, see “Committing a Transaction” on page 2-31 and “Aborting a Transaction” on page 2-36.

Objectivity/DB Features to Enhance Performance

You can improve the flexibility, robustness, and performance of your C++ applications by taking advantage of Objectivity/DB database features in your code.

To implement some of these features, Objectivity/DB provides classes that include higher-level capabilities for manipulating data than those found in the C++ language or in standard C++ function libraries. These classes enable you to easily model and manipulate data, saving you the effort of designing, developing, and testing your own classes. They are specifically designed to improve the robustness and performance of your Objectivity/DB applications.

In many cases, you can take advantage of Objectivity/DB classes by replacing existing code in your application. For example, the variable-size array, string, and dictionary classes perform operations for persistent objects similar to those you may have written yourself. Other features, such as associations, iterators, and indexing offer more powerful features that may influence the basic design of your data model. You can determine which Objectivity/DB classes to include in your C++ application as your requirements change and you discover new ways to use them.

Variable-Size Arrays

Variable-size arrays (VArrays) are similar to standard C++ arrays, except that they can change in size at runtime. Like standard C++ arrays, they are zero-based with indices ranging from 0 to n-1 (where n is the number of elements in the array). A variable-size array takes up only as much storage as required for the number of elements currently allocated to the array, and the array size can either grow or shrink at runtime. The ooVArray classes also provide a degree of reliability by performing bounds checking on array references. Objectivity/DB manages both in-memory and on-disk storage automatically.
The variable length array is a parameterized class for a particular type of element ooVArray(type). You can use ooVArray(type) anywhere in your application. If an ooVArray(type) is embedded within a persistent-capable class, that variable length array is persistent-capable, otherwise it is transient.

Before creating an instance of ooVArray(type) you must declare the class using the C++ declare macro in your .ddl file.

For more information, see Chapter 10, "Using Variable-Size Arrays".

Example

Consider a new class Magazine for a library information system. A Magazine object represents a title, and contains an array in which each element represents an Issue of the magazine.

```cpp
// DDL schema file
...
struct Issue {
    unsigned volume, issue, day, month, year;
};
...
// declares ooVArray(Issue)
declare(ooVArray,Issue);
```

If the ooVArray is used in more than one source file, you may want to put its declaration in a .h file that is included by those source files. You must also generate the actual definition for ooVArray(type) in one of your C++ source files using the C++ implement macro.
On UNIX systems, if a persistent-capable C++ class includes a pointer to an array of storage that is allocated in memory using the C++ new function, you must allocate the array in persistent storage when you convert the application for use with Objectivity/DB. A simple way to replace the pointer and storage allocation code in the original C++ class is by an ooVArray in the persistent-capable class.

Example

To make the class Magazine persistent-capable, we make the following changes.
(Additions are in bold; deletions are crossed out.)

```cpp
Δ class Magazine : public ooObj {
    ...
    - Issue* issues;   // list of magazine issues
    - unsigned issueSize;
    + ooVArray(Issue) issues;
    ...
};
...
- Magazine* magazine;  // a magazine object
+ ooRef(Magazine) magazine;
    Issue newIssue;    // new issue to be added
...```
// expand array by growth factor if full
-   if (magazine->issueSize % 10 == 0) {
+   if (magazine->issues.size() % 10 == 0) {
      // expand memory array
-     Issue* tmpIssues = new(Issue)[magazine->issueSize +
-           10];
-     for (unsigned i = 0; i < magazine->issueSize; i++)
-       tmpIssues[i] = magazine->issues[i];
-     delete magazine->issues;
-     magazine->issues = tmpIssues;
+   magazine->issues.resize(magazine->issues.size() + 10);
+   magazine->issues[size] = newIssue;
   }

---

Strings

The Objectivity/DB string classes are persistent-capable character string classes that provide more functionality than normal character arrays and C++ string libraries. Objectivity/DB defines two string classes: ooVString and ooString(N). Objects of these classes behave like C++ strings and can be passed as arguments using the type char* during a function or member function call. You can use ooVString and ooString(N) anywhere in your program. If either is embedded within a persistent-capable class, that string is persistent-capable, otherwise it is transient. Choose a string class to use based on what you know about the likely lengths of the strings.

For more information, see Chapter 12, "Using the String Classes".

On UNIX systems, if a C++ class includes a pointer to an array of characters that is allocated in memory using the C++ new function or one of the C malloc functions, the array must be allocated in persistent storage when you convert the application for use with Objectivity/DB. The most efficient way to replace the pointer in the original C++ class is by using either an ooVString or an ooString(N).
ooVString

ooVString uses a VArray of characters to implement a C++ string. If the lengths of the strings to be used are not known or if these lengths vary widely, you should use the ooVString class. The ooVString class also provides a degree of reliability by performing bounds checking on references to elements of the ooVString class.

Example

To make a library Loan class persistent-capable, we make these changes:

class Loan {
  ...  
  - char *dueDate;     // due date of loan
  + ooVString dueDate;
  ...  
};

Loan::Loan (char *date) {
  ...  
  - dueDate = malloc(strlen(date) + 1);
  - strcpy(dueDate, date);  // allocate memory
  + dueDate = date;
  ...  
}
**ooString(N)**

*ooString(N)* is a parameterized string class that contains a VArray of characters and a fixed character array. Use this string class to optimize disk space usage and runtime performance if you know that most of the strings are about the same length.

Before creating an instance of *ooString(N)*, you must declare the class using the C++ *declare* macro in one of your DDL schema files.

---

**Example**

```c++
// DDL schema file
declare(ooString,6); // declares ooString(6)
```

---

If *ooString* is used in more than one source file, you may want to put its declaration in a .h file that is included by those source files. You must also generate the actual definition for *ooString(N)* in one of your C++ source files using the C++ *implement* macro.

---

**Example**

```c++
// One of the C++ source files
implement(ooString,6); // defines ooString(6)
```

---

If your application uses a number of different lengths of *ooString*, you may find it convenient to put all declarations into a separate .h file and all implementations into a single C++ source file.

The *ooString* classes also provide a degree of reliability by performing bounds checking on references to elements of the *ooString*.
Associations

Objectivity/DB provides a set of parameterized classes called associations that provide higher level capabilities than simple pointers for modeling and managing relationships between objects. To allow an association between objects, you must define an association link in your data model. In most cases, you can use association links to replace pointers in existing C++ classes that you have made persistent-capable.

You can define associations so that a database operation will propagate from one object to the next along the association. Propagating operations include delete and lock. Propagation is a very useful property when you wish to treat associated objects as a group, known as a composite object. You specify which operations should propagate, and the direction of propagation, when you define the association links in your data model. (Propagation along an association is optional.) For more information, see Chapter 7, "Using Associations".

Example

If a library information system allowed the deletion of Patron objects, the hasLoans association could be defined to propagate to Loan objects, causing all Loan objects for a patron to be deleted when the Patron object is deleted:

```c++
// DDL schema file
class Patron : public ooObj {
    ...
    ooRef(Loan) hasLoans[] -> byPatron : delete(propagate);
    ...
};
```

The propagation in this case is only in one direction; Patron objects should not be deleted automatically when associated Loan objects are deleted.
Iterators

Objectivity/DB provides iterator classes that enable your application to navigate through a collection of objects, filtering for those objects that:

- Are of a particular class and its subclasses
- Belong to a one-to-many or many-to-many association
- Are one level lower in the storage hierarchy
- Have a scope name for an object
- Are named in a given scope
- Meet the conditions in a predicate

An iterator is a non-persistent-capable object, existing only in the context of the process that creates it. When you add a class to a schema, the declaration for the corresponding iterator class is automatically generated for you by the DDL processor. To use an iterator, you only need to declare an instance of the iterator class.

For more information, see Chapter 9, "Using Iterators".

Example

This example iterates for all patrons through a library `memberOfLibrary` one-to-many association:

```cpp
ooRef(Library) library;
ooItr(Patron) nextPatron;
...
library->memberOfLibrary(nextPatron); // initialize iterator
while(nextPatron.next()) { // advance to next patron
    ooRef(Patron) patron = nextPatron; // get patron object
}
```

In this example, the first statement initializes the `Patron` iterator `nextPatron` to step through the `Library association memberOfLibrary`. Calls to the iterator `next` member function in the second statement advance the iterator to the next item in the association. As long as there are more items in the association, the `next` member function returns the non-zero boolean value `oocTrue`. When there are no
more items in the association, the `next` function returns the boolean value `oocFalse`, which has the value 0.

---

**System Names**

A system name serves a purpose similar to a file name, uniquely identifying a persistent object to Objectivity/DB. System names are mandatory for federated databases and databases; they are optional for containers.

For more information, see Chapter 8, "Naming Objects".

System names have the following characteristics:

- Each federated database, database, or container can have only one system name.
- The system name is set when the object is created and cannot change for the lifetime of the object.
- The system name must be unique in the containing object. For example, no two containers in the same database may have the same system name.
- Valid system names follow the same rules for files within the operating system.

Several Objectivity/DB member functions on `ooRef` and `ooHandle` use system names to locate and identify objects. For example:

- The `exist` member function uses the system name of an object to check whether the object exists.
- The `open` member function uses the system name of an object to look it up in order to open it.
- The `name` member function obtains the system name of a federated database, database, or container.

**Dictionaries**

Objectivity/DB includes a dictionary class, `ooMap`, which provides the most efficient way to create names for many objects and use these names to locate the objects. `ooMap` provides a persistent hash table, where each element in the table consists of an object name and object identifier (OID). The key into the table is a C++ or C string that names the object. The `ooMap` dictionary class allows you to create and manage names for objects that reflect the logical organization of your data model.
To use the ooMap dictionary class, you must include the ooMap.h header file in your application.

For information about defining and using ooMap, see the Chapter 13, "Using the Map Dictionary Classes".

Example

If books are frequently looked up by their international standard book number (ISBN), keeping an ooMap of ISBNs speeds up access compared to a linear search through the Library allBooks association:

```c++
#include <ooMap.h>
class Library : public ooObj {
  ...
  ooRef(ooMap) isbnMap; // map of ISBNs
  ...
};
...
ooRef(Library) library;
...
// use ooMap lookup function to locate book with ISBN
ooRef(Book) book = (ooRef(Book) &)
  library->isbnMap.lookup("245678901");
```
Indexes

Indexes enable fast access to basic objects within a specified container by sorting objects of a particular class according to the value in one or more members (fields) of the class. You can use indexes to access basic objects based on ranges of data values or exact matches (for example, “locate all basic objects in a container whose ID ranges from 100 to 200”).

Indexes have the following benefits and characteristics:

- They do not need to be defined in your .ddl files.
- They can be created and deleted at runtime from any application.
- You can have as many indexes as you need.
- You can reuse the same description for many separate indexes.
- Indexes can span containers, databases, or the entire federated database.
- You can use many different indexes on the same objects.

For information on creating and using indexes, see Chapter 14, "Using Indexes".
Transaction Model

This chapter describes the transaction model for Objectivity/DB C++ applications.

All Objectivity/DB database operations occur within one or more transactions. The effect of all operations during a transaction are visible only within the application performing them. When the transaction completes, all of the changes also become visible to other applications. This allows an application to perform a set of coordinated operations that appear as a consistent unit to other applications.

Data manipulation is handled within transactions. The general model for an Objectivity/DB C++ transaction is as follows:

1. Initialize database services.
   To do this, use the ooInit function. This happens just once per process. See “Initializing Database Services” on page 2-26.

2. Start a transaction.
   See “Starting a Transaction” on page 2-29.

3. Open the federated database.
   To do this, declare a federated database object reference or handle using the open member function. Your application can open only one federated database within a process. Within a process, you can have multiple serial transactions that access the same federated database.

   See “Creating a Federated Database” on page 3-2 and “Opening a Federated Database” on page 5-3.

   If you only want to read the objects and do not intend to modify them, and other processes need to read objects in the same database, you should open both the federated database and the database in read mode. See “Concurrency Rules” on page 11-4 for a discussion of concurrent access and the open member function.
4. Open an existing database.
   To do this, declare a database object reference or handle and using the `open`
   member function. In some cases, this step is performed implicitly by
   Objectivity/DB. Alternatively, create a new database using the `new` operator.
   See “Creating Databases” on page 3-3 and “Opening a Database” on page 5-5.

5. Open an existing container.
   To do this, declare a container object reference or handle. In some cases, this
   step is performed implicitly by Objectivity/DB. Alternatively, create a new
   container using the `new` operator. Objectivity/DB automatically locks the
   container object.
   See “Creating Containers” on page 3-7 and “Opening a Container or Basic
   Object” on page 5-6.

6. Obtain an object reference or handle to a basic object.
   If the object does not exist, create it using the `new` operator. Objectivity/DB will
   automatically open and lock this object for your use.
   See “Creating Basic Objects” on page 3-13 and “Opening a Container or Basic
   Object” on page 5-6.

7. Access and manipulate the basic object.
   You work with a basic object through its object reference or handle, which you
   got in the previous step. Within a transaction, you can work with several basic
   objects (just repeat this and the previous steps).
   See “Accessing Objects” on page 4-1.

8. Optionally, close the objects opened during the transaction.
   You can close the basic objects, containers, and databases opened. See “Closing
   Objects” on page 5-10.

9. Complete the transaction.
   You do this either by committing, committing and holding, or aborting the
   transaction. See “Committing a Transaction” on page 2-31 and “Aborting a
   Transaction” on page 2-36.
Warning

On UNIX, if your application creates one or more child processes, you should not perform any Objectivity/DB operations within the child process until you have issued an `exec` call. Violating this rule may lead to database corruption. Specifically, before calling `ooInit` you can fork child processes since Objectivity/DB has no state.

After calling `ooInit`:
- Calling `fork` followed by a call to `exec` is supported. However, remember to call `_exit` in the child process if the `exec` fails (standard UNIX programming practice).
- Calling `fork` and not following it with a call to `exec` is not supported (even if there are no Objectivity/DB operations performed in the child process).

Using Multiple Transactions

You can create and use multiple transactions within the same application process. To do so you must start new transactions and reopen the federated database serially. You can also use concurrent transaction feature to create transactions within non-preemptive threads. For more information, see “Concurrent Transaction Support” on page 2-39.
Initializing Database Services

The first step your application must perform is to initialize database services by calling the `ooInit` function:

```c
ooStatus ooInit(
    const uint32 nFiles  = 12,
    const uint32 nPages  = 200,
    const uint32 nMaxPages  = 500,
    const ooBoolean installSigHandler  = oocTrue)
```

where

- `nFiles` Number of active file descriptors reserved for Objectivity/DB. The default value is 12.
- `nPages` Number of buffer pages initially allocated. The default value is 200.
- `nMaxPages` Maximum number of buffer pages Objectivity/DB can allocate for your application. Setting `nMaxPages` to 0 allows the cache to grow to an unlimited size. The default value is 500.
- `installSigHandler` Flag indicating whether or not you want Objectivity/DB to install the operating system signal handler. The default value is `oocTrue`, which implies that the Objectivity/DB automatically installs the signal handler. If the value is `oocFalse`, no signal handler is installed.

The maximum value of the argument `nMaxPages` is limited by the amount of available swap space.

While the default values should prove sufficient for most applications, choosing other values may improve performance in some cases. See “Find the Best Page Size” on page 21-13 for details on how these parameters affect performance.

You should invoke `ooInit` only one time in each application. Subsequent calls to this function are ignored.
Specifying Memory Size for Large Objects

If your application uses large objects (for example, objects larger than a page), you may want to specify the maximum amount of space that can be used for them. This allows you to bypass the default limit set by Objectivity/DB that may be too small for your application. This limit is set to maximum number of buffer pages \( n_{\text{MaxPages}} \) times the page size. You set \( n_{\text{MaxPages}} \) using the \texttt{ooInit} function described in "Initializing Database Services" on page 2-26. You set the page size when you create your federated database using \texttt{oonewfd} in "Creating a Federated Database" on page 3-1 in Objectivity/DB Administration. To bypass the default memory size used for large objects, call the \texttt{ooSetLargeObjectMemoryLimit} function:

\begin{verbatim}
  void ooSetLargeObjectMemoryLimit ( \\
     uint32 size);
\end{verbatim}

where

\begin{itemize}
  \item \texttt{size}  Size limit (in bytes) for large objects in memory
\end{itemize}

The limit is used to close large objects that are not currently being used so that the amount of memory used for large objects is always less than \( size \). However, the limit will not be enforced if it cannot close enough objects to keep the memory below \( size \). You can call this function anytime after \texttt{ooInit} and as often as needed to increase or decrease the amount of space used by large objects.
Activating Hot Mode

When an application opens an object, Objectivity/DB performs various internal operations to prepare the object. These operations include any necessary heterogeneity conversions, and the allocation of certain in-memory data structures, among others. Similarly, when an object is closed, Objectivity/DB deallocates various internal data structures, requiring some performance overhead.

For most applications, the overhead of performing these open/close operations is insignificant. In fact, if the object to be opened must be read from disk, the cost of open operations are often completely masked by I/O costs of bringing the object into memory. However, if the application is data intensive and accesses many objects that all fit in memory, the cost of the open/close operations can be sometimes significant.

For these kinds of applications, you can use hot mode. Normally, an object is kept open if there are active handles that refer to it, and closed when its last active handle is deactivated.

In hot mode, when an application requests to close an object, the operations to close it are postponed, even though the last open handle to the object is deactivated. The object is closed at the last possible moment, just before removing the object's page from the buffer pool. This can give a performance advantage if the application returns to an object that was previously opened, since the overhead to open the object is avoided.

By default, hot mode is not activated. To activate hot mode, use the \texttt{ooSetHotMode} function:

\begin{verbatim}
  extern void ooSetHotMode(const ooBoolean hotMode = oocTrue);
\end{verbatim}

To disable hot mode once it is activated, call \texttt{ooSetHotMode} with the \texttt{oocFalse} parameter.

Hot mode affects memory usage. Every open object uses 48 bytes of additional in-memory data structures. So, depending on the application, the additional swapping done by the operating system because of the additional memory usage can offset the performance gain from hot mode. For this situation, you should try reducing the size of the buffer pool. Fewer pages in the buffer means that fewer objects can be open at any time, which reduces the amount of memory used for open objects.
Starting a Transaction

As you perform work on a database, changes are recorded by Objectivity/DB, but are not actually written to permanent storage until you explicitly tell it to do so. The unit of work that is applied to the database is called a transaction, and the process of writing (saving) a unit of work to permanent storage is called committing a transaction.

Once a transaction is committed, it cannot be undone. However, at any time until a commit occurs you may choose to abort the transaction, leaving the database in the same logical state as if you had never started the transaction.

To control transactions, your application must first create an instance of the system-defined class ooTrans. Each instance of this class is called a transaction object. Each transaction object uniquely identifies a transaction in progress and is used to store the state of the transaction. For example, to declare a transaction object, your application should have a declaration similar to the following:

- `ooTrans transName; // Declare the transaction object`

where

- `transName` Name of the transaction object to be used by the application.

Objectivity/DB allows one active transaction object per process. Nested or overlapping transactions are not allowed.

After declaring a transaction object and initializing database services with `ooInit`, your Objectivity/DB application must start the transaction before accessing any data. Once started, a transaction will continue until it is either aborted (by the `abort` member function) or committed (by either the `commit` or `commitAndHold` member function). In general, once you have started a transaction there are no requirements on your application to maintain the transaction.

When you create a database or container, it is invisible (except for its name) to other transactions until you commit the transaction that created it (by calling either `commit` or `commitAndHold`).
To start a transaction, invoke the `start` member function on the transaction object:

```cpp
ooStatus ooTrans::start(
    ooMode openMode = oocNoMROW,
    const int32 waitOption = oocTransNoWait,
    ooIndexMode indexMode = oocInsensitive);
```

where

- **openMode** Activate or deactivate MROW (Multiple Readers, One Writer). MROW can improve concurrent access to objects. The default, `oocNoMROW`, specifies a non-MROW transaction. Alternatively, `oocMROW` activates MROW for this transaction. For more information, see “Multiple Readers, One Writer (MROW)” on page 11-8.

- **waitOption** One of four options on how to wait for locks held by other transactions—`oocTransNoWait`, `oocNoWait`, `oocWait`, and `n`. See “Activating Lock Waiting” on page 11-6.

- **indexMode** One of three index modes—`oocInsensitive`, `oocSensitive`, and `oocExplicitUpdate`. See “Setting Index Modes” on page 14-15.

---

**Example**

The following example demonstrates how various member functions of class `ooTrans` affect the state of database transactions during program execution. These member functions are described in subsequent sections of this chapter.

```cpp
void main()
{
    ooTrans transaction;
    transaction.start(); // begin transaction
    ...
    transaction.commit(); // end transaction
    transaction.start(); // begin transaction
    ...
    transaction.abort(); // end transaction
    transaction.start(); // begin transaction
    ...
}
```
Committing a Transaction

At the end of a transaction, you can either commit or abort it. When you commit a transaction, all changes made during the transaction are saved to the database and made visible to other applications. When you abort a transaction, changes made since the last checkpoint or since the beginning of the transaction (if there was no checkpoint) are rolled back and not saved to the database.

During a transaction, you can checkpoint the changes made so far during the current transaction. This makes all changes permanent (commits) and allows other applications to access the changes, but the transaction remains active and all previously set locks are still held by the application. Aborting the transaction at a later time only discards changes made since last checkpoint (commitAndHold).

To commit a transaction, apply one of the member functions discussed in this section.

Commit

- `ooStatus ooTrans::commit();`

As illustrated in Figure 2-2, the `commit` member function does the following:

1. Terminates the current transaction and writes all database changes to disk
2. Modifies object references and handles to contain Object Identifiers (OIDs); see “Using Object References and Handles” on page 4-2
3. Updates all user-defined indexes created or modified by this transaction; see “Creating a Key Description” on page 14-3 and “Creating Key Fields” on page 14-5 for information on creating and modifying indexes
4. Closes the federated database if it is open (this also closes any persistent objects opened during the transaction)
5. Releases any locks acquired in the course of the transaction
6. Invalidates pointers to objects

```
transaction.commitAndHold(); // end transaction
// begin new transaction
...
transaction.commit(); // end transaction
```
To continue with another transaction, your application must:

1. Start a new transaction using the `start` member function
2. Open the same federated database again (you can only open one federated database per process)
3. Locate and open any persistent objects you wish to access

Example

In this example, a federated database with system name `ECAD` and a database with system name `ALU` are both opened for read access. After some processing, the transaction is committed.

```c++
#include "exampleSchema.h" // User DDL schema header file
...
ooHandle(ooFDObj) fdHandle;
ooHandle(ooDBObj) dbHandle;
ooTrans transaction;
...
ooInit(); // Initialize services
transaction.start(); // Start transaction
fdHandle.open("ECAD", oocRead); // Open federated database
dbHandle.open(fdHandle, "ALU", oocRead); // Open database
...
// Perform whatever processing is necessary
...
returnCode = transaction.commit(); // commit transaction
...
```
Committing a Transaction

Key To Symbols

- = Step performed by Objectivity/DB
- = Step performed by Application

Figure 2-2 Committing a Transaction
commitAndHold

Alternatively, your application can commit and hold the transaction using the following member function:

```c++
void ooTrans::commitAndHold(
    ooDowngradeMode mode = oocNoDowngrade);
```

where

- `mode` - Lock downgrade mode. If set to `oocNoDowngrade` (the default), `commitAndHold` preserves all locks held by the transaction. If set to `oocDowngradeAll`, update locks are downgraded to read locks (MROW read if the transaction is an MROW transaction, and normal read locks otherwise). Downgrading the lock mode is an advanced feature. You should only use this feature if you have a thorough understanding of Objectivity/DB transaction and locking semantics.

As illustrated in Figure 2-3, this member function:

1. Terminates the current transaction and writes all database changes to disk
2. Preserves validity of all system buffers, including pointers to objects
3. Preserves (holds) all locks acquired during the transaction, or downgrades update locks to read locks if `mode` is set to `oocDowngradeAll`
4. Updates all applicable indexes defined within the federated database
5. Does not close the federated database or any persistent objects that were opened during the transaction
6. Implicitly starts a new transaction

Using `commitAndHold`, an application can continue as if no interruption of the transaction has occurred. This member function is equivalent to the checkpoint facility found in traditional DBMSs.

To continue the processing after invoking `commitAndHold`, the application could simply locate the already opened persistent objects you wish to access.
Commit And Hold Transaction

Write Changes To Disk

All System Buffers Including Object Pointers Remain Valid

Update Indexes

Hold Locks Acquired During Transaction Or Downgrade Locks From Update To Read

Objects Opened During Transaction Remain Open

Start New Transaction

Key To Symbols

= Step performed by Objectivity/DB

= Step performed by Application

Figure 2-3 Committing and Holding a Transaction
Aborting a Transaction

To abort a transaction, invoke the following member function on the transaction object:

```cpp
ooStatus ooTrans::abort(
    ooHandleMode mode = oocHandleToNull);
```

where

- `mode` Determines what happens to active object references and handles of the transaction after it is aborted. The default is `oocHandleToNull`, which converts all active object references and handles to null. Alternatively, `mode` can be set to `oocHandleToOID` to convert all active object references and handles to Object Identifiers (OIDs).

As shown in Figure 2-4, this member function terminates the current transaction and restores the database to its original logical state. All locks acquired during the aborted transaction are also released. This member function does the following:

1. Terminates the current transaction and does not write changes to disk
2. Modifies object references and handles to contain either null or Object Identifiers (OIDs); see “Using Object References and Handles” on page 4-2
3. Closes the federated database if it is open (this also closes any persistent objects that were opened during the transaction)
4. Invalidates system buffers, including pointers to objects
5. Releases any locks acquired in the course of the transaction

When an application terminates (either by a call to the `exit` function or otherwise), any active transaction is aborted automatically.

If a transaction is terminated either by stopping the process within a debugger or by killing the process with a `SIGKILL` signal, the locks acquired during the transaction are not released. To clear these locks, follow the procedures described in the “Automatic and Manual Recovery” chapter in Objectivity/DB Administration.
Warning

If the parameter oocHandleToOID is specified in the argument of abort, some of the OIDs might be invalid after the aborted transaction. Your application must avoid reusing these invalid OIDs again.

Figure 2-4  Aborting a Transaction
Example

In this example, a federated database with system name ECAD and a database with system name ALU are both opened for read access. After some processing, the transaction is aborted.

```
#include "exampleSchema.h" // User DDL schema header file

ooHandle(ooFDObj) fdHandle;
ooHandle(ooDBObj) dbHandle;
ooTrans transaction;

ooInit(); // Initialize services
transaction.start(); // Start transaction
fdHandle.open("ECAD", oocRead); // Open federated database
dbHandle.open(fdHandle, "ALU", oocRead); // Open database

// Perform whatever processing is necessary

returnCode = transaction.abort(); // abort transaction
```

Checking for an Active Transaction

You can use the following member function to check whether or not a transaction is active:

- `ooBoolean ooTrans::isActive();`

The member function returns the constant `oocTrue` if a transaction is active or the constant `oocFalse` if there is not an active transaction.
Improving Concurrency

Locks held during a transaction are only released after the transaction is either committed or aborted. This allows other applications to gain write access to data your application has written, and greatly improves the ability for multiple applications to operate concurrently.

With commitAndHold, although the transaction does not release its locks, you can improve concurrency by downgrading locks from update to read as described in “commitAndHold” on page 2-34.

You can also improve concurrency using the concurrent transactions feature described in “Concurrent Transaction Support” on page 2-39.

Concurrent Transaction Support

The Objectivity/DB Objectivity/C++ product supports concurrent transactions. This feature allows you to run multiple transactions at the same time in the same process. The transactions must operate on the same federated database, and each transaction is started in its own non-preemptive thread.

In this documentation, the term thread differs from threads used in preemptive thread packages from other vendors. Objectivity/DB threads cannot be preemptively switched. Your application must explicitly switch them. They contain a full set of the global state associated with an Objectivity/DB session (including the buffer cache). You can explicitly create new threads, terminate threads, and switch between threads.

For example, consider a graphics editor package in which each displayed window represents a thread. The application explicitly switches between windows and threads through mouse events (such as clicking in a new window or entering a new window). Each window (or set of windows displaying the same item) can have its own transaction, which can be committed or aborted independently of the other transactions.
Warning

The Objectivity/DB concurrent transaction feature is designed only for situations where the application program explicitly transfers control among threads. This feature does not allow you to use preemptive thread packages with Objectivity/DB transactions in more than one thread.

In addition to kernel-based thread packages from other vendors, other thread packages that transfer control through the libc library, signal handlers, or timers are also preemptive and cannot be used with the Objectivity/DB concurrent transaction feature.

In summary, consider the following when using concurrent transactions:

- All threads in a process must operate on the same federated database
- Each thread will have its own copy of all global state information (including its own copy of the buffer cache, error handlers, and so on)
- Threads are non-preemptive. You must explicitly switch between sessions/threads outside of signal handlers. You cannot use this feature with a thread package that uses preemptive scheduling.

Type and Constant for Concurrent Transactions

To manage multiple threads, Objectivity/DB concurrent transaction functions use the opaque type ooThreadId type to identify a thread to Objectivity/DB. These functions use constant oocMainThreadId to specify the main thread that is initialized when ooInitAllThreads is called.

Error Handling and Concurrent Transactions

Objectivity/DB error handlers are specific to the thread in which they are created. Therefore, to use error handlers with concurrent transactions, you should register a new error handler for each thread your application uses. For more information about using error handlers, see the “Error Handling” chapter.
Concurrent Transaction Support

Initializing Objectivity/DB for Concurrent Transactions

To initialize Objectivity/DB for concurrent transactions, call the function ooInitAllThreads instead of ooInit. As with ooInit, you should call this function as the first Objectivity/DB call and only one time per process.

ooInitAllThreads initializes Objectivity/DB for the application process to use multiple threads, including allocating main thread and global data structures. The syntax for ooInitAllThreads is as follows:

```
extern "C" ooStatus ooInitAllThreads(
  IN uint32 nFiles = 12,
  IN uint32 nPages = 200,
  IN uint32 nMaxPages = 500,
  IN ooBoolean installSigHandler = oocTrue);
```

where

- **nFiles** Number of active file descriptors reserved for Objectivity/DB. Default value is 12.
- **nPages** Initial number of buffer pages initially allocated for the main thread’s cache. Default value is 200.
- **nMaxPages** Maximum number of buffer pages that Objectivity/DB may allocate for the main thread’s cache. Default value is 500.
- **installSigHandler** Indicates whether you want the system to install the operating system signal handler or not. If the value is oocFalse, then no signal handler will be installed. Default value is oocTrue.

If successful, ooInitAllThreads returns oocSuccess. Otherwise, it signals an error and returns oocError.
Initializing a Single Thread

To initialize a single thread, use the `ooInitThread` function. You must call this function for a thread before it makes any other Objectivity/DB calls. This function is used to initialize (allocate) the Objectivity/DB data structures for this thread. `ooInitThread` does not automatically switch thread context. The syntax of `ooInitThread` is as follows:

```c
extern "C" ooStatus ooInitThread(
    IN uint32 nPages,
    IN uint32 nMaxPages,
    OUT ooThreadId* threadIDPtr);
```

where

- `nPages` Initial number of buffer pages allocated for this thread's cache
- `nMaxPages` Maximum number of buffer pages Objectivity/DB may allocate for this thread's cache
- `threadIDPtr` Pointer to identifier for this thread

If successful, `ooInitThread` assigns a thread identifier to `*threadIDPtr` and returns `oocSuccess`. Otherwise, it signals an error and returns `oocError`.

Switching Context to a Thread

Call `ooSwitchThread` to switch context to another thread. `ooSwitchThread` switches to the appropriate thread data structures for use in subsequent Objectivity/DB calls by the thread. Since `ooInitThread` does not automatically switch thread context, you must call `ooSwitchThread` after `ooInitThread` to switch to the thread being initialized.

After calling `ooInitThread` and `ooSwitchThread`, a thread can start a transaction just as if the process were single threaded. The syntax of `ooSwitchThread` is as follows:

```c
extern "C" ooStatus ooSwitchThread(
    IN ooThreadId targetThreadId);
```

- `targetThreadId` Identifier for target thread

If successful, this function returns `oocSuccess`. Otherwise, it signals an error and returns `oocError`. 
Terminating a Thread

To terminate a thread, call `ooTermThread`. You must call this function after a thread has completed all other Objectivity/DB calls. This function terminates (frees) data structures for the thread being terminated.

If you call `ooTermThread` within a thread using its own thread identifier, the context is switched back to the main thread. Otherwise, `ooTermThread` leaves the context unchanged. The syntax of `ooTermThread` is as follows:

```c
extern "C" ooStatus ooTermThread(
    IO ooThreadId* threadIDPtr);
```

`threadIDPtr` Pointer to identifier for the thread being terminated

If successful, this function sets the given thread identifier to zero (an invalid thread identifier) and returns `oocSuccess`. Otherwise, it signals an error and returns `oocError`.

Determining Which Thread is Active

To determine which thread is active, call the `ooCurrentThread` function. This function returns the thread identifier to which the context is currently set (the active thread). The syntax of `ooCurrentThread` is as follows:

```c
extern "C" ooThreadId ooCurrentThread();
```
Example of Using Concurrent Transactions

In this example, a federated database with system name `myFD` and a database with system name `myDB` are both opened for read access. Two threads are created, `thread1` and `thread2`. Transactions are started within these threads and objects are processed. `thread2` commits its transaction and `thread1` aborts its transaction.

---

Example

```c
#include "exampleSchema.h" // User DDL schema header file
...
ooHandle(ooFDObj) fdHandle1, fdHandle2;
ooHandle(ooDBObj) dbHandle1, dbHandle2;
ooTrans transaction1, transaction2;
ooThreadId thread1, thread2;
...
// In main thread
ooInitAllThreads(); // Initialize Objectivity/DB
ooInitThread(50, 100, &thread1); // Initialize thread1
ooInitThread(20, 50, &thread2); // Initialize thread2
//
ooSwtichThread(thread1); // Switch to thread1
transaction1.start(); // Start transaction1 for thread1
fdHandle1.open("myFD", oocRead); // Open federated database
dbHandle1.open(fdHandle1, "myDB", oocRead); // Open database

// read and write to objects as necessary for thread1
...
ooSwitchThread(thread2); // Explicitly switch to thread2
transaction2.start(); // Start transaction2 for thread2
fdHandle2.open("myFD", oocRead); // Open federated database
dbHandle2.open(fdHandle2, "myDB", oocRead); // Open database
```
// read and write to objects as necessary for thread2
...
transaction2.commit(); // Commit transaction2 for thread2
ooSwitchThread(thread1); // Explicitly switch back to thread1
// read and write to objects as necessary for thread1
...
transaction1.abort(); // Abort transaction1 for thread1
ooTermThread(&thread2); // Terminate thread2 within thread1
// Check current thread
if (ooCurrentThread() == thread1)
    printf("Thread context is in thread1.\n");
ooTermThread(&thread1); // Terminate thread1
// Check current thread
if (ooCurrentThread() == oocMainThreadId)
    printf("Thread context is in main thread.\n");

Creating and Deleting Objects

This chapter describes how to create, replace, and delete persistent and transient objects using Objectivity/DB.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form \texttt{const ooHandle( className )}& are interchangeable with \texttt{const ooRef( className )}&.

Creating Persistent Objects

Objectivity/DB stores and provides facilities for managing persistent objects, which continue to exist and retain their object properties outside the scope of program execution. These objects are standard C++ objects with powerful extensions. For example, persistent objects can:

- Be very large (exceeding the size of main memory)
- Contain multiple variable-size arrays
- Be logically grouped together to form composite objects
- Be versioned

Persistent objects reside in the federated database, and are brought into virtual memory when requested by an application. Once they are in virtual memory the application can manipulate the persistent object.
Objectivity/DB supports four kinds of persistent objects:

- Federated databases
- Databases
- Containers
- Basic objects

For a description of these objects, see “Logical Storage Classes” on page 1-1.

Federated databases and databases have direct relationships to physical files. Understanding these relationships may help you when debugging and testing your application. For more information on objects as physical files, see the “Objectivity/DB Basics” chapter of Objectivity/DB Administration.

**Declaring Persistent Classes**

For an object to exist in an Objectivity/DB federated database, the object must be an instance of a persistent class you declare in a schema file using the Data Definition Language (DDL). The object’s persistent class must be a derived class of one of several system-defined persistent classes. These classes specify the various properties that a user-defined persistent class can inherit.

To add persistent classes to the database schema, you must process your DDL schema file with the DDL processor. This process also generates a C++ source file and a C++ header file with the declarations that allow you to create and manipulate objects of the defined persistent class. See Using Objectivity/C++ Data Definition Language for more information on schemas and the DDL.

**Creating a Federated Database**

To create a federated database, you or your database administrator must use the onewfd tool and procedures described in the “Federated Database Tasks” chapter of Objectivity/DB Administration.
Creating Databases

Your application can create and access one or more databases. Since databases are maintained as files on the host file system, they provide a convenient way to administer related containers and basic objects at a particular physical location.

A database is a persistent object of class ooDBObj. You can create databases using either the new operator or the ooReplace macro.

You specify where the database will be created either explicitly as part of the initializer, or implicitly using the host and directory of the federated database clustering directive. By default, operator new creates databases on the same host and in the same directory as the current federated database.

Newly-created databases are automatically opened in update mode so they can be immediately modified. See “Read and Update Access Modes” on page 5-2. Creating a database also creates a default container of class ooDefaultContObj.

Using new

To create a database, the new operator is generally used as follows:

```c
ooHandle(ooDBObj) dbH = new ooDBObj(
    char* dbSysName,
    const uint32 defContInitPages = 0,
    const uint32 defContGrowth = 0,
    const char* hostName = 0
    const char* pathName = 0
    uint32 weight = weight);
```

where

- `dbH` Handle of new database
- `dbSysName` System name of the database to create. `dbSysName` follows the same naming rules as files of your operating system.
- `defContInitPages` Initial number of pages to allocate for the default container. The value of this argument is limited to a maximum of 65535. If equal to zero, a default value of 4 is used.
- `defContGrowth` Specifies that, when necessary, the default container for the database should grow by `defContGrowth` percent of its current size. If equal to zero, a default value of 10% is used.
hostName  Name of the host system on which the database file is to be created. If you specify a host name, you must also specify a path name in pathName.

pathName  Full path name of the directory where the database file is to be created. If you specify a pathname, you must also specify a host name in hostName. If no path information is provided (the value of pathName is 0), the database file is created in the same directory as the federated database file. Optionally, pathName can include the file name for the database file. (This file naming option is not available on VMS.)

weight  Weight of the first database image if Objectivity/DB Data Replication Option is being used. The default weight is 1.

Operator new creates a database and returns a memory pointer to it, which you can assign to a handle. It returns a null pointer if a database named dbName already exists or if an error occurs while creating the database. You can verify the creation of the database by comparing the value of the handle to 0 (null) using the C++ operator ==.

You must assign the result of the new operator to a handle. You can then optionally assign the handle to an object reference. It is illegal to assign the result of the new operator to a pointer or object reference.

The ooDBObj::operator new is formally defined as:

```cpp
void* operator new(  
    size_t,  
    const ooHandle(ooFDObj)& fdH = oovTopFD);  
```

where

size_t  Automatically initialized by the compiler with the size of the class type in bytes and should not be passed by the user

oovTopFD  Default federated database (the federated database opened in this process)
The following example creates a database named CMOS. For a more complete example, see “Example Using new and delete” on page 3-22.

```
// Create new database CMOS in federated database
// referenced by fdH
ooHandle(ooDBObj) dbH = new(fdH) ooDBObj("CMOS");
```

This example creates a new database for all information at the science library, which is located in a separate facility from other libraries in the collection, and creates a `Library` object identifying the library in the default container of the database.

```cpp
class Library : public ooObj { // declaration from .ddl file
    ...};
void main() {
    ooHandle(ooDBObj) scienceLibDB;
    ooHandle(Library) scienceLib;
    ... scienceLibDB = new ooDBObj
        ("newtonLibrary");
    scienceLib = new (scienceLibDB) Library
        ("Isaac Newton Science Library", "100 Campus Way", "7-8056");
}
```

The initializer for the database specifies a system name for the database. See “Using System Names” on page 8-1 for more about system names.
Using ooReplace

You can use the ooReplace macro to replace an existing database. The syntax for ooReplace is as follows:

```c
ooHandle(ooDBObj) ooReplace (ooDBObj,
   (char* dbSysName,
   const uint32 defContInitPages,
   const uint32 defContGrowth,
   const char* hostName,
   const char* pathName),
   const ooHandle(ooFDObj)& fdH);
```

Parameters for this macro are similar to those for new. The ooReplace macro creates a database in the federated database and returns a handle to the new database. This macro deletes any existing database named `dbSysName` in the federated database before creating the new one.

⚠️ Warning

Once an existing database has been removed by ooReplace, it cannot be recovered. That is, aborting the transaction after ooReplace has removed an existing database does not restore the original database. The old database is gone even after the abort operation.

Therefore, unless absolutely necessary, avoid using ooReplace on ooDBObj. Use the new operator to create a database instead.
Creating Containers

A container is a persistent object that derives from class ooContObj. To create a container you use the new operator. You can also create a batch of containers using the ooNewConts macro (see “Creating a Batch of Containers” on page 3-11).

Your application can create and access multiple containers. Since locking takes place at the container level, placing objects that will be accessed together in the same container and objects that will not be accessed together in different containers helps optimize concurrent access among multiple applications.

Table 3-1 summarizes the use of new for transient and persistent containers.

<table>
<thead>
<tr>
<th>Operator/Macro</th>
<th>Container Class Type</th>
<th>Container Created</th>
</tr>
</thead>
<tbody>
<tr>
<td>new(0) className(args)(^1)</td>
<td>Persistent</td>
<td>Transient</td>
</tr>
<tr>
<td>new className(args)</td>
<td>Persistent</td>
<td>Persistent</td>
</tr>
<tr>
<td>new(objH) className(args)</td>
<td>Persistent</td>
<td>Persistent</td>
</tr>
</tbody>
</table>

1. Applying the new operator on a persistent container class with a clustering directive of 0 or a pointer to a transient object will create a transient container.

Hashed Containers

Objectivity/DB containers can be either hashed or non-hashed. Only hashed containers support scope naming and keyed objects. For example, to use a container (or any basic objects it contains) as the scope name space for another object, you must use a hashed container. For an object named in the scope of a database (or federated database), the hash pages of its default container are used. Similarly, keyed objects must be created within a hashed container.

To create a non-hashed container, specify a hash value of zero. Neither this container nor any basic objects it contains can be used as a name scope when naming an object. You also cannot create keyed objects within this non-hashed container. For more information about scope names and keyed objects, see “Using Scope Names” on page 8-6 and the “Using Keyed Objects” chapter.

To create a hashed container, you must specify a hash value of one or greater. The hash value is used as a hash clustering factor when you create keyed objects and use scope names. A default container is always created with a hash value of 1.
There are performance tradeoffs between using hashed and non-hashed containers. For additional information, see the “Monitoring and Tuning Performance” chapter.

Using new

To create a container, the new operator is generally used as follows:

- \( \text{oohandle}(\text{className}) \ contH = \text{new}( \)
  \( \text{const oohandle(ooObj)} & \ objH) \)
  \( \text{className}(\text{initializers}); \)

- \( \text{oohandle}(\text{className}) \ contH = \text{new}( \)
  \( \text{const char*} \ contSysName, \)
  \( \text{const uint32} \ hash, \)
  \( \text{const uint32} \ initPages, \)
  \( \text{const uint32} \ percentGrowth, \)
  \( \text{const oohandle(ooObj)} & \ objH) \)
  \( \text{className}(\text{initializers}); \)

where

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>Name of class of container to create</td>
</tr>
<tr>
<td>initializers</td>
<td>List of arguments passed to the constructor for class className</td>
</tr>
<tr>
<td>contH</td>
<td>Handle to the new container</td>
</tr>
<tr>
<td>contSysName</td>
<td>System name of the container. If no name, specify a null pointer or empty string</td>
</tr>
<tr>
<td>hash</td>
<td>Type of container as either hashed or non-hashed. See “Hashed Containers” on page 3-7. If hash = 0, then a non-hashed container is created. If hash &gt; 0, then a hashed container is created. If you do not care about the clustering factor, set the value of hash to 1.</td>
</tr>
<tr>
<td>initPages</td>
<td>Initial number of pages to allocate for the container. The value of this argument is limited to a maximum of 65535. If equal to zero, the default value is used (4 for a hashed container, 2 for non-hashed). For information on determining initPages, see “Find the Best Page Size” on page 21-13.</td>
</tr>
<tr>
<td>percentGrowth</td>
<td>Specifies this container should grow by a percentage of the current size. If equal to zero, the default value of 10% is used (that is, an integer value of 10).</td>
</tr>
</tbody>
</table>
Handle (or object reference) to use as a clustering directive specifying where the new container will be created. If the clustering directive is a database handle, the container will be created in the specified database. If the clustering directive is a container handle or a basic object handle, the new container will be created within the database that contains the container or basic object referenced by the handle. The default value is the handle of the most-recently opened or created database.

Operator `new` creates a container and returns a memory pointer to it, which you can assign to a handle. It returns a null pointer if a container named `contSysName` already exists or if an error occurs while creating the container. You can verify the creation of the container by comparing the value of the handle to 0 (null) using the C++ operator `==`.

You must assign the result of the `new` operator to a handle. You can then optionally assign the handle to an object reference. It is illegal to assign the result of the `new` operator to a pointer or object reference.

The `ooContObj::new` operator is formally defined as:

```cpp
void* operator new(
    size_t,
    const ooHandle(ooObj)& objH = oovTopDB);
```

```cpp
void* operator new(
    size_t,
    const char* contSysName,
    const uint32 hash,
    const uint32 initPages,
    const uint32 percentGrowth,
    const ooHandle(ooObj)& objH = oovTopDB)
    className (initializers);
```

The parameters are the same as those described in the usage synopsis above with the addition of:

- `size_t` Automatically initialized by the compiler with the size of the class type in bytes and should not be passed by the user
- `oovTopDB` Handle of the most recently opened or created database
Since `operator new` is defined in each persistent container class automatically (inserted in the persistent class by the DDL processor), a member function name collision may occur between Objectivity/DB’s defined `operator new` and your application’s `operator new`. If this happens, redesign your application to change your `new` operator to avoid the collision.

---

**Example**

The following example provides the schema class definition, `Cell`, for a container and the application code to create an instance of this class. For a more complete example, see “Example Using new and delete” on page 3-22.

```cpp
// DDL schema code
class Cell : public ooContObj {
    public:
        Cell(char *cellName) { strcpy(name, cellName); }
        char name[32];
};
```

```cpp
// Application code
ooHandle(ooDBObj) dbH;
...
// Create a new container under the database referenced by dbH
ooHandle(Cell) cellH;
cellH = new("adder", 1, 10, 40, dbH) Cell("adder");
```

---

**Clustering Containers**

You can also provide a clustering directive for `new` that ensures the new container is created near another container within the same database. If the clustering directive is a basic object, the new container is created near the one containing the basic object. Using a database as a clustering directive ensures only that the container is created within that database. By default, `operator new` creates containers in the most recently opened or created database.
Creating a Batch of Containers

You can use the macro `ooNewConts` to create a batch of containers. The main benefit of using this macro is better performance over calling `new` a number of times. The arguments required for `ooNewConts` are similar to those for `new` and with some additions:

- `void ooNewConts (className, uint32 number, const ooHandle(ooObj) objH, uint32 hash, initPages, uint32 percentGrow, ooBoolean open, ooHandle(className) *pHandle)`

  where

  - `number` Number of containers to be created, limited to 32766
  - `open` Boolean flag indicating whether to open or close the batch containers after creation. If `oocTrue`, all containers are left open after creation. If `oocFalse`, all containers are closed. It is better to close all the containers if none will be used immediately.
  - `pHandle` Pointer to an array of container handles. This array is used to store all the handles of the newly created containers. The number of handles in this array should be at least equal to the number of containers to be created.

The remaining parameters are used the same way as in the `new` operator. The `hash`, `initPages`, and `percentGrow` parameters are applied to each new container. You can examine the status of running `ooNewConts` by checking if the first handle in the handle array is 0 or not. If it is not 0, then the operation was successful.

If there are any constructors in the container class, you must also add a constructor that takes no argument. This is similar to rules for creating an array of objects.

With `ooNewConts` you cannot:

- Give a system name to each container
- Assign `ooNewConts` to any handle
- Use `ooNewConts` as an expression
Example

class Computer : public ooContObj {
    public:
        Computer();
        Computer(char* name);
        char name[32];
        uint32 id;
    }

    ooHandle(ooDBObj) dbH;
    ooHandle(Computer) compH[10];
    ...

    // Create 10 Computer objects, and then close these 10
    // objects after its creation.
    ooNewConts(Computer, 10, dbH, 1, 0, 0, oocFalse, compH);

    if (compH[0]==0) {
        fprintf(stderr, "Fail in ooNewConts()\n");
    }

    // Create 10 Computer objects, and then close these 10
    // objects after its creation.
    ooNewConts(Computer, 10, dbH, 1, 0, 0, oocFalse, compH);

    if (compH[0]==0) {
        fprintf(stderr, "Fail in ooNewConts()\n");
    }
Creating Basic Objects

An object must be an instance of a derived class of the Objectivity/DB class `ooObj` to exist persistently in an Objectivity/DB federated database. After you have defined a persistent-capable class, you can create new persistent basic objects that are instances of that class using the operator `new`.

You can also provide a clustering directive to operator `new` that ensures the new basic object is created near another basic object and within the same container. Using a container as a clustering directive ensures only that the basic object is created in that container. Using a database as a clustering directive ensures only that the basic object is created within the default container for that database. By default, operator `new` creates basic objects in the default container of the current database.

Your application can use the `new` operator to create basic objects. The following table summarizes the use of `new` for transient and persistent basic objects.

<table>
<thead>
<tr>
<th>Operator/Macro</th>
<th>Basic Object Class Type</th>
<th>Basic Object Created</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new(0) className(args)</code>(^1)</td>
<td>Persistent</td>
<td>Transient</td>
</tr>
<tr>
<td><code>new className(args)</code></td>
<td>Persistent</td>
<td>Persistent</td>
</tr>
<tr>
<td><code>new(objH) className(args)</code></td>
<td>Persistent</td>
<td>Persistent</td>
</tr>
</tbody>
</table>

\(^1\) Applying the `new` operator on a persistent basic object class with a clustering directive of 0 or a pointer to a transient object will create a transient basic object.

Using `new`

To create a persistent basic object, the `new` operator is generally used as follows:

```cpp
ooHandle(className) newObjH = new(
    const ooHandle(ooObj)& objH)
    className(initializers);
```

where

- `className` Name of the class of basic object to create
- `newObjH` Handle of the new basic object
Handle (or object reference) to use as a clustering directive. If you specify a database handle, \texttt{new} creates the basic object in the default container of the specified database. If you specify a container handle, \texttt{new} creates the basic object in the specified container. If you specify a basic object handle, \texttt{new} creates the basic object near the specified basic object, if space is available, and if not, somewhere in the same container. If you don't give a clustering directive, \texttt{new} creates the basic object in the default container of the default database, where the default database is the most recently created or opened database.

List of arguments passed to the constructor for class \texttt{className}

Operator \texttt{new} creates a basic object and returns a memory pointer to it, which you can assign to a handle. It returns a null pointer if an error occurs while creating the basic object. You can verify the creation of the basic object by comparing the value of the handle to 0 (null) using the C++ operator \texttt{==}.

You must assign the result of the \texttt{new} operator to a handle. You can then optionally assign the handle to an object reference. It is illegal to assign the result of the \texttt{new} operator to a pointer or object reference.

The \texttt{ooObj::new} operator is formally defined as:

\begin{verbatim}
void* ooObj::operator new(
    size_t,
    const ooHandle(ooObj)& objH = oovTopDB);
\end{verbatim}

where

- \texttt{size_t} Automatically initialized by the compiler with the size of the class type in bytes and should not be passed by the user
- \texttt{oovTopDB} Handle of the most-recently opened or created database

Since operator \texttt{new} is defined in each persistent class automatically (inserted in the persistent class by the DDL processor), a member function name collision may occur between Objectivity/DB's defined operator \texttt{new} and your application's operator \texttt{new}. If this happens, redesign your application to change your \texttt{new} operator to avoid the collision.
**Examples**

The following example provides the schema class definition `Net` for a basic object and the application code to create an instance of this class. For a more complete example, see “Example Using new and delete” on page 3-22.

```c++
// DDL schema class definition for basic object Net
class Net : public ooObj {
public:
    char name[32];
    Net(char *netName) { strncpy(name, netName, 32); } 
};

// Application code to create basic object
// net A within a Cell container adder
ooHandle(Net) net1 = new(cellH) Net("A");
```

This example creates a new persistent `Book` basic object.

```c++
class Book : public ooObj {  // declaration from .ddl file

};
void main() {
    ooHandle(Book) book;

    book = new Book("Moby Dick", "Fiction", "Melville", "1234567890");
}
```

This example creates a new persistent `Book` basic object near an existing `Book` basic object within the same container.

```c++
class Book : public ooObj {  // declaration from .ddl file

};
```
Creating Transient Objects

Since persistent classes have all the characteristics of C++ classes, they can also be instantiated as automatic variables on the stack, or allocated on the heap using the C++ new operator. Objects created this way are not persistent and some class member functions may not apply to the transient objects. Because transient objects do not have object references or handles assigned to them, they cannot be:

- Used in associations, including unidirectional associations
- Used in iterations
- Placed in maps
- Given system or scope names

Table 3-1 summarizes the use of new for transient objects.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Class Type</th>
<th>Object Created</th>
</tr>
</thead>
<tbody>
<tr>
<td>new(0) className(args)</td>
<td>Persistent</td>
<td>Transient</td>
</tr>
</tbody>
</table>

1. Applying the new operator on a persistent object class with a clustering directive of 0 or a pointer to a transient object will create a transient object of that persistent class.
You should only use transient objects as isolated scratch pads for temporary data values. If you need a temporary area to store transient data and associations, you should create a scratch container and put your transient data in it. Doing so allows you to use all of the facilities provided by Objectivity/DB (for example, crash recovery, handling temporary data that exceeds the size of virtual memory, and so on) to manipulate your transient data. Before committing a transaction, you can delete your scratch container to avoid consuming additional disk space.

To create a transient object, use the class-specific operator new with a clustering directive of 0.

```cpp
class Net : public ooObj {
public:
    char name[32];
    Net(char *netName) { strcpy(name, netName); }
};
// Create a transient object for class Net
Net *pNet;
pNet = new(0) Net("sum");
```

Deleting a Federated Database

To delete a federated database, you or your database administrator must use the oodeletefd tool and procedures described in “Deleting a Federated Database” on page 3-4 of Objectivity/DB Administration.
Deleting Databases, Containers, and Basic Objects

Your application can use the `delete` operator, the `ooDelete` macro, or the `ooDeleteNoProp` macro to delete an object. The following table summarizes their use for deleting transient and persistent objects.

<table>
<thead>
<tr>
<th>Operator/Macro</th>
<th>Objects Class</th>
<th>Deleted Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>::delete</code></td>
<td>Persistent</td>
<td>Transient</td>
</tr>
<tr>
<td><code>delete</code></td>
<td>Persistent</td>
<td>Persistent</td>
</tr>
<tr>
<td><code>ooDelete(args)</code></td>
<td>Persistent</td>
<td>Persistent</td>
</tr>
<tr>
<td><code>ooDeleteNoProp(args)</code></td>
<td>Persistent</td>
<td>Persistent</td>
</tr>
</tbody>
</table>

where

```
args
```

One or more arguments other than 0

For example, `::delete` must be used to delete a transient object, while `delete` deletes a persistent object.

Deleting Persistent Objects

To delete a persistent object, you can use either `delete` or `ooDelete`. The `ooDelete` macro performs exactly the same function as the `delete` operator. The destructors for all the persistent classes are virtual, which require you to use an explicit object pointer as the argument for operator `delete`. If you use operator `delete` to delete a persistent object, you will need to extract the pointer on a handle first, and then call operator `delete`.

You must use the `ooDelete` macro to delete a database.

The `delete` operator, `ooDelete`, and `ooDeleteNoProp` will delete a persistent object even if there are multiple open handles to the object. That is, an object will be deleted no matter how many times it has been opened.
When you delete a container, it deletes the basic objects it contains. In doing so, it maintains referential integrity by deleting any associations that are set between the deleted objects and other objects. However, for performance reasons, deleting a container does not call the destructors of the basic objects in it. To delete basic objects of a container by calling their destructors, iterate through the basic objects in the container and explicitly delete them.

Similarly, deleting a database does not call the destructors of its containers and basic objects. You must explicitly delete these objects before deleting the database.

**Using the delete operator**

The `delete` operator is class-specific and is defined in the system class `ooObj`. The syntax for the `delete` operator is:

```c
void operator delete(
    void *);
```

For an example of using `delete` to delete persistent objects, see “Example Using `new` and `delete`” on page 3-22.

**Using ooDelete and ooDeleteNoProp**

Alternatively, your application can use the `ooDelete` or `ooDeleteNoProp` macro to delete a persistent object. The syntax for these macros are:

```c
ooStatus ooDelete(
    const ooHandle(class_name) &objH);
```

```c
ooStatus ooDeleteNoProp(
    const ooHandle(class_name) &objH);
```

where

- `className` Class of object to be deleted
- `objH` Handle of object to be deleted
When either macro is applied to an object handle, the following sequence of events occurs:

1. If `className` has a destructor, the destructor is called on the object being deleted before the storage is deallocated. If called on a container or database, these macros will not call the destructors of the objects they contain.
2. All associations attached to the object are deleted. All bidirectional associations will be updated automatically to maintain federated database integrity.

⚠️ Warning

It is your responsibility to clean up dangling references that may be left by unidirectional links pointing to the deleted object.

3. If the persistent object being deleted is a container or database, all persistent objects contained within the persistent object being deleted are also removed.
4. The storage allocated for the object is reclaimed by Objectivity/DB.

If `ooDelete` or operator `delete` is used, the deletion will be propagated along associations that have delete propagation. If `ooDeleteNoProp` is used, deletion will not be propagated. For more information about propagation, see “Composite Objects” on page 2-29 of Using Objectivity/ C++ Data Definition Language.

⚠️ Warning

Once a database has been removed by `delete` or `ooDelete`, it cannot be recovered. That is, aborting a transaction in which `delete` or `ooDelete` has removed an existing database does not restore the removed database.
Example

This example deletes the object pointed to by `rectH`.

```c
ooHandle(Rectangle) rectH;
...
// rectH is set to a valid handle value
...
ooDelete(rectH);
```

Deleting Transient Objects

The following table summarizes how to delete a transient instance of a persistent class using Objectivity/DB. Unlike standard C++, you must use the global operator `::delete`.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Object Class</th>
<th>Deleted Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>::delete</code></td>
<td>Persistent</td>
<td>Transient</td>
</tr>
</tbody>
</table>

Example

```c
// Delete a transient object for class Net
// Net is assumed to be a persistent class
Net *pNet;
pNet = new(0) Net("carry-in");
::delete pNet;
```
Example Using `new` and `delete`

The following example illustrates the use of `new` and `delete` operators for creating and deleting objects in an Objectivity/DB federated database.

```cpp
// DDL schema code for persistent classes Cell, Net, and Port
class Cell : public ooContObj {
public:
    char name[32];
    Cell(char* cellName) { strncpy(name, cellName, 32); }
};

class Net : public ooObj {
public:
    char name[32];
    Net(char* netName) { strncpy(name, netName, 32); }
};

class Port : public ooObj {
public:
    char name[32];
    Port(char* termName) {
        strcpy(name, termName);
        // Create a net attached to the port, and use the
        // pointer to this Port object as a clustering
        // directive.
        ooHandle(Net) netH;
        netH = new(this) Net(termName);
    }
};
```

3-22 Using Objectivity/C++
// Application code to create and delete persistent objects using handles.

// Create new database CMOS in federated database referenced by fdH
ooHandle(ooDBObj) dbH = new(fdH) ooDBObj("CMOS");

// Create a net in the default container of the database referenced by dbH
ooHandle(Net) vdd = new(dbH) Net("vdd");

// Create a cell container in the database referenced by dbH
ooHandle(Cell) cellH = new(dbH) Cell("adder");
if (cellH == 0) {
    printf("Error !\n");
}

// Create a net A within the cell container adder
ooHandle(Net) net1 = new(cellH) Net("A");

// Create a net B close to the net A
ooHandle(Net) net2 = new(net1) Net("B");

// Create a port cin close to the net B
ooHandle(Port) portH = new(net2) Port("cin");

// Nullify a handle
portH = 0;

// Delete the net vdd
Net *pVdd = vdd;
delete pVdd;
// Delete the cell container
delete (Cell*)cellH;
Accessing Objects

This chapter describes how Objectivity/DB identifies basic objects, and the mechanisms you can use in your C++ applications to access persistent objects.

Object Identifiers

To identify a persistent basic object, Objectivity/DB uses an object identifier (OID). An OID identifies the database and container to which a basic object belongs, as well as other storage information. OID values do not change during the lifetime of an object.

OIDs allow Objectivity/DB to locate and manage basic objects with more flexibility and safety than direct memory access. In addition, Objectivity/DB uses OIDs to provide:

- object independence for an application through transparent access at runtime to objects located anywhere in the network
- full interoperability across all platforms
- access to more objects and databases than a direct memory address permits
- integrity constraints and runtime type checking that are not possible through direct memory addresses

Warning

Every persistent basic object has a unique OID within a given federated database. However, when an object is removed (using the delete operator or ooDelete macro) its OID may be reused as new objects are created.
Access Mechanisms

Figure 4-1 illustrates the two primary mechanisms to access persistent objects: object references and handles.

![Object Access](image)

**Using Object References and Handles**

Objectivity/DB provides two special classes that use OIDs to access persistent objects: object references and handles.

- An object reference (ooRef) provides a type-safe way to directly access a persistent object. You can also use an object reference as a data member within a persistent object to store a reference to another persistent object.

- A handle (ooHandle) also provides a type-safe way to directly access persistent objects. Unlike object references, however, you cannot use a handle as a data member within a persistent object to store a reference to another persistent object. ooHandle classes maintain additional state information that allows them to more efficiently access multiple fields of the same object than ooRef.

A corresponding handle class is automatically created in the schema header file by the DDL processor for each persistent class in your application's schema.
The ooRef and ooHandle classes provide the same functionality and can be used virtually interchangeably. Both ooRef and ooHandle are type-safe classes that provide many of the same operations (such as `->`) available on C++ pointers, in addition to Objectivity/DB-specific database operations. Several member functions only available for ooHandle classes rely on the additional state information maintained by ooHandle classes; these member functions include `operator ooObj*`, and `operator*`.

Where syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle.

### Using the ooRef and ooShortRef Classes

Objectivity/DB defines parameterized object reference classes ooRef(className) and ooShortRef(className), where className is a persistent class. Object references of class ooRef(className) access objects using long typed OIDs, while objects of class ooShortRef(className) use short typed OIDs.

Long OIDs identify a basic object in the scope of the entire federated database. Short OIDs take up approximately half as much space as long OIDs, and may be desirable for applications where physical storage is a concern. Short OIDs identify a basic object within the scope of a single container.

### Using the Member Function `operator ->`

When `operator ->` is used on an object reference, it causes a transient object to be generated. C++ mechanisms cause `operator ->` to be propagated from an object reference to a transient object and to a virtual memory pointer to an object of the correct class.
If you use `operator ->` several times on an object reference to access object member functions or data members, it is more efficient to assign the object reference to an explicitly-declared handle, as shown in the following example:

```
Example

ooRef(A) aR;             // declare an object reference
...
ooHandle(A) aH = aR;     // initialize a handle
```

On cfront 2.1-based systems (for example, Sun C++ 2.1), the `operator ->` for object references cannot be used in the position of a loop variable since it returns a temporary variable. Programs that use this construct will not compile. To work around the problem, use the object reference outside the loop. The following example illustrates this problem and shows a work around solution.

```
Example

ooRef(A) aR;
ooHandle(A) aH;
int j, n;

// the next two lines will not compile
for (j=0; j<(aR->partNumber); j++);
for (j=0; j<n=(aR->partNumber); j++);

// the next two lines are fine
n = aR->partNumber;
for (j=0; j<n; j++);
```
Using isValid

To check the validity of an object reference or handle, you can use the following member function:

- ooBoolean ooRefHandle(ooObj)::isValid() const;

The performance of this member function is most efficient if the corresponding basic object is already open. If both the basic object and the container containing the basic object are not open, the container remains locked in read mode after the checking.

Since OIDs may be reused, isValid indicates only that the object reference or handle points to a basic object; it does not indicate that the class of that basic object corresponds to the class of the object reference or handle.

If Objectivity/DB cannot open the federated database, database, container, or basic object for checking because it is already locked by another process, then isValid will return a value of oocFalse.

Accessing Persistent Objects

To access a persistent object, follow these steps:

1. Create an object reference or handle of the appropriate class.
   For example, to access a persistent object of class className you must first create an object of class ooRef(className) or ooHandle(className).

2. Set the object reference or handle to reference the desired persistent object.

3. Invoke the appropriate object reference or handle member functions and operators to access and manipulate the persistent object.

The details of these steps are covered in the remainder of this section.

Example

Figure 4-2 demonstrates the handle declaration process. On the left in the figure is a schema file with declarations for classes Layer, Cell, and Shape. On the right is the program source, which includes the DDL output header file and declares a handle type for each of the classes. These handle classes are ooHandle(Layer), ooHandle(Cell), and ooHandle(Shape), respectively.
Figure 4-2  Declaring Object Handles
Setting an Object Reference or Handle

Before you can do anything with a persistent object, you must have an object reference or handle that references it. Object references and handles can be set to reference a persistent object by one of the following operations:

◆ creating a new persistent object; see “Creating Persistent Objects” on page 3-1
◆ using an iterator; see “Using Iterators” on page 9-1
◆ accessing an association link; see “Using Associations” on page 7-1
◆ assignment using another object reference or handle; see “Class Compatibility and Casting”
◆ using one of several member functions defined on object reference or handle classes; see “Member Functions and Operators” on page 4-8

All object references and handles can become invalid after you finish a transaction. For example, following an abort, all handles are set to null and any reference to an old handle will either result in an exception or reference to an incorrect object. To avoid these potential side effects, it is good design practice to revalidate each object reference or handle before using them in subsequent transactions.

Class Compatibility and Casting

Both object reference and handle classes form class hierarchies that corresponds exactly to that of persistent object classes. This means that you can assign to an object reference or handle another object reference or handle of the same class, or its derived classes.

The ability to assign an instance of an object reference or handle derived class to an instance of an object reference or handle base class can be useful when you wish to treat persistent objects in a generic fashion. However, if you do so, you must be careful when invoking object reference or handle member functions, because the member function invoked will be that defined for the base class. To invoke the member function defined on the actual derived class, you should explicitly cast the object reference or handle back to the derived class and then invoke the member function. If you do not want to do type casting, you can make use of virtual member functions to achieve a similar effect.
Example

Assume that we have a handle of class `ooHandle(Rectangle)`, which is set to reference a Rectangle. The following code segment shows how to:

1. assign the Rectangle handle to a basic object handle.
2. cast the basic object handle back to a Rectangle handle.
3. invoke the `name` member function, which is defined on `Rectangle` but not on `ooObj`.

```c++
\begin{verbatim}
fooHandle(ooObj) objH; // basic object handle.
fooHandle(Rectangle) rectH; // Rectangle handle.
...
// rectH is set to reference a Rectangle Object
...
// Assign instance of fooHandle(Rectangle) to instance of fooHandle(ooObj).
objH = rectH;
...
// Now cast back to Rectangle to invoke the name member function.
((fooHandle(Rectangle)&)objH)->name("nmos.1");
\end{verbatim}
```

Member Functions and Operators

The object reference and handle class declarations generated by the DDL processor include a number of member functions and operators that allow you to manipulate the persistent object's database attributes, and also to access the persistent object itself. The member functions depend on the system-defined persistent class from which a user-defined class is derived.

Detailed syntax and parameter information for all of the object reference and handle member functions is available in the “Object Reference and Handle Classes” appendix. The remainder of this chapter explains how to use object reference and handle member functions.
Invoking Member Functions

Objectivity/DB uses two operators to invoke member functions from an object reference or handle: the arrow (\(\rightarrow\)) operator and the dot (\(\cdot\)) operator. There are two fundamental rules regarding the use of these operators (see Figure 4-3):

◆ If the member function is defined on the persistent object that the object reference or handle references, then use the arrow operator.
◆ If the member function is defined on the object reference or handle itself, then use the dot operator.

You may also find the following guidelines helpful when choosing which operator to use. Assuming you have an object reference or handle object that references a persistent object, you should use the arrow operator if any of the following conditions are true:

◆ member functions or data members you want to access are defined in the schema file as part of a persistent object class.
◆ association member functions generated by the DDL processor are defined as member functions on a persistent object class.
◆ you are using an Objectivity/DB-defined member function with a name prefixed by `oo` (for example, `ooGetTypeN`), since these member functions are all defined on a persistent object class.

You should use the dot operator if the following is true:

You are invoking any Objectivity/DB-defined member function that is not prefixed by `oo` (for example, `open`), since these member functions are defined on an object reference or handle class.

![Figure 4-3  Dot and Arrow Operators](image-url)
Example

Assume that we have a handle `objH` that has been initialized to reference a basic object. This example invokes the `open` member function, which is defined on the handle, and also the `ooGetTypeN` member function, which is defined on the basic object itself.

```cpp
ooHandle(ooObj) objH;
...
// objH is initialized to reference a basic object
...
// Invoke the open handle member function on objH
objH.open(oocRead);
// Invoke the ooGetTypeN member function on the object
// referenced by objH.
objH->ooGetTypeN();
```

Accessing Data and Member Functions

Once you have set an object reference or handle to reference a persistent object, you can use that object reference or handle just as you would use a C++ pointer to the object. The arrow (`->`) operator defined on `ooRefHandle(ooObj)` and `ooRefHandle(ooContObj)`, the star (`*`) operator defined on `ooHandle(ooObj)` and `ooHandle(ooContObj)`, and their derived classes allow you to do this.

Thus, given an object reference or handle to an object, there are two ways to access a particular member (field or member function):

- `refHandle -> member`
- `(*handle).member`

where

- `refhandle` Object reference or handle
- `handle` Handle.
- `member` Desired member (field or member function)
When an object is dereferenced using either operator, it is implicitly opened in read mode if it is not already open.

---

**Example**

This example invokes the member function `setTitle` and sets the value of attribute `number` on the persistent object referenced by handle `chapterH`. Note how the handle is used like a standard C++ pointer to dereference the persistent object.

```cpp
toHandle(Chapter) chapterH;
...
// Assume currentDB is set to a valid handle value
...
// Create an instance of class Chapter
chapterH = new chapter();
...
// Use the -> operator
chapterH->setTitle("Introduction");
chapterH->number = 1;
...
// Do the same using the * operator
(*chapterH).setTitle("Introduction");
(*chapterH).number = 1;
```
Obtaining an Object Reference or Handle Using ooThis

When writing an object member function, you may wish to invoke a system-defined member function on the object's object reference or handle. Rather than passing the object reference or handle to your member function as a parameter, you may use either form of the ooThis member function, which is defined for all persistent classes.

The first form of ooThis returns an object reference or handle pointing to the object:

- ooHandle(className) & className::ooThis(
  ooHandle(className) & objH) const;
- ooRef(className) & className::ooThis(
  ooRef(className) & objId) const;

where

    className    is the name of the object's class.
    objH          is a handle for storing the value returned by the member function.
    objId         is an object reference for storing the value returned by the member function.

The second form returns a transient handle to the object:

- ooHandle(className) className::ooThis() const;

Of these two, the first form is recommended. The second form is less efficient and is provided solely for programming convenience.
Using Object References and Handles

Example

This example shows a code segment from the member function `name` defined in class `Rectangle`. This code segment uses both forms of the `ooThis` member function to invoke the `nameObj` member function on the object's handle. See "Setting Scope Names" on page 8-7 for information on the `nameObj` member function.

```cpp
class Rectangle : public ooObj {
  void name(char* objName)
  {
    ...
    ooHandle(Rectangle) rectH;
    // Use the reference version of ooThis()
    ooThis(rectH).nameObj(scopeH, objName);
    // Use the temporary, less efficient version of ooThis
    ooThis().nameObj(scopeH, objName);
    ...
  }
}
```

Handle Registration

When an object reference or handle is declared in a program, Objectivity/DB registers the object reference or handle and places it on the top of the stack. When the transaction commits, Objectivity/DB automatically:

- invalidates the cache memory pointer in each active object reference or handle
- sets each handle to the Object Identifier (OID) of the object it references so it can be used again by another transaction.

If the transaction is aborted, Objectivity/DB sets the OID in each handle to null. If at any time the handle goes out of scope, or if the destructor is called on the handle, it is removed from the stack.
Managing the Handle Stack

Objectivity/DB uses a stack to keep track of all active handles (that is, still accessible and not out of program scope). If your application performs error handling that requires cleanup of the stack, you can use two functions—\texttt{ooSetMark} and \texttt{ooReleaseMark} to mark and unmark the stack. Using these functions, you can clean up active handles in your application.

\begin{itemize}
\item \texttt{ooMark ooSetMark();} \\
This function sets a new mark in the stack, and returns the mark.
\item \texttt{ooStatus ooReleaseMark(ooMark mark);} \\
This function releases the mark specified in the argument. All active handles above this mark will be cleaned up and popped off the stack.
\end{itemize}

Using Pointers with Object References or Handles

Objectivity/DB automatically performs memory management functions when you use object references or handles exclusively in an application. If you are using object references or handles, and still choose to use pointers as well, serious problems may result if you are not careful. To avoid these problems, follow the recommendations in this section.

Assigning a Pointer to an Object Reference or Handle

You must not use a pointer to a persistent object where a member function or operator expects an object reference or handle. In particular, you should never assign a pointer to an object reference or handle, either explicitly or implicitly.

You must assign the result of the \texttt{new} operator to a handle. You can then optionally assign the handle to an object reference. It is illegal to assign the result of the \texttt{new} operator to a pointer or object reference.
Warning

C++ imposes a very tight restriction on all `new` operators: they must return a pointer to void (that is, `void *`). So, to allow the assignment of the result of the `new` operator to an object reference or handle, the C++ programming interface to Objectivity/DB defines an assignment operator from a pointer to each handle class. You cannot safely use pointer-to-handle assignment for anything except its designed purpose: assignment of the result of the `new` operator to a handle.

Example

```cpp
// safe assignment to handle from result of operator new
ooHandle(B) bH;  // declare handle bH
bH = new(aH) B;  // assign result of new to handle bH
// dangerous assignment to handle from a standard pointer
ooHandle(A) aH;  // declare handle aH
A *pA;          // declare pointer pA
...             // initialize pointer pA
aH = pA;        // do not assign pointer to handle!
```

Initializing an Object Reference or Handle with a Pointer

Since C++ initialization is a special case of assignment, you should not initialize an object reference or handle with a pointer. The only exception to this rule is for the pointer returned by the `new` operator.
Using Pointers with Object References or Handles

Example

// safe initialization of handle using operator new
ooHandle(A) aH = new(cH) A;  // declare and initialize handle
// using operator new pointer

// dangerous initialization of handle using standard pointer
A *pA;  // declare pointer pA
...  // initialize pointer pA
ooHandle(A) aH = pA;  // do not assign pointer to handle!

Using ooThis Instead of Pointer this

The C++ reserved keyword this is used within a member function body to represent a pointer to the object that invoked the member function. Because this is a pointer, you must not assign it to an object reference or handle. Instead, use the member function ooThis, which returns an object reference or handle, when assigning to or initializing an object reference or handle.

Example

// dangerous assignment of pointer this to a handle
B::link()
{
    ooHandle(B) bH;
bH = this;  // do not assign pointer this to handle!
...}

4-16 Using Objectivity/C++
Using Pointers with Object References or Handles

// obtain a handle to an existing object using ooThis
B::link()
{
    ooHandle(B) bH;
    ooThis(bH); // assign handle bH using member function
    ooThis();   // ooThis as an alternative, bH =
    // works safely, but is not as efficient
...
}

Passing a Pointer as an Object Reference or Handle

You must not pass a pointer to a member function or function that expects an object reference or handle. Doing so causes a transient handle to be constructed so that the pointer can be converted to the needed object reference or handle. Such a conversion is just as dangerous as any other pointer-to-handle assignment. For example, the following code may cause serious problems.

Example

// dangerous pass of pointer as argument
extern void test(ooHandle(A));
A *pA;
...
// pA is initialized
test(pA);    // dangerous pass of pointer to function.
            // The function is expecting a handle as an
            // not a pointer.
Object Reference Example

The following example shows object reference assignment and the use of the object reference dereference operator `operator->`.

Example

```
// DDL schema file - a.ddl
#include <string.h>

class A : public ooObj {
public:
    A(const char *name, const uint32 id) : id_number(_id) {
        strcpy(name, _name);
    }
    char name[32];
    uint32 id_number;
};

class B : public ooObj {
public:
    float64 salary;
    ooRef(A) aR;
};

// Application source file - test.C
#include "a.h"
ooHandle(ooDBObj) dbH;
...
// Assume dbH is a valid handle to a database.
// Create an object of A.
ooHandle(A) aH;
aH = new(dbH) A(\"Peter Pan\", 154776898);
```
// Create an object of class B.
ooHandle(B) bH;
bH = new(aH) B();
bH->salary = 10000.0;
bH->aR = aH;

// Use operator-> to access the object of class A attached
// to the object of class B
printf("name = %s, id number = %d, salary = %g\n",
    bH->aR->name, bH->aR->id_number, bH->salary);

// Save the object reference of the object of class B
ooRef(B) bR = bH;

// Assign an object reference to a handle
ooHandle(B) b2H = bR;

Accessing Containers

You can access containers in many ways, including:

- By using the system name assigned to the container (see Chapter 8, "Naming Objects")
- By looking up the container in a dictionary (ooMap) (see Chapter 13, "Using the Map Dictionary Classes")
- By following an association to the container from another object (see Chapter 7, "Using Associations")
- By iterating through the database where the container is located (see Chapter 9, "Using Iterators")
- Through the containedIn member function on ooRef or ooHandle for any object within the container
Example

This example locates the container of an existing Book object using the containedIn member function on the book persistent handle:

class Book : public ooObj {  // declaration from .ddl file
    ...
};

void main() {
    ooHandle(Book) book;
    ...
    ooHandle(ooContObj) bookCont = book.containedIn();
}

Accessing Databases

Whenever you create or explicitly open a database, the Objectivity/DB system updates the global oovTopDB database variable to point to the newly opened or created database. The new database pointed to by oovTopDB becomes the default database used by many Objectivity/DB functions.

If your application uses multiple databases, you can continue to access the current default database, by assigning a database handle to the current default database before creating or opening another database.
This example saves a handle to the default database previously opened before creating a new Science Library database. Notice that once the new database is created it can be used as the default database for creating a new Library basic object.

class Library : public ooObj {  // declaration from .ddl file
  ...
};

void main() {
  ooHandle(ooDBObj) defaultDB;
  ooHandle(ooDBObj) scienceLibDB;
  ooHandle(Library) scienceLib
  ...
  defaultDB = oovTopDB;       // maintain access to default DB
  scienceLibDB = new ooDBObj
    ("newtonLibrary");
  scienceLib = new Library("Isaac Newton Science Library",
         "100 Campus Way", "7-8056");
}
Opening and Closing Objects

This chapter describes techniques to open and close objects in your application. The process of opening and closing objects is closely related to concurrency and locking mechanisms. To fully understand the implications of opening and closing objects, you should also see the “Locking and Concurrency” chapter.

Where syntax and parameter information is identical for the object reference and handle classes, `ooRefHandle` is used to represent either `ooRef` or `ooHandle`. Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`.

Opening Objects

Once you have set a handle to reference a persistent object, your application can open the persistent object either implicitly or explicitly using an appropriate member function or operator. For example, calling the open member function in your application will explicitly open an object. However, by traversing a group of objects using an iterator member function such as `scan`, Objectivity/DB will implicitly open objects if necessary during the iteration. See the “Using Iterators” chapter for more information on iterators.
The act of opening a persistent object causes Objectivity/DB to fetch the object from secondary storage into virtual memory, if it is not already present. When you open an object you also inform Objectivity/DB whether or not you intend to modify the object.

Read and Update Access Modes

When a persistent object is opened (either implicitly by Objectivity/DB or explicitly by your application) it is assigned one of two access modes. The access mode of an object tells Objectivity/DB that the object is either available for reading (read mode) or updating (update mode). By default, Objectivity/DB implicitly opens objects in update mode when they are created, and implicitly opens objects in read mode when they are accessed (that is, when an object's handle is dereferenced).

Your application can explicitly open an object in read mode to indicate that you intend to view the object and not modify it. If you make changes to the object, the changes will be written to the federated database if the object resides on the same memory page as another object opened for updating. Because of this danger, you should not modify objects that are open in read mode.

Your application can also explicitly open an existing object in update mode to indicate that you intend to modify the object. All changes you make to this object will be saved to the federated database. The update mode for an object applies only to the object itself, and not to objects it contains. For example, if you open a container in update mode, and make changes to the container itself, your changes will be written to the federated database when you commit the transaction. However, if you were to modify a basic object within this container, the changes to the basic object would not be saved unless you also opened the basic object in update mode.

Objectivity/DB automatically promotes the access mode for a container from read mode to update mode when a basic object that it contains is opened in update mode. However, you must also explicitly open the federated database for update to guarantee that changes you make to these basic objects are saved in the federated database. For example, to modify a basic object, you must open for update both the federated database and the basic object itself. If you do not open the federated database for update, you will get an error message when you attempt to open the basic object for update.

When updating an object, you should change the object's access mode to update mode before making any changes to the object. Otherwise, changes you make may be lost.
When a process opens a container in update mode, the container is locked to prevent other processes from reading or updating the container and its basic objects. This locking mechanism may reduce the concurrency of your application. To improve concurrency, you can postpone locking a container by opening it in read mode to allow the container and its basic objects to be read by other processes. You can then automatically promote the container to update mode by opening one of its basic objects for update when necessary. You can also improve concurrency by using the Multiple Readers, One Writer (MROW) mechanism. For more information on concurrency, locking, and MROW, see the “Locking and Concurrency” chapter.

You should open an object in update mode if you are using a member function that modifies a persistent object. Because the C++ function prototype syntax does not convey this information, you should adopt a convention of documenting this for every member function you define on a persistent class.

### Opening a Federated Database

To explicitly open a federated database, use the following member function:

```cpp
ooStatus ooRefHandle(ooFDObj)::open(  
    const char *bootFilePath,  
    const ooMode openMode = oocRead,  
    ooBoolean recover = oocFalse);
```

- **bootFilePath**
  Path to the boot file of the federated database. The default value is zero. You can specify this path with or without a host name. If you specify it as a host path, use the format `host::path`.

- **openMode**
  Specifies the open mode for the federated database. Valid values for `openMode` are the constants `oocRead` and `oocUpdate`. The default open mode is `oocRead`.

- **recover**
  Specifies whether automatic recovery on the local host should be performed on application startup.

You can open only one federated database within a process, although there is no limitation on the number of times it may be opened and closed within the process.

To enable automatic recovery from C++ application failures, call `ooRefHandle(ooFDObj)::open` with the `recover` parameter set to `oocTrue` when opening a federated database for the first time in an application. For performance reasons, you may want to do this only one time in your application.
For more information about automatic recovery, see the “Automatic and Manual Recovery” chapter in Objectivity/DB Administration.

When you open a federated database with the `open` member function, you can include a network path or relative path to the boot file of the federated database. This allows you to place the boot file in a directory other than the one running your application.

For example, the following are valid boot file paths for the boot file ECAD:

- Single name (for example, `ECAD`)
- Local full path name (for example, `/mnt/john/design/ECAD`)
- Network path name (for example, `/net/object/mnt/john/design/ECAD`)
- Relative path name (for example, `../design/ECAD`)
- Zero or empty string (for example, `(char*)0` or `""`). This applies only to the `open` member function. In this case, Objectivity/DB obtains the boot file path from the environment variable `OO_FD_BOOT`.

---

**Example**

The following example uses the `open` member function to open a federated database in update mode. The federated database system name is `ECAD`, and the boot file path is `/net/object/mnt/john/design/ECAD`.

```c
ooHandle(ooFDObj) fdH; // federated database handle.
...
fdH.open("/net/object/mnt/john/design/ECAD", oocUpdate);
```

The following example uses the `open` member function to open a federated database in update mode. The federated database system name is `MCAD`, and the boot file path is `design/ECAD on host sun22`:

```c
ooHandle(ooFDObj) fdH; // federated database handle.
...
fdH.open("sun22::design/MCAD", oocUpdate);
```
Opening Objects

If you intend to modify any object contained in the federated database, you must open the federated database in update mode to guarantee your changes will be saved in the federated database.

You can also check to see if a federated database exists before attempting to open it by using the `exist` member function. See “Checking for Object Existence” on page 8-3 for a description of the `exist` member function.

**Opening a Database**

To explicitly open a database, use the following member function:

```cpp
ooStatus ooRefHandle(ooDBObj)::open(
    const ooHandle(ooFDObj)& fdH,
    const char* dbSysName,
    const ooMode openMode = oocRead);
```

where

- **fdH** Handle of the containing federated database.
- **dbSysName** System name of the database.
- **openMode** Mode in which to open the database (oocRead or oocUpdate). The default open mode for the `open` member function is oocRead.

You can also check to see if a database exists before attempting to open it by using the `exist` member function. See “Checking for Object Existence” on page 8-3 for a description of the `exist` member function.

You can open a database in update mode using the following member function:

```cpp
ooStatus ooRefHandle(ooDBObj)::update() const;
```

This member function is equivalent to `open(oocUpdate)`.
Opening Objects

Example

The following code segment opens the federated database named ECAD in update mode, and then opens the database named UPROC in read mode, and the database named EPROM in update mode. Note that ECAD must be opened for update because changes are anticipated in the database EPROM.

```cpp
ooHandle(ooFDObj) fdH;    // Federated database handle
ooHandle(ooDBObj) dbH1;   // Database handle
ooHandle(ooDBObj) dbH2;   // Database handle

// open the federated database named ECAD
fdH.open("ECAD", oocUpdate);

// open the database named UPROC for read access
dbH1.open(fdH, "UPROC", oocRead);

// open the database named EPROM for update access
dbH2.open(fdH, "EPROM", oocUpdate);
```

Opening a Container or Basic Object

If you know the system name of a container, you may open it using the following member function:

```cpp
ooStatus ooRefHandle(ooContObj)::open(
    const ooHandle(ooDBObj)& dbH,
    const char* contName,
    const ooMode openMode = oocRead);
```

where

- `dbH` Handle of the database in which the container resides.
- `contName` System name of the container.
- `openMode` Mode in which to open the container (oocRead or oocUpdate). The default open mode for the open member function is oocRead.
You may also open a container or basic object using one of the following member functions:

1. \[
\text{ooStatus \ ooRefHandle(className)}::\text{open}(
   \text{const \ ooMode \ openMode = oocRead}) \text{ const;}
\]
   where
   
   className is ooObj, ooContObj, or a derived class

2. \[
\text{ooStatus ooRefHandle(className)}::\text{update}() \text{ const;}
\]
   This member function is equivalent to \text{open(oocUpdate)}.

You do not need to explicitly open a container or basic object using the \text{open} member function to obtain read access.

Objectivity/DB will automatically open a container or a basic object for read access when it is accessed (that is, when its object handle is dereferenced).

If you have initially opened a container or basic object for read access, you may use either the \text{open} or \text{update} member function to later change the access mode from to read to update.

If the object is an instance of a versionable class and has versioning enabled, a new version of the object is created when the object is opened in update mode.

See “Creating a Version” on page 16-6 for more details on creating a new version.

Example

In this example, the federated database named ECAD and the database named EPROM are opened in update mode. The handles rectH1 and rectH2 are set to reference basic objects located in the container referenced by handle rectContH.

Note that since the object referenced by the handle rectH2 is not modified, it is implicitly opened in read mode by Objectivity/DB when it is dereferenced.

See “Using the Member Function operator ->” on page 4-3 for information on accessing persistent objects using the arrow operator, ->.

\[
\text{ooHandle(ooFDObj)} \text{ fdH;}
\text{ooHandle(ooDBObj)} \text{ dbH;}
\text{ooHandle(ooContObj)} \text{ rectContH;}
\text{ooHandle(Rectangle)} \text{ rectH1; \ // \ Rectangle \ is \ a \ subclass \ of \ ooObj}
\text{ooHandle(Rectangle)} \text{ rectH2;}
\]
fdH.open("ECAD", oocUpdate); // Open FD for update
dbH.open(fdH, "EPROM", oocUpdate); // Open database for update

// rectContH is set to reference container in database EPROM, and
// rectH1, rectH2 are set to reference basic objects in the
// container
rectContH.open(dbH, "rectangle", oocUpdate);

rectH1.open(oocUpdate); // Open basic object for update
rectH1->x = 0; // Update basic object
// Access object for read
printf("Value of x is %d\n", rectH2->x);

---

**Determining the Current Open Mode**

To determine if an object has been opened from a given object reference or handle, and if so in what mode, you can use the following member function:

- `ooMode ooRefHandle(className)::openMode() const;`

where

- `className` : `ooObj`, `ooContobj`, `ooDBObj`, or `ooFDObj`

This member function returns the access mode in which the object was opened. Possible return values are the constants `oocRead` (open in read mode), `oocUpdate` (open in update mode), and `oocNoOpen` (not opened from this object reference or handle).
Upgrading Access from Read to Update

Once a persistent object has been opened in read mode, you may promote the access to update mode by opening the object again for update access. There is no need to first close the object.

For all persistent objects you may use the `open` member function and specify the constant `oocUpdate`. You can also use the `update` member function.

When updating an object, you should change the object’s access mode to update mode before making any changes to the object. Otherwise, changes you make may be lost.

Example

In this example, the basic object pointed to by `rectH` is opened with read access implicitly when it is accessed the first time using the arrow operator, `->`. Its access mode is then promoted to update.

```cpp
ooHandle(Rectangle) rectH; // Rectangle is a subclass of ooObj
...
// rectH is set to a valid handle value.
...
rectH->draw();
rectH.update();
rectH->layerN = 10;
```

Using `ooUpdate` Within a Member Function

When writing a member function within a persistent class, you can use the `ooUpdate` member function defined on the persistent class `ooObj` to indicate the intention to update an object. The syntax for this member function is:

- `ooStatus ooUpdate();`
Closing Objects

Example

Using the ooUpdate member function:

```cpp
class Inventory : public ooObj {
    public:
        void change(int newCount) {
            ooUpdate(); // Set to access mode of the
            // basic object to
            // oocUpdate.
            count = newCount;
        }
    private:
        int count;
};
```

Closing Objects

Objectivity/DB will automatically close any object when its object reference or handle goes out of scope or has been reassigned to point to another object. However, you can explicitly close an object using the following member function:

```
bool ooStatus ooRefHandle(className)::close() const;
```

where

- `className` is `ooObj`, `ooContobj`, `ooDBObj`, or `ooFDObj`

Once an object is closed, Objectivity/DB may remove the object from virtual memory if necessary. If you wish to access the object after it is closed, you must open it again.

Closing a federated database will also close any open persistent objects it contains. You cannot open a different federated database within a process even if you have closed the first one. That is, you can only open one federated database per process, although you can open and close the same federated database as often as you like.
To keep virtual memory requirements low, it is good design practice to keep iterators open while you are using them, and close objects when you no longer need them. See the “Using Iterators” chapter for information about iterators.

`close` does not affect locking; that is, it does not release the lock on the object.

---

**Example**

This example demonstrates reopening an object after it has been closed. In this case, the object indicated by the handle `polyH` is reopened implicitly for read access.

```cpp
ooHandle(Polygon) polyH;
...
// Obtain a valid handle for polyH.
...
// Open and close object explicitly.
polyH.open(oocUpdate);
polyH->set_origin(0,0,5,5);
polyH.close();

// Open object implicitly and close explicitly.
polyH->draw();
polyH.close()
```

---
Closing Objects
Copying and Moving Basic Objects

This chapter describes how to copy and move basic objects between containers. Where syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle. Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form const ooHandle(className)& are interchangeable with const ooRef(className)&.

Copying a Basic Object

You can create a copy of a basic object using the copy member function. During the copy operation, Objectivity/DB automatically invokes the virtual function ooCopyInit. You can define ooCopyInit for your classes to perform copy constructor or object assignment operations when an object is copied, or to propagate the copy operation across association links.

Using copy

The syntax descriptions for copy are as follows:

- ooRef(ooObj)& ooRefHandle(ooObj)::copy(
  const ooHandle(ooObj)& nearH,
  ooRef(ooObj)& newObjId) const;
- ooHandle(ooObj)& ooRefHandle(ooObj)::copy(
  const ooHandle(ooObj)& nearH,
  ooHandle(ooObj)& newObjH) const;
- ooHandle(ooObj) ooRefHandle(ooObj)::copy(
  const ooHandle(ooObj)& nearH) const;
where

nearH is the handle of the object to use as a clustering directive. If you specify a database handle, copy places the copy of the basic object in the default container of the specified database. If you specify a container handle, copy places the copy of the basic object in the specified container. If you specify a basic object handle, copy places the copy of the basic object near the specified basic object if space is available, and if not, in the same container.

newObjID is an object reference to the new basic object.

ewObjH is a handle to the new basic object.

The copy member function creates a copy of the basic object and places it in the location specified by the clustering directive. All of the VArrays in the source object are bit-wise copied to the new object. Any associations on the source object are treated according to the copy behavior specifier given for the association when the object class was declared in a DDL schema file (see the “Data Definition Language” chapter of Using Objectivity/ C++ Data Definition Language).

The copy member function performs a shallow copy only, that is, it does not cause propagation of the copy operation across association links. To achieve a deep copy, you can define your own member function that uses the virtual function ooCopyInit. For more information, see “Using ooCopyInit” on page 6-4.

Example

The following DDL schema file defines class A and class B. It specifies that when an object of class A is copied, the association between that object and an object of class B is to be dropped. It also specifies that when an object of class B is copied, the association between that object and an object of class A is to be copied to the new object (that is, the new object will also be associated with the same object of class A):

class A : public ooObj {
public
  A(const char* newName) { name = newName; }
  ooVString name;
  ooRef(B) bA[] <-> aA: copy(drop);
}; // drop is the copy behavior specifier
class B : public ooObj {
public:
    B(const char* newName) { name = newName; }
    ooVString name;
    ooRef(A) aA <-> bA[]: copy(copy);
}; // copy is the copy behavior specifier

The following source code copies an object of class A and an object of class B, treating the association links as specified in the class definitions. This code copies the objects to a new container specified by new_locH:

ooHandle(A) original_aH, new_aH;

// New location for copied objects
new_locH = new (dbH) ooContObj();

// Copy an object of class A
// Association is dropped in the new object
original_aH.copy(new_locH, new_aH);

ooHandle(B) original_bH, new_bH;

// Copy an object of class B
// Association is added between original_aH and new_bH
original_bH.copy(new_locH, new_bH);
Using ooCopyInit

The virtual function ooCopyInit is defined for ooObj. This function does nothing except return a success status. The syntax for ooCopyInit is as follows:

```cpp
virtual ooStatus ooCopyInit();
```

You can use ooCopyInit in your application to perform constructor or object assignment operations as part of the copy operation performed by the copy member function. During the copy operation, Objectivity/DB automatically invokes ooCopyInit on the new object after it has successfully created it and made a bitwise copy of the source object.

---

Examples

This example demonstrates how to copy an object and how to write the virtual function ooCopyInit to support a deep copy of objects. To simplify the example, the virtual function is inlined.

The DDL schema file declares classes A, B, and C, and defines the virtual function ooCopyInit in class C to perform a deep copy (that is, to propagate the copy operation across association links between class C and other classes):

```cpp
class A : public ooObj {
  public:
    A(const char* newName) { name = newName; }
    ooVString name;
    ooRef(B) bS[] <-> a: copy(drop);
};
```
class B : public ooObj {
    public:
        B(const char* newName) { name = newName; }
        ooVString name;
        ooRef(A) a <-> bs[]: copy(copy);
    }
}

class C : public ooObj {
    public:
        C() { }
        C(const C& c) { x = c.x; y = c.y; }
        ooRef(A) x;
        ooRef(A) y;
        ooStatus ooCopyInit() { // argument is not used
            // Deep copy
            ooRef(A) original_aH, new_aH;
            if (x) {
                // Copy the object pointed by OID x
                original_aH = x;
                original_aH.copy(original_aH, new_aH);
                x = new_aH;
            }
            if (y) {
                // Copy the object pointed by OID y;
                original_aH = y;
                original_aH.copy(original_aH, new_aH);
                y = new_aH;
            }
        }
    }
An application source code file contains the following code, which copies an object of class A and an object of class B, and deep copies an object of class C:

```c
#include <stdio.h>
#include "test.h"

ooTrans trans;
 ooHandle(ooFDObj) fdH;
 ooHandle(ooDBObj) dbH;
 ooHandle(ooContObj) contH;

 ooInit();
 trans.start();
 fdH.open("SHIP", oocUpdate);
 dbH.open(fdH, "shop", oocUpdate);

 // Assume there is a container with the system name
 // bookstore.
 contH.open(dbH, "bookstore", oocUpdate);

 // Create an object of class A.
 ooHandle(A) aH = new(contH) A("magazine");

 // Create an object of class B.
 ooHandle(B) bH = new(aH) B("sports");

 // Add an association between the objects attached to aH and bH.
 aH->add_bS(bH);

 // Create another object of class B.
 bH = new(bH) B("business");

 // Add another association, to the (new) object attached to bH.
 aH->add_bS(bH);
```
// Copy an object of class A.
 ooHandle(A) new_aH;

// association is dropped in the new object
if (aH.copy(aH, new_aH) == 0)
    printf("Error in copying a new object\n");

// Copy an object of class B.
 ooHandle(B) new_bH;

// association is added between the
// objects attached to aH and new_bH
bH.copy(bH, new_bH);

// Reset the name.
 bH->name = "computer";

// Create an object of class C.
 ooHandle(C) cH, new_cH;
 cH = new(aH) C;

// Deep copy an object of class C.
 cH.copy(cH, new_cH);

// Commit the transaction.
 trans.commit();
Moving a Basic Object

You can move a basic object to a different container using the `move` member function defined on each handle class.

During the move operation, Objectivity/DB automatically invokes the virtual functions `ooPreMoveInit` and `ooPostMoveInit`, which return the status of the move operation. You can define these functions for your classes to perform cleanup or other operations when an object is moved.

Using move

The syntax for `move` is as follows:

```cpp
ooStatus ooRefHandle(ooObj)::move( const ooHandle(ooObj)& targetH);
```

`targetH` is the handle of the object to which to move the basic object. Specifying a database handle moves the basic object to the default container of the specified database. Specifying a container handle moves the basic object to the specified container. Specifying a basic object handle moves the basic object to the container near the specified basic object if space is available; if not, to the same container.

After a successful move, the object reference or handle references the new basic object. If the move is unsuccessful, the object reference or handle remains valid and still references the source basic object.

If the moved basic object has bidirectional associations with other objects, Objectivity/DB maintains referential integrity for those objects. If the moved basic object has unidirectional associations with other objects, your application must maintain referential integrity for those objects.

If you want to move an object whose OID is contained in an map dictionary element, you must remove the element before the object is moved, and then add a new element after the move is successfully completed. For more information about map dictionaries, see the “Using the Map Dictionary Classes” chapter.
Example

This example shows how to move a basic object to a different container.

The following DDL schema file defines class A and class B:

class A : public ooObj {
    public:
        A(const char* newName) { name = newName; }
        ooVString name;
        ooRef(B) bS[] <-> a: copy(drop);
    }

class B : public ooObj {
    public:
        B(const char* newName) { name = newName; }
        ooVString name;
        ooRef(A) a <-> bS[]: copy(copy);
    }

#include <stdio.h>
#include "test.h"
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;
ooHandle(ooContObj) contH, cont2H;

ooInit();
trans.start();
fdH.open("SHIP", oocUpdate);
dbH.open(fdH, "shop", oocUpdate);

// Assume there is a container with the system name bookstore.
contH.open(dbH, "bookstore", oocUpdate);

// Assume there is another container with the system name front.
cont2H.open(dbH, "front", oocUpdate);

// Create a basic object of class A within the container
// referenced by the handle contH.
ooHandle(A) a1H = new(contH) A("magazine");

// Create another basic object of class A within the container
// referenced by the handle cont2H.
ooHandle(A) a2H = new(cont2H) A("magazine");

// Create a basic object of class B near the object referenced
// by a1H.
ooHandle(B) bH = new(a1H) B("sports");

// Move the basic object referenced by the handle a1H to the
// container referenced by the handle cont2H.
a1H.move(cont2H);

// Move the basic object referenced by the handle a2H to the
// container containing the basic object referenced by the handle
// bH.
a2H.move(bH);

// Move the basic object referenced by the handle bH to the
// default container of the database referenced by the handle
// dbH.
bH.move(dbH);

// Commit the transaction.
trans.commit();
Using ooPreMoveInit and ooPostMoveInit

The virtual functions ooPreMoveInit and ooPostMoveInit are defined for the Objectivity/DB-defined class ooObj. These functions do nothing except return the success status. The syntax descriptions are as follows:

- virtual ooStatus ooPreMoveInit();
- virtual ooStatus ooPostMoveInit();

ooPreMoveInit is invoked before the object is moved, while ooPostMoveInit is invoked after the object has been successfully moved. You can use these functions in your application to perform cleanup or other operations.

Example

This example shows the virtual functions ooPreMoveInit and ooPostMoveInit for a persistent class A. To simplify the example, the virtual functions are inlined.

class A : public ooObj {
public:
    int32 size
    ooRef scope;
    ooStatus ooPreMoveInit() {
        ooHandle(A) aH;
        ooHandle(B) bH = scope; // B is also a persistent class
        ooThis(aH);
        aH.unnameObj(bH); // remove the scope name
    }
    ooStatus ooPostMoveInit() {
        ooHandle(A) aH;
        ooHandle(B) bH = scope;
        ooThis(aH);
        aH.nameObj(bH, "xxx"); // name the object
    }
};
Moving a Basic Object
Using Associations

An association logically links objects. To allow associations, you must define an association link in each object's class. An association can be unidirectional or bidirectional.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`.

Directionality of Associations

A unidirectional association is one in which only one class defines an association link to another class. This means that an object with a unidirectional association can locate its associated object, but the associated object cannot locate the associating object. Unidirectional associations correspond most closely to the use of pointers to link objects in a standard C++ data model.

An association between two objects in which each object's class has an association link to the other class is called a bidirectional association. Bidirectional associations allow you to locate an associated object from either of two associated objects. These associations can be connected and disconnected with a single member function invocation.
Bidirectional associations provide Objectivity/DB with enough information to ensure referential integrity, contributing to the robustness of the system. As with C++ pointers in a standard C++ data model, Objectivity/DB does not enforce referential integrity for unidirectional associations, so use them cautiously. Unidirectional associations do, however, require somewhat less overhead and offer better performance.

In Figure 7-1, objects B and C are linked by a unidirectional association, where the class of object C includes the link declaration. This means that C can locate B, but B cannot locate C, both using the association. Objects A and B are linked by a bidirectional association. This means A can locate B, and B can locate A using the association.

![Figure 7-1  Bidirectional and Unidirectional Associations](image_url)
Number of Associations per Object

A non-inline association between objects is stored in a system default variable-size array (VArray) for each object of a class that declared the link. When open, a VArray must fit into available swap space, so there is an implied limit on the number of associations per object that are allowed.

The formula for determining the size in bytes of a non-inline association VArray for an object is approximately as follows:

\[
\text{Non-inline association size} = (14 \text{ bytes VArray overhead} + (12 \text{ bytes per one side of association link} \times \text{numberOfLinks})) + 4 \text{ byte short OID from an object to the association VArray}
\]

where

\[
\text{numberOfLinks} \quad \text{All non-inline association types defined for the particular object class}
\]

For information about how Inline associations are stored, see the “Data Definition Language” chapter of Using Objectivity/C++ Data Definition Language.

Default Association Ordering

The default ordering of associations is chronological (based on their creation times).

Association Cardinality

An association’s cardinality is the number of objects on one side of an association that may be associated with objects on the other side. Objectivity/DB associations support four categories of cardinality:

- **1:1** one-to-one
- **1:m** one-to-many
- **m:1** many-to-one
- **n:m** many-to-many

You specify the cardinality of an association when you define a class association link in the database schema.
Using One-to-One and Many-to-One Associations

The following member functions are automatically defined by the DDL Processor for all user-defined derived classes of ooObj and ooContObj that have a one-to-one or many-to-one association link defined (see “Associations” on page 2-16 of Using Objectivity/ C++ Data Definition Language for information on defining association links using DDL).

Getting the Handle of the Object Associated on a to-One Link

The member function `nameOfLink` returns the object reference or handle of the object (of class `className`) associated via the one-to-one or many-to-one association link `nameOfLink`. A null object reference or handle is returned if no association exists on the link.

The following syntax returns the handle by value:

- `ooHandle(className) nameOfLink(`
  const ooMode openMode  = oocNoOpen) const;

The following syntax returns the object reference or handle in the user supplied buffer `objH`:

- `ooRef(className)&  nameOfLink(`
  ooRef(className)& objId,
  const ooMode openMode  = oocNoOpen) const;

- `ooHandle(className)&  nameOfLink(`
  ooHandle(className)& objH,
  const ooMode openMode  = oocNoOpen) const;

where

- `nameOfLink` Name of the association link
- `className` Class of object to which the link points
- `openMode` Access mode in which the object at the other end will be automatically opened. It may be any one of the constants `oocRead`, `oocUpdate`, or `oocNoOpen`. The default value is `oocNoOpen`, that is the object is not automatically opened.
- `objId` Object reference of object to which to set the link
- `objH` Handle of object to which to set the link
Assume the following class declarations are contained in the schema for a federated database named ECAD:

```cpp
class Cell: public ooContObj {
public:
    ...
    ooRef(Layer) layers[] <-> cell; // one-to-many bidirectional
    ooRef(Pad) pad <-> cell;       // one-to-one bidirectional
};

class Layer: public ooObj {
public:
    ...
    ooRef(Cell) cell <-> layers[]; // many-to-one bidirectional
};

class Pad: public ooObj {
public:
    ...
    ooRef(Cell) cell <-> pad;     // one-to-one bidirectional
};
```

The following code demonstrates how to get the handle of an object that is associated to another via a one-to-one association link (pad in class Cell), and a many-to-one association link (cell in class Layer).

```cpp
ooHandle(Cell) cellH, assocCellH;
ooHandle(Layer) layerH;
ooHandle(Pad) assocPadH;
ooHandle(ooFDObj) fdHandle;
ooHandle(ooDBObj) dbHandle;
```
Using One-to-One and Many-to-One Associations

... fdHandle.open("ECAD", oocRead);
  dbHandle.open(fdHandle, "ALU", oocRead);
...

// Assume cellH and layerH are set to valid handle values.
assocPadH = cellH->pad(); // one-to-one
assocCellH = layerH->cell(); // many-to-one

// Check to see if handles are null or not.
if assocPadH == 0 {
  printf("An association does not exist on link pad\n");
}
if assocCellH == 0 {
  printf("An association does not exist on link cell\n");
}

Handle assocPadH is set to point to the object of class Pad that is associated to the object of class Cell indicated by handle cellH. Also, handle assocCellH is set to point to the object of class Cell associated to the object indicated by handle layerH.

Setting an Association on a to-One Link

Use the member function set_nameOfLink to set the one-to-one or many-to-one association link nameOfLink with an association to the object referenced by objH.

- ooStatus set_nameOfLink(const ooHandle(className) & objH);

where

nameOfLink Name of the association link
className Class of object to be associated to
objH Handle of object to set link to

If an association already exists, you must first delete it before setting it. You can delete it by calling del_nameOfLink or by calling set_nameOfLink with objH as null. If this is a bidirectional association, both this link and its inverse are updated.
at the same time. This member function promotes the access mode of the associated objects to update mode.

**Deleting an Association on a to-One Link**

Use the member function `del_nameOfLink` to delete the association on association link `nameOfLink`.

```cpp
ooStatus del_nameOfLink();
```

where

- `nameOfLink` Name of the association link

If the association is bidirectional, both this link and its inverse are updated at the same time.

This member function promotes the access mode of the associated objects to update mode.

**Using One-to-Many and Many-to-Many Associations**

The following member functions are automatically defined by the DDL Processor for persistent classes derived from `ooObj` and `ooContObj` that have a one-to-many or a many-to-many association link defined.

**Initializing an Iterator for Associations on a to-Many Link**

Use the following member function to initialize an iterator to traverse the objects associated via the one-to-many or many-to-many link. The iterator is returned empty (that is, the first `next` returns a null handle) if no associations currently exist:

```cpp
ooStatus nameOfLink( ooItr(className) &itr,
    const ooMode openMode = oocNoOpen) const;
```

where

- `nameOfLink` Name of the association link
- `className` Class of objects to which the link points
- `itr` Iterator to be initialized
openMode

Mode that each object is opened when \textit{itr.next} is invoked. May be any of the constants \texttt{oocRead}, \texttt{oocUpdate}, or \texttt{oocNoOpen}. Default value is \texttt{oocNoOpen}; that is, the objects are not automatically opened.

### Adding an Association to a \texttt{to-Many} Link

Use the member function \texttt{add\_nameOfLink} to add an association to the object referenced by \texttt{objH} to the one-to-many or many-to-many association link \texttt{nameOfLink}.

\begin{verbatim}
void add\_nameOfLink(const ooHandle<className>& objH);
\end{verbatim}

where

- \texttt{nameOfLink} Name of the association link
- \texttt{className} Class of objects to which the link is associated
- \texttt{objH} Handle of object to add to link

No action is performed if \texttt{objH} is null. If this is a bidirectional association, both this link and its inverse are updated. This member function promotes the access mode of the associated objects to update mode.

Objectivity/DB does not perform redundancy checking for associations. Thus it is possible to use \texttt{add\_nameOfLink} to create duplicate associations between the same two objects (even though it could be semantically meaningless to do so).

### Deleting Specified Associations from a \texttt{to-Many} Link

Use the member function \texttt{sub\_nameOfLink} to remove the single association or a specified number of associations to the object referenced by \texttt{objH} from the one-to-many or many-to-many link \texttt{nameOfLink}.

\begin{verbatim}
void sub\_nameOfLink(const ooHandle<className>& objH, const uint32 number = 1);
\end{verbatim}

where

- \texttt{nameOfLink} Name of the association link
- \texttt{className} Class of objects to which the link is associated
- \texttt{objH} Handle of object to remove from link
number  Number of associations to be removed. If you specify 0, all associations to the object are removed. If you specify a number greater than 1, the first number associations encountered are removed. The default value of number is 1.

No action is performed if objH is null. If the association is bidirectional, both this link and its inverse are updated. This member function promotes the access mode of the associated objects to update mode.

Deleting All Associations from a to-Many Link

Use the member function del_nameOfLink to delete all associations from link nameOfLink.

```cpp
doStatus del_nameOfLink();
```

where

nameOfLink  Name of the association link

If the association is bidirectional, both this link and its inverse are updated at the same time.

This member function promotes the access mode of the associated objects to update mode.

Example

Assume the following class declarations are contained in the schema for a federated database named ECAD:

```cpp
class Cell: public ooContObj {
public:
    ...
    ooRef(Layer) layers[] <-> cell;//one-to-many bidirectional
};

class Layer: public ooObj {
public:
    ...
    ooRef(Cell) cell <-> layers[];//many-to-one bidirectional
```
The following code demonstrates how to initialize an iterator for a one-to-many association link, and also how to add another object handle to the link.

In this example, iterator `layerI` is initialized to traverse the association link `layersAssoc` of the object indicated by `cellH`. The object handle `layerH` is added to the link.

```c++
#include test.h

ooHandle(Cell) cellH;
ooHandle(Layer) layerH;
ooItr(Layer) layerI;
ooHandle(ooFDObj) fdHandle;
ooHandle(ooDBObj) dbHandle;

... 
fdHandle.open("ECAD", oocRead);
dbHandle.open(fdHandle, "ALU", oocRead);
...

// Assume cellH and layerH are set to valid handle values.
...

cellH->layers(layerI); // Initialize iterator

cellH->add_layers(layerH); // Add an association

... 
```

Checking for the Existence of an Association

You can check for the existence of an association between two objects on an association link using the member function `exist_nameOfLink`. The syntax of this member function is as follows:

```c
ooBoolean exist_nameOfLink(
    const ooHandle<className>& objH) const;
```

where

- `nameOfLink` Name of the association link
- `className` Class of objects to which the link is associated
- `objH` Handle of the object

This member function returns `oocFalse` if there are no associations between the two specified objects on the link, otherwise it returns `oocTrue`. If `objH` is 0, then the function only checks for the existence of an association.

Composite Objects and Propagating Operations

You can define associations so that a federated database operation will propagate from one object to the next along the association. You can propagate the following operations:

- Deletion
- Explicit locking

Propagation is a very useful property when you wish to treat a group of associated objects as a single composite object. Composite objects are constructed from simpler objects using associations. An operation on a composite object can be propagated to all of its associated objects.

For example, in Figure 7-2, several objects are related by both propagating and non-propagating associations. Here, only objects A, B, C, and H are composite objects. The others are basic objects and do not propagate operations along their associations.

Assuming the propagating associations in this example support the delete operation, objects A, B, C, D, F, and G will be deleted when the delete operation is applied to object A.
To specify which operations should propagate and the direction of propagation, you define the propagation behavior of the association links in your DDL schema. For more information see the “Data Definition Language” chapter of *Using Objectivity/ C++ Data Definition Language*.

Once the propagation behavior is defined for an association, you can propagate an operation by calling the appropriate member function, operator, or macro.

To delete with propagation, use either the macro `ooDelete` or operator `delete`. If you want to prevent propagation of the delete operation for a propagating association, use the `ooDeleteNoProp` macro. For more information, see “Deleting Databases, Containers, and Basic Objects” on page 3-18.

To lock an object with propagation, use the `lock` member function. To prevent lock propagation, use the `lockNoProp` member function. For more information, see “Explicit Locking” on page 11-12.

![Composite Objects](image)

**Figure 7-2  Composite Objects**
Adding Information to Associations

You may find it desirable to store additional information along with each association. For instance, you may want to record the date and time that an association was made. While associations themselves do not have any provision for storing data, the same effect may be achieved by storing the data in the objects of the association.

one-to-one associations For a one-to-one association, the association data could be defined in the class of either of the two objects that are associated.

one-to-many associations For a one-to-many association, the association data could be defined in the class of the “many” objects of the association.

many-to-many associations For a many-to-many association, the association data could be defined in a basic object class, an instance of which serves as an intermediate object for each association. Thus an instance of this intermediate object class must be created when establishing an association. If desired, you can write member functions to conceal the intermediate object.

This approach is shown in Figure 7-3, where we wish to keep information about each association between instances of class A and class B. To solve the problem, a class C is defined with the necessary information, and the appropriate association links are defined on class A, class B, and class C.
Figure 7-3  Adding Information to a Many-To-Many Association
Naming Objects

This chapter describes system names and scope names, and how to use them to identify Objectivity/DB objects.

Where the syntax and parameter information is identical for the object reference and handle classes, `ooRefHandle` is used to represent either `ooRef` or `ooHandle`.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className)&` are interchangeable with `const ooRef(className)&`.

Using System Names

A system name serves a purpose similar to a file name. A file name uniquely identifies a file to the operating system, and a system name uniquely identifies a persistent object to Objectivity/DB. You can give system names to federated databases, databases, and containers. Each of these objects can have only one system name.

Federated databases and databases are required to have system names. For these objects, system names are used to determine physical file names for the objects. See the “Objectivity/DB Basics” chapter of Objectivity/DB Administration for details.

System names are optional for containers, since they are not represented as separate physical files.

You assign an object a system name when you create the object. In general, this name exists for the lifetime of the object. If the named object is deleted, its system name will also be deleted. If necessary, your database administrator may be able to change an object’s system name by using a database administration tool. For more information, see Objectivity/DB Administration.
The system name for an object must be unique to all others for objects in the same containing object. For example, no two containers in the same database can have the same system name. System Names follow the same naming conventions as file names within the host operating system.

**Setting System Names**

A system name is set at the time an object is created. For a container you can use the operator `new` to specify a system name.

See “Creating a Federated Database” on page 3-2, “Creating Databases” on page 3-3, and “Creating Containers” on page 3-7 for more information.

**Obtaining an Object’s System Name**

Use the following member function to obtain an object’s system name:

```cpp
char *ooRefHandle(className) :: name() const;
```

where

```cpp
className ooContObj, ooDBObj, or ooFDObj
```

This member function returns a pointer to a string containing the name of the object. If the object does not have a system name, the pointer will be null ((char *)0). The string is statically allocated by the member function and is overwritten with each invocation of `name`.

Since the string returned by `name` is statically allocated and overwritten with each invocation, you should always make a local copy of the returned string if you intend to use it later in your application.

**Object Lookup Using System Names**

Lookup of objects using system names is achieved through the use of the `open` member function. See “Opening a Container or Basic Object” on page 5-6 for details.
Checking for Object Existence

Use the \texttt{exist} member functions to determine the existence of objects. If you want \texttt{exist} to test for an object's existence, and try to open the object for update or read, use the \texttt{oocUpdate} or \texttt{oocRead} open mode, respectively. In this case, a false return status from \texttt{exist} indicates that either the object does not exist or that an error occurred when \texttt{exist} tried to open the object. Use this member function with an open mode of \texttt{oocNoOpen} whenever you only want to test for the existence of an object and not open it. The \texttt{exist} function returns a value of true if the object exists and false if it does not.

Table 8-1 summarizes the effects of the \texttt{exist} member function on objects for various open mode settings.

<table>
<thead>
<tr>
<th>\texttt{ooMode}</th>
<th>\texttt{Return Value for exist}</th>
<th>\texttt{Object Exists and is accessible}</th>
<th>Effect on Object</th>
<th>Value of Handle or Object Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{oocRead}</td>
<td>\texttt{oocTrue}</td>
<td>Yes</td>
<td>Opened for read</td>
<td>The object</td>
</tr>
<tr>
<td>\texttt{oocRead}</td>
<td>\texttt{oocFalse}</td>
<td>No</td>
<td></td>
<td>null</td>
</tr>
<tr>
<td>\texttt{oocUpdate}</td>
<td>\texttt{oocTrue}</td>
<td>Yes</td>
<td>Opened for update</td>
<td>The object</td>
</tr>
<tr>
<td>\texttt{oocUpdate}</td>
<td>\texttt{oocFalse}</td>
<td>No</td>
<td></td>
<td>null</td>
</tr>
<tr>
<td>\texttt{oocNoOpen}</td>
<td>\texttt{oocTrue}</td>
<td>Yes</td>
<td>Not opened</td>
<td>The object</td>
</tr>
<tr>
<td>\texttt{oocNoOpen}</td>
<td>\texttt{oocFalse}</td>
<td>No</td>
<td></td>
<td>null</td>
</tr>
</tbody>
</table>
Checking for the Existence of a Federated Database

Use the following member function to determine if a federated database exists.

❑ ooBoolean ooRefHandle(ooFDObj)::exist(  
    const char *bootFilePath,
    const ooMode openMode = oocNoOpen);

where

bootFilePath       Path to the boot file of the federated database
openMode           Open mode of the federated database if it exists

This member function returns the constant oocTrue if the federated database exists, and the constant oocFalse if it does not.

Checking for the Existence of a Database

Use the following member function to determine if a database exists.

❑ ooBoolean ooRefHandle(ooDBObj)::exist(  
    const ooHandle(ooFDObj)& fdH,
    const char *dbSysName,
    const ooMode openMode = oocNoOpen);

where

fdH               Handle of federated database that contains the database
dbSysName         System name of the database
openMode          Open mode of the database if it exists

This member function returns the constant oocTrue if the database exists, and the constant oocFalse if it does not.
Checking for the Existence of a Container

Use the following member function to determine if a container exists.

```cpp
❑ ooBoolean ooRefHandle(ooContObj)::exist(
    const ooHandle(ooDBObj) & dbH,
    const char * contName,
    const ooMode openMode = oocNoOpen);
```

where

- `dbH` Handle of database that contains the container
- `contName` System name of the container
- `openMode` Open mode of the container if it exists

This member function returns the constant `oocTrue` if the container with the given system name exists, and the constant `oocFalse` if it does not.

Example

This example opens a federated database with the system name `shapeExample`, checks to see if a database with the system name `simpleShapes` exists, and if so it retrieves the database name in `dbName`.

```cpp
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;
char* dbName;
...
fdH.open("shapeExample", oocUpdate);
if (!dbH.exist(fdH, "simpleShapes"))
    dbH = new(fdH) ooDBObj("simpleShapes");
else
    ...
dbName = dbH.name();
```

Naming Objects 8-5
Using Scope Names

A scope name is a name that uniquely identifies an object within the name scope of itself or other objects. A name scope is an individual object’s set of scope names. Therefore, there are as many name scopes as there are individual objects.

For example, suppose we have a family of three people. If the mother of the family, Mary, is called Darling by her husband and Mom by her child, she has three scope names. In her own name scope, her name is Mary, in the scope of her husband her name is Darling, and in the scope of her child her name is Mom.

Within Objectivity/DB, any persistent object can serve as a name scope for other objects. However, only containers and basic objects can actually have scope names. Therefore, although federated databases and databases can serve as name scopes for basic objects and containers, they cannot have scope names themselves.

A container or basic object can have no more than one scope name within a particular name scope, and all scope names within a particular name scope must be unique.

Scope names are completely optional and can be set and unset dynamically at run time by the application. Valid scope names are null-terminated strings that can contain any non-null character.

If a container (or any basic objects it contains) is used as a name scope for another object, the container must be a hashed container. For an object named in the scope of a database (or federated database), the hash pages of the database's default container are used. For more information about hashed containers, see “Hashed Containers” on page 3-7.
Setting Scope Names

Use the member functions described in this section to work with scope names. The functions are defined on the object handle class for ooObj, ooContObj, and their subclasses.

Giving an Object a Scope Name

Use the following member function to give an object a scope name:

```cpp
ooStatus ooRefHandle(ooObj)::nameObj(
    const ooHandle(ooObj) & scopeH,
    const char *scopeName) const;
```

where

- `scopeH` Handle of an object that serves as the scope for the name.
- `scopeName` Scope name to give the object. This name cannot exceed 500 characters.

This member function gives an object the specified name in the scope of the object referenced by `scopeH`. An object is allowed only a single name within a given scope.

Removing an Object’s Scope Name

Use the `unnameObj` member function to remove the scope name of an object.

```cpp
ooStatus ooRefHandle(ooObj)::unnameObj(
    const ooHandle(ooObj) & scopeH,
    const char *scopeName = 0) const;
```

where

- `scopeH` Handle of an object that serves as the scope for the name
- `scopeName` Scope name to remove. This argument is optional. If you do not specify a scope name, it defaults to null

It is more efficient to use a system name for a container than to use a scope name with the container’s containing database as a scope. Doing so also provides better database concurrency.
Be aware that invoking either the `nameObj` or `unnameObj` member function will cause Objectivity/DB to automatically secure update access to the scope object and the named object for the remainder of the transaction.

Example

This example changes the scope name of the object referenced by handle `cellH` in the scope of the objects referenced by `obj2H` and `obj3H` from `current` to `old`.

```c++
ooHandle(Cell) cellH; // Cell is a subclass of ooObj
ooHandle(ooContObj) obj2H, obj3H;
...
// cellH, obj2H, obj3H are set to reference objects.
...
// Give object the name current in scope of obj2H and obj3H.
cellH.nameObj(obj2H, "current");
cellH.nameObj(obj3H, "current");
...
// Remove the name current from the scope of objects referenced
// by obj2H and obj3H.
cellH.unnameObj(obj2H);
cellH.unnameObj(obj3H);
// Now give object the name old in scope of obj2H and obj3H.
cellH.nameObj(obj2H, "old");
cellH.nameObj(obj3H, "old");
...
```
Obtaining an Object’s Scope Name

Use the following member function to obtain an object’s name:

\[
\text{char* ooRefHandle(ooObj)::getObjName(}
\text{ const ooHandle(ooObj)& scopeH) const;}\]

where

\(\text{scopeH}\) Handle of object used as a name scope

This member function returns a pointer to a string buffer containing the name of the object within the scope specified by \(\text{scopeH}\).

The string returned by \text{getObjName} is statically allocated, and is modified every time \text{getObjName} is invoked. For this reason, you should copy the returned string if you wish to use it later in your application.

Object Lookup Using Scope Names

Use the member functions described in this section to look up objects using scope names.

Getting the Handle of an Object by Its Scope Name

Use the following member function to get the handle of an object by specifying its scope name:

\[
\text{ooStatus ooRefHandle(ooObj)::lookupObj(}
\text{ const ooHandle(ooObj) &scopeH,}
\text{ const char *scopeName,}
\text{ const ooMode openMode = oocRead) const;}\]

where

\(\text{scopeH}\) Handle of object used as name scope

\(\text{scopeName}\) Scope name of the object in scope of \text{scopeHandle} to look up

\(\text{openMode}\) Mode in which the object is opened. May be any of the constants \text{oocRead, oocUpdate}, or \text{oocNoOpen}. This is an optional parameter, and if not specified it defaults to \text{oocRead}.
This member function sets the handle to that of the object named scopeName in the scope specified by scopeH, and opens the object in the specified mode. oocSuccess is the returned value if the object is found. This member function is not type safe.

### Initializing an Iterator to Traverse an Object’s Name Scopes

Use the getNameScope member function to initialize an iterator to traverse the objects that serve as name scopes of the specified object.

```c++
ooStatus ooRefHandle(ooObj)::getNameScope(
   ooItr(ooObj) &objI) const;
```

Where:

- `objI` - Iterator initialized to traverse objects that serve as name scopes

The `getNameScope` member function initializes the iterator `objI` to traverse all objects in the federated database that serve as name scopes for this object.

#### Example

The following code example demonstrates the use of the `lookupObj` and the `getNameScope` member functions.

```c++
ooHandle(Person)   personH;   // Person is a subclass of ooObj
ooHandle(Person)   maryH, husbandH, kidH;
ooItr(Person)      personI; // Person iterator
...
// maryH, husbandH, and kidH are set to reference Person Objects
...
// Give the object referenced by maryH a name darling in the
// scope of husbandH
maryH.nameObj(husbandH, "darling");
...
// Give the object referenced by maryH a name mom in the scope
// of kidH
maryH.nameObj(kidH, "mom");
```
Using Scope Names

// Look up the object with the name mom in the scope of kidH, // personH will be pointing to the object referenced by maryH personH.lookupObj(kidH, "mom", oocRead);

// Initialize the iterator to traverse all the objects that serve // as a name scope for the object referenced by maryH // The iterator should return the objects referenced by husbandH // and kidH maryH.getNameScope((ooItr(ooObj)&) personI);

Finding Named Objects in a Given Name Scope

Use the following member function to initialize an iterator to find all named objects in a given scope:

- ooStatus ooRefHandle(ooObj)::getNameObj( ooItr(ooObj) &objI) const;

where

- objI Iterator to be initialized

You cannot use getNameObj on a handle for a federated database or database.

Example

// Person is a subclass of ooObj ooHandle(Person)   maryH, spouseH, kidH; ooItr(Person)      childrenI; // Person iterator …

// maryH, spouseH, and kidH are set to reference Person objects. // Give object referenced by maryH name darling in // scope of spouseH. maryH.nameObj(spouseH, "darling");
// Give object referenced by kidH name Fred in scope of maryH.
kidH.nameObj(maryH, "Fred");

// Initialize iterator to traverse all named objects in
// scope of maryH. Iterator should return at least the
// objects referenced by kidH and possibly others.
maryH.getNameObj((ooItr(ooObj)&) childrenI);

while (childrenI.next()) {
    printf("Child is named %s\n", childrenI.getObjName(maryH));
}

_________
Using Iterators

This chapter describes how to traverse groups of objects using iterators.

An iterator is a non-persistent programming construct that allows you to traverse a collection of objects. For example, you can use an iterator to traverse all persistent objects that:

- Are of a particular class and its subclasses
- Belong to a one-to-many or many-to-many association
- Are one level lower in the storage hierarchy
- Have a scope name for an object
- Are named in a given scope
- Meet the conditions in a predicate

Every iterator class is a subclass of an object reference or handle class. The DDL Processor automatically declares an iterator subclass for every object reference or handle class declaration. Therefore, an iterator for a particular persistent class can invoke any of the member functions defined on the object reference or handle class.

Where the syntax and parameter information is identical for the object reference and handle classes, \texttt{ooRefHandle} is used to represent either \texttt{ooRef} or \texttt{ooHandle}.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form \texttt{const ooHandle(className)\&} are interchangeable with \texttt{const ooRef(className)\&}.

Figure 9-1 illustrates the steps for using an iterator.
Figure 9-1  Using an Iterator

Iterators are not ordered. Therefore, the order in which objects of an iteration are traversed is not fixed, and may vary. You can nest iterators (see “Nesting Iterators” on page 9-11) and can pass an iterator as an argument to a function, which can then continue the iteration without having to know where the iterator originated. If a new element is added to a container or association on which an iterator is currently operating, the new element may not appear in the remainder of the iteration.

You may delete the current element of an iteration (using the handle returned by the iterator) without affecting the remainder of the iteration. However, deleting the container or the association the iterator is operating on will result in a program error.

In general, you should always pass handles and iterators by reference, to save any unnecessary copying and housekeeping that would be triggered if they were passed by value.

9-2  Using Objectivity/C++
When an iterator has returned all the object handles of a traversal, the iterator will return the constant `oocFalse`. Applying `next` to an empty iterator does nothing (see “Getting an Iteration Handle” on page 9-7).

All iterators allow you to specify the mode in which each object in the iteration will be automatically opened and locked when reached (using the `openMode` parameter).

This approach simplifies programming and maximizes concurrency, but can fail in the middle of an iteration if an object in the iteration has been locked by another process. For this reason, you sometimes may want to lock all of the objects in the iteration before you start your processing.

**Declaring an Iterator**

You declare an iterator in your application as you would for any other non-persistent object. The name of each iterator class is of the following form:

- `ooItr(className)`

where

- `className` Name of the persistent class on which this iterator class is defined

**Initializing an Iterator**

After declaring an iterator, you must initialize it. This section describes the member functions that initialize an iterator: `scan`, `nameOfLink`, `contains`, `getNameScope`, and `getNameObj`.

After you initialize an iterator, its handle is undefined. The handle becomes defined when you call the `next` member function on the iterator.
Using scan

To initialize an iterator to traverse all persistent objects contained in the scanned object whose class either matches or is a derived class of the iterator’s object class, use the following member function:

```cpp
ooStatus ooItr(className)::scan(
    const ooHandle(scopeClass)& scopeH,
    const ooMode openMode = oocNoOpen);
```

where

- `className` Class of the persistent objects to be traversed. This can be `ooObj`, `ooContObj`, `ooDBObj`, or a derived class
- `scopeClass` Class of the scope object to be scanned for target objects. This can be `ooContObj`, `ooDBObj`, `ooFDObj`, or a subclass of `ooContObj`
- `scopeH` Handle of the scope object to be scanned for target objects
- `openMode` Optional parameter that specifies the mode each object is opened in when the next member function is used. May be any of the constants `oocRead`, `oocUpdate`, or `oocNoOpen`. If `openMode` is not specified, the default is `oocNoOpen` (that is, the objects returned by `next` are not automatically opened and only their Object Identifiers (OIDs) are returned).

You can also use an index lookup key with the `scan` member function. This activates an index scan and allows you to traverse the objects that satisfy the criteria specified in the lookup key. You can use this feature to iterate over indexed objects based on the order of their key field values. For more information, see “Initializing an Iterator to Use the Index and Lookup Key” on page 14-23.

The `scan` member function will return `oocSuccess` even if there are no objects of type `scanClass` in the scanned object. In this circumstance, the `next` member function (see “Getting an Iteration Handle” on page 9-7) will return `oocFalse` when you try to get the handle of the first object in the iteration. Therefore, your code should check for the success of `scan` before calling `next`, as illustrated in the following example.
Example

This example iterates over all objects of class `NewWidget` in the database whose handle is in `dbh`, and invokes the member function `widgetMethod` on each object:

```c++
  ooHandle(ooDBObj) dbH;
  ooItr(NewWidget) widgetI;
  ooStatus rc;
  ...
  // dbH is initialized here
  ...
  rc = widgetI.scan(dbH);
  if (rc != oocSuccess)
    cerr << "Scan operation failed." << endl;
  else {
    // next member function returns non-zero value
    // if widget is found, including the first one
    while (widgetI.next()) {
      widgetI->widgetMethod();
    }
  }
```
Using `nameOfLink`

To initialize an iterator to traverse the objects in a one-to-many or many-to-many association link, use the following member function, which is defined by the DDL Processor for the persistent class that owns the association link:

```cpp
ooStatus userClassName::nameOfLink(
    ooItr(className)& objI,
    const ooMode openMode = oocNoOpen) const;
```

where

- `nameOfLink` Name of the association link
- `className` Class of the linked objects
- `objI` Name of iterator to initialize

See “Using One-to-Many and Many-to-Many Associations” on page 7-7 for detailed information and an example.

Using `contains`

The `contains` member function initializes an iterator to traverse all persistent objects that are one level lower in the storage hierarchy. For example, the `contains` member function invoked on a federated database object reference or handle initializes an iterator to traverse the databases it contains.

See “Using the `contains` Member Function” on page 9-13 for detailed information on the `contains` member function.

Using `getNameScope`

To initialize an iterator to traverse all the persistent objects that have a scope name for an object, use the following member function:

```cpp
ooStatus ooRefHandle(ooObj)::getNameScope(
    ooItr(ooObj)& objI) const;
```

where

- `objI` Iterator to be initialized

See “Object Lookup Using Scope Names” on page 8-9 for detailed information and an example.
Getting an Iteration Handle

Using `getNameObj`

To initialize an iterator to find all named objects in a given scope, use the following member function:

- `ooStatus ooRefHandle(ooObj)::getNameObj(
  ooItr(ooObj)& objI) const;`

See “Finding Named Objects in a Given Name Scope” on page 8-11 for detailed information and an example.

You cannot use `getNameObj` directly on a handle for a federated database or database.

Getting an Iteration Handle

To set the iterator to the next handle of the iteration, use the following member function:

- `ooBoolean ooItr(className)::next();`

where

- `className` The iterator class. This can be `ooObj`, `ooContObj`, or `ooDBObj` or a derived class.

This member function returns the constant `oocFalse` if there are no more handles in the iteration, otherwise it returns the constant `oocTrue`.

After initializing the iterator, you must invoke the `next` member function to access the first handle of the iteration.
Example

This example initializes the iterator `layerI` to traverse the objects associated via the association link `layers`, and invokes the `draw` member function on the first object of the iteration.

```cpp
// Assume class Cell contains the association link:
// ooHandle(Layer) layers[] <-> cell;
ooItr(Layer) layerI;
ooHandle(Cell) cellHandle;

...  
// cellHandle is set to a valid handle value
...
cellHandle->layers(layerI);
layerI.next();
layerI->draw();
...
```
Accessing an Object In the Iteration

Once you have initialized an iterator, you can use it to access the object it currently references with the following syntax:

- `iterator->member`

See “Invoking Member Functions” on page 4-9 for more information.

---

Example

This example initializes the iterator `layerI` to traverse the association link `layers`, and invokes the `draw` member function on the first object of the iteration. It also upgrades access to the object to update mode.

```cpp
// Assume class Cell contains the association link:
// ooRef(Layer) layers[] <-> cell;

ooItr(Layer) layerI;
 ooHandle(Cell) cellH;
...
// cellH is set to a valid handle value
...
 cellH->layers(layerI);
 layerI.next();
 layerI->draw()
 layerI.update();               // Upgrade access
 layerI->set_origin(0,0,5,5);
```
Iteration Techniques

To initialize and use an iterator, you can use the `while` or `for` statements.

Using the `while` Statement

The following example shows how to initialize and use an iterator in a `while` loop. In this case, the iterator is initialized using the `scan` member function, although it could be initialized with any of the initialization member functions described above.

Example

```c
ooItr(ooObj) objI;
ooHandle(ooContObj) contH;
...
// contH is set to a valid container handle
...
objI.scan(contH); //initialize the iterator
while (objI.next()) { //loop over each element
    ...
}
```

Using the `for` Statement

The following example shows how to initialize and use an iterator in a `for` loop. In this case, the iterator is initialized using the `scan` member function, with any of the initialization member functions described above.

The initializer of the `for` statement also initializes the iterator.

The conditional test of the `for` statement actually places the next handle in the iterator. This works because the `next` member function returns the constant `oocTrue` when there still exists a handle to return. The value of the constant `oocTrue` is nonzero, and the constant `oocFalse` equals zero (the C++ `for` statement executes the statement for the conditional test).
There is no increment clause to the for statement because the conditional test takes care of this too.

---

**Example**

```c
ooItr(ooObj) objI;
ooHandle(ooContObj) contH;
...
// contH is set to a valid container handle
...
for (objI.scan(contH); objI.next();){
    ...
}
```

---

**Nesting Iterators**

You can nest iterators to iterate over objects of different types within the same while or for loop.

---

**Example**

```c
ooItr(ooContObj) contI;
ooHandle(ooDBObj) dbH;
...
// dbH is set to a valid database handle

dbI.scan(dbH);
```
while (dbI.next()) {
    ...
    ooItr(ooContObj) contI;
    contI.scan(...);
    while (contI.next()) {
        ...
    }
    ...
}

**Iteration and Concurrency**

Objectivity/DB uses two-phase locking to maintain the consistency of transactions. To improve concurrency, Objectivity/DB performs a special locking procedure for iterator-based transactions searching for one of the following:

- Database in a federated database
- Container in a database

The special locking procedure locks and unlocks objects during the iteration, which increases concurrency significantly.

If your application requires repeatable read operations during one of the iterations described above, you should explicitly lock the iteration object (database or container) using the `lock` member function. See “Explicit Locking” on page 11-12.

**Iterating on a Federated database**

When iterating to get the next database in a federated database, Objectivity/DB will automatically release the lock on the current database provided the following two conditions are met:

- The database is implicitly opened in read mode during the iteration. For example, at the start of the iteration the database is not open, and the `scan` or `contains` member function is called to open and scan the database.
- There are no open container or basic objects within the database.
Iterating on a Database

When iterating to get the next container in a database, Objectivity/DB will automatically release the lock on the current container provided the following two conditions are met:

- The container is implicitly opened in read mode during the iteration. For example, at the start of the iteration the container is not open, and scan or contains is called to open and scan the container.
- There are no open basic objects within the container.

Determining the Containment Hierarchy

There are two member functions to determine the containment hierarchy of an object — contains and containedIn. These member functions are discussed in the following sections.

Using the contains Member Function

The contains member function initializes an iterator to traverse only those objects at the next level of the storage hierarchy of the containing object. Thus:

- Invoking contains on a federated database object reference or handle initializes the iterator to traverse databases only:
  ```cpp
  ooStatus ooRefHandle(ooFDObj)::contains(
      ooItr(ooDBObj)& objI,
      const ooMode openMode = oocNoOpen) const;
  ```
- Invoking contains on a database object reference or handle initializes the iterator to traverse containers only:
  ```cpp
  ooStatus ooRefHandle(ooDBObj)::contains(
      ooItr(ooContObj)& objI,
      const ooMode openMode = oocNoOpen) const;
  ```
- Invoking contains on a container object reference or handle initializes the iterator to traverse basic objects only:
  ```cpp
  ooStatus ooRefHandle(ooContObj)::contains(
      ooItr(ooObj)& objI,
      const ooMode openMode = oocNoOpen) const;
  ```
where

\( objI \)  
Iterator to initialize

\( openMode \)  
Mode in which each object is opened when \( objI\texttt{.next} \) is invoked. This mode may be any of the constants \( \text{oocRead} \), \( \text{oocUpdate} \), or \( \text{oocNoOpen} \). \( openMode \) is optional and defaults to \( \text{oocNoOpen} \), that is, the objects are not opened when the \( \text{next} \) member function is invoked.

When iterating over a database, the iterator must be of type \( \text{ooContObj} \). Therefore, \( \text{contains} \) will return all basic objects within the database. If you want to iterate over basic objects of a particular subclass of \( \text{ooContObj} \), you can use type-selection member functions such as \( \text{ooIsKindOf} \) and \( \text{ooTypeN} \) within the iteration loop. See “Determining an Object’s Class” on page 20-4. The most efficient way to iterate over basic objects of a particular subclass is to use \( \text{scan} \) instead of \( \text{contains} \).

Similarly, when iterating over a basic object, the iterator must be of type \( \text{ooObj} \). Therefore, \( \text{contains} \) will return all basic objects within the basic object.

Examples

You may need to do explicit typecasting when you use an iterator; for example:

```c++
 ooHandle(ooContObj) contH;
 ooItr(Part)         partI;    //Part is a persistent class

// Only Part objects are contained in the container.
contH.contains((ooItr(ooObj)&) partI);
```

This example iterates over all basic objects in the database referenced by handle \( \text{dbh} \).

```c++
 ooHandle(ooDBObj) dbH;
 ooItr(ooContObj) contI; // iterator must be of type ooContObj
 ooStatus rc;
...
// dbH is initialized to a valid handle
```
... 
rc = dbH.contains(contI);
if (rc) {
    while (contI.next()) {
        ...
    }
}

This example iterates over all basic objects in the database referenced by handle dbH, and then selectively filters out basic objects of type Cell, which is derived from ooContObj.

ooHandle(ooDBObj) dbH;
ooItr(ooContObj) contI; // iterator must be of type ooContObj
ooStatus rc;
...
// dbH is initialized to a valid handle
...
rc = dbH.contains(contI);
if (rc) {
    while (contI.next()) {
        if (contI.typeN() == ooTypeN(Cell)) {
            ...
        }
    }
    ...
}

...
This example iterates over all basic objects in the database referenced by handle `dbH`, and then selectively filters out basic objects of type `Cell` and its derived classes.

```cpp
ooHandle(ooDBObj) dbH;
ooItr(ooContObj) contI; // iterator must be of type ooContObj
ooStatus rc;
...
// dbH is initialized to a valid handle
...
rc = dbH.contains(contI);
if (rc) {
    while (contI.next()) {
        if (contI->ooIsKindOf(ooTypeN(Cell)) {...
        ...
    }
    ...
}
}
```

### Using the containedIn Member Function

The `containedIn` member function returns an object reference or handle to the persistent object in which an object is contained. Thus the following are true:

- **Invoking containedIn on a basic object, object reference or handle returns an object reference or handle to a container:**
  - `ooHandle(ooContObj) ooRefHandle(ooObj)::containedIn() const;
  - `ooHandle(ooContObj)& ooRefHandle(ooObj)::containedIn(ooHandle(ooContObj)&) const;
  - `ooRef(ooContObj)& ooRefHandle(ooObj)::containedIn(ooRef(ooContObj)&) const;`
Invoking containedIn on a container object reference or handle returns an object reference or handle to a database:

- ooHandle(ooDBObj)
  ooRefHandle(ooContObj)::containedIn() const;
- ooHandle(ooDBObj)&
  ooRefHandle(ooContObj)::containedIn(
    ooHandle(ooDBObj)&) const;
- ooRef(ooDBObj)&
  ooRefHandle(ooContObj)::containedIn(
    ooRef(ooDBObj)&) const;

Invoking containedIn on a database object reference or handle returns an object reference or handle to a federated database:

- ooHandle(ooFDObj) ooRefHandle(ooDBObj)::containedIn() const;
- ooHandle(ooFDObj)& ooRefHandle(ooDBObj)::containedIn(
    ooHandle(ooFDObj)&) const;
- ooRef(ooFDObj)& ooRefHandle(ooDBObj)::containedIn(
    ooRef(ooFDObj)&) const;

Example

This example obtains the handle of the database that contains the container referenced by handle contH, using both the reference and value forms of the containedIn member function.

ooHandle(ooDBObj) dbh1, dbh2;
ooItr(ooContObj) contH;
...
// contH is set to a valid container

dbh1 = contH.containedIn();    // Handle is auto allocated
contH.containedIn(dbh2);      // Specify handle to use
...
Using Predicate Query

The predicate query feature uses predicate forms of iterator member functions. If your code initializes an iterator using the `scan` or `nameOfLink` member functions, a predicate query allows you to tighten the selection criteria based on a user-defined condition called a conditional predicate. Where appropriate, the predicate query feature uses indexes you created using the indexing facility (see “Using Indexes” on page 9-21).

A predicate query uses a predicate language that allows you to specify predicates using C++ and C syntax for standard operators, user-defined operators, constant literals, regular expressions, and object data member access. For a description of this language, see the “Predicate Query Language” appendix.

You can embed queries directly in your applications to initialize an iterator to select all objects of a given type that match your query within the scope of a container, a database, or a federated database. You can compile the query expression directly into your program or create a query during execution using standard C++ and C character string operations.

Using `scan` with a Conditional Predicate

The following syntax for the `scan` member function initializes the iterator with only those persistent objects that are both in the scanned storage object and satisfy the specified condition:

- `ooStatus ooItr(className)::scan(
  ooHandle(scopeClass)& scopeObject,
  const char* predicate);`
- `ooStatus ooItr(className)::scan(
  ooHandle(scopeClass)& scopeObject,
  const ooMode openMode,
  const ooAccessMode access,
  const char* predicate);`

where

- `scopeObject` Handle of the object to scan
- `scopeClass` Class of the object to scan. `scopeClass` may be either `ooContObj`, `ooDBObj`, `ooFDObj`, or a subclass of `ooContObj`.

9-18 Using Objectivity/C++
openMode
Optional parameter specifying the mode in which each object is opened when the next member function is used. Its value may be any of the constants oocRead, oocUpdate, or oocNoOpen. If openMode is not specified, the default is oocNoOpen.

access
Specifies the accessibility of the class members in the predicate. Its value must be one of the constants oocPublic or oocAll.
If you do not specify access, or specify oocPublic, the default value, you can access only public class members in the predicate, thus preserving encapsulation. If you use oocAll, you can access any class member in the predicate. Using oocAll decreases the encapsulation of your system. To preserve maximum encapsulation, use it only within member functions of the class you are querying.

predicate
Condition that each retrieved object must satisfy. The iterator returns only the objects that match the predicate.

Example
The following code initializes an iterator layerI to return only objects of type Layer whose data member x has a value greater than 10.

```
 ooHandle(ooContObj) contH;
 ooItr(Layer) layerI;
 char pred[32];
 (void) sprintf(pred,"x>%d",10);
 layerI.scan(contH, oocRead, oocPublic, pred);
```
Using *nameOfLink* with a Conditional Predicate

The following syntax for the *nameOfLink* member function initializes an iterator to traverse the objects associated by the one-to-many or many-to-many link that satisfies the specified condition. If no associations exist that match the specified predicate, the iterator is returned empty (that is, the first *next* returns a null handle).

```c++
❑ ooStatus className::nameOfLink(
    ooItr(className)& itr,
    const char *predicate);
❑ ooStatus nameOfLink(
    ooItr(className)& iterator,
    const ooMode openMode,
    const ooAccessMode access,
    char *predicate);
```

where

- *nameOfLink* Name of the association link
- *className* Class of the linked objects
- *iterator* Optional parameter that specifies the mode in which each object is opened when the *next* member function is used. Its value may be any of the constants *oocRead*, *oocUpdate*, or *oocNoOpen*. If *openMode* is not specified, the default is *oocNoOpen*.
- *openMode* Specifies the accessibility of the class members in the predicate. Its value must be one of the constants *oocPublic* and *oocAll*. *oocPublic* is the default value. If you do not specify *access*, or if you use a value of *oocPublic*, you can access only public class members in the predicate, thus preserving encapsulation. If you specify *oocAll*, you can access any class member in the predicate. Using *oocAll* decreases the encapsulation of your system, so use it only within member functions of the class you are querying to preserve maximum encapsulation.
- *access* Condition that each retrieved object must satisfy. The iterator returns only the objects that match the *predicate*. The

9-20 Using Objectivity/C++
Example

The following code demonstrates how to restrict the range of an iterator traversing a one-to-many association. The conditional predicate restricts the iterator to return only the associated objects of class `Layer` whose data member `number` has a value greater than 5:

```cpp
ooHandle<Cell> cellH;
ooItr(Layer) layerI;

// Assume cellH is attached to a cell.
cellH->layers(layerI, "number > 5");
```

Using Indexes

You can enable the predicate query feature to use indexes to increase the speed of queries based on the specified predicate. For more information on indexes, see "Using the Indexes You Create" on page 14-14.

To enable predicate query to use indexes, call the `ooUseIndex` function with `useIndex` set to `oocTrue` (the default value). The syntax for `ooUseIndex` is as follows:

```cpp
extern void ooUseIndex(
    const ooBoolean useIndex = oocTrue);
```

*useIndex* Set this to `oocTrue` to use indexes during scan or `oocFalse` to disable the use of indexes.

Indexes are used in the optimization of queries under the following circumstances:

* The predicate must be an AND of one or more terms of the form:
  * `(fieldName operation constant)`

where

* `fieldName` Name of the data member
* `operation` Relational Operator, including `=`, `>`, `<`, `>=`, `<=`, `~`
Any operation that can be used in the specification of ooLookup key (with the exception of =~. See the table below) can be used in an index. You can specify AND\'ed terms that do not conform to the above form, but they will not be used in locating an appropriate index. The use of OR at the top level will result in indexes not being used.

constant

- Integer, numeric, or string constant
- Only indexes that exist at the same hierarchical level (container, database, or federated database) as the scan you specify will be used.
- The semantics you use for multiple-key index lookup is the same as those for a primitive index scan lookup (see "Initializing an Iterator to Use the Index and Lookup Key" on page 14-23). However, the predicate scan interface correctly filters returned objects, based on the content of the original predicate. Currently, the index facility will not always filter based on all keys that are specified in the lookup.
- Regular expression case-sensitive equality operations are optimized when the right-hand side regular expression begins with non-magic characters. For example:
  Me.er (for matching Meier and Meyer), Lot001[a-d].

The following represents examples of regular expressions that will not be optimized:

.*xyz, [a-z]Hello.
The following are examples of indexes and expressions that will/will not be optimized:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Index On</th>
<th>Use Index?</th>
<th>Why/Why Not</th>
</tr>
</thead>
<tbody>
<tr>
<td>x&gt;1</td>
<td>x</td>
<td>Yes</td>
<td>x&gt;1 is an indexable relation</td>
</tr>
<tr>
<td>y&gt;1</td>
<td>x</td>
<td>No</td>
<td>No index on y</td>
</tr>
<tr>
<td>x != 0</td>
<td>x</td>
<td>No</td>
<td>Not equal is not an index operation</td>
</tr>
<tr>
<td>(x&gt;1) &amp;&amp; (x&lt;4)</td>
<td>x</td>
<td>Yes</td>
<td>AND of two indexable relations</td>
</tr>
<tr>
<td>(x&gt;1) &amp;&amp; (x!=5)</td>
<td>x</td>
<td>Yes</td>
<td>x&gt;1 is an indexable relation</td>
</tr>
<tr>
<td>(x&gt;1)</td>
<td></td>
<td>(x&lt;4)</td>
<td>x</td>
</tr>
<tr>
<td>(y&gt;1)</td>
<td>x, y</td>
<td>No</td>
<td>First field (x) is not in the predicate</td>
</tr>
<tr>
<td>(y&gt;1) &amp;&amp; (x&gt;34)</td>
<td>x, y</td>
<td>Yes</td>
<td>First field (x) is in the predicate</td>
</tr>
<tr>
<td>(x &gt; y)</td>
<td>x</td>
<td>No</td>
<td>y not a constant</td>
</tr>
<tr>
<td>name =~ &quot;Me.er&quot;</td>
<td>name</td>
<td>Yes</td>
<td>Starts with non-magic 'Me&quot;</td>
</tr>
<tr>
<td>name =~ &quot;.*ing&quot;</td>
<td>name</td>
<td>No</td>
<td>Starts with magic &quot;.&quot;</td>
</tr>
</tbody>
</table>

Disabling Indexes

There may be times when you will not want predicate query to use indexes. Two possible scenarios for disabling indexes are when:

- A query is going to scan the entire range of the type and sorting will not be necessary. In such a case, indexes will not speed up your query.
- Data to be scanned has not been committed to indexes since the data was created or last modified.

To prevent predicate query from using indexes, use the ooUseIndex function with useIndex argument set to oocFalse.
Using ooQuery

You can use the class ooQuery to iterate over sequences of objects that are not directly supported by the predicate query feature. For example, you can use ooQuery to iterate over sequences other than those defined by associations and containment, such as a VArray of object references, ooVArray(ooRef(userClass)).

ooQuery provides the following member functions.

- ooStatus setup(char *predicate, ooTypeNumber typeN, oqrUserOperatorSet * = 0);

  Setup sets the predicate string, type number, and user operator set for the query.

- ooBoolean evaluate(ooHandle(ooObj) &objH);

  Evaluates the query for the object referenced by objH.

Example

The following code uses ooQuery to iterate over objects of class myClass and select those objects whose y and z fields match the predicate y > 4 && z = 4.

```
ooItr(myClass) myClassI;
ooQuery *x = new ooQuery;
x->setup("y > 4 && z = 4", ooTypeN(myClass));
myClassI.scan(dbH);
while (myClassI.next() && x->evaluate(myClassI)) {
    // operate on myClassI
}
```

The above code performs the same operations as the following predicate query code:

```
ooItr(myClass) myClassI;
myClassI.scan(dbH, "y > 4 && z == 4");
```
while (myClassI.next()) {
    // operate on myClassI
}

User-Defined Operators

In addition to standard operators, you can define your own relational operators for use in conditional predicates. The operators you define have the same precedence as other operators.

You can define a set of operators by creating an instance of `ooOperatorSet`. For your convenience, a global instance called `ooUserDefinedOperators` is already created for you. You can use this instance, or create operator sets for different context within your application. For an example of how to use `ooUserDefinedOperators`, see “Using User-Defined Operators” on page 9-28.

Initially, `ooUserDefinedOperators` is empty. To add operator to this set, you need to create an operator function and register it along with the token that represents it. If you create a new instance of `ooOperatorSet`, you must set `ooUserDefinedOperators` to the new object so its operators are available.

Creating an Operator

To create a user-defined operator, define a function using the following syntax:

- `ooBoolean userDefinedFunctionName(
  const void* lPtr,
  const void* rPtr,
  ooDataType lAType,
  ooDataType rAType);

where

- `userDefinedFunctionName` Function name
- `lPtr` Pointer to the left operand of the operator
- `rPtr` Pointer to the right operand of the operator
- `lAType, rAType` Data types for the left and right operands, respectively. Every operator function is expected to handle arguments of whatever type the query yields.
A return value of `oocTrue` signifies that the expression satisfied the condition of the operator. A value of `oocFalse` signifies that the expression did not satisfy the condition.

The enumerated type, `ooDataType`, used by the operator is defined as follows:

```c
enum ooDataType {
    oocInt32T,
    oocUint32T,
    oocFloat64T,
    oocCharPtrT,
    oocBooleanT,
    oocInvalidTypeT
};
```

Table 9-2 shows the mapping of `ooDataType` to base Objectivity/DB types.

<table>
<thead>
<tr>
<th><code>ooDataType</code> Type</th>
<th>Objectivity/DB Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>oocInt32T</td>
<td>int32</td>
</tr>
<tr>
<td>oocUint32T</td>
<td>uint32</td>
</tr>
<tr>
<td>oocFloat64T</td>
<td>float64</td>
</tr>
<tr>
<td>oocCharPtrT</td>
<td>char*</td>
</tr>
</tbody>
</table>

Some types are not included in the enumerated type `ooDataType`. Instead, these types are passed as corresponding larger types, as shown in Table 9-3.

<table>
<thead>
<tr>
<th>Objectivity/DB Type</th>
<th>Corresponding Expanded Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8 and int16</td>
<td>int32</td>
</tr>
<tr>
<td>uint16</td>
<td>uint32</td>
</tr>
<tr>
<td>float32</td>
<td>float64</td>
</tr>
<tr>
<td>all forms of character strings (ooString(N), ooVString, ooVArray(char) and fixed character arrays)</td>
<td>char *</td>
</tr>
</tbody>
</table>
Registering an Operator

After creating a user-defined operator, you need to register it using the member function `ooOperatorSet::registerOperator`.

- `ooStatus ooOperatorSet::registerOperator(const char * name, ooQueryOperatorPtr funcPtr);`

  where
  
  `name` Operator name. This name should not begin or end with the following symbols or symbol combinations: `)(&&||!',,.`

  If the name is the same as an existing operator, the new operator will override the standard behavior.

- `funcPtr` Pointer to the function that defines the operator’s behavior

Clearing User-Defined Operators

To clear all user-defined operators for a given operator set, use the `ooOperatorSet::clear` function.

- `ooOperatorSet::clear();`

Creating a User-Defined Operator Set

To create a user-defined operator set, use the following constructor:

- `ooOperatorSet::ooOperatorSet();`
Using User-Defined Operators

The following example shows how to create, register, and use user-defined operators. Note that you must use whitespace to separate user-defined operators from their operands.

Example

```c
ooBoolean sameLen(const void* a, const void* b,
   ooDataType aT, ooDataType bT)
{
    if (aT != oocCharPtrT || bT != oocCharPtrT) {
        return oocFalse;
    }
    if (!a || !b) {
        return oocFalse;
    }
    if (strlen((char *) a) == strlen((char *) b)) {
        return oocTrue;
    }
    return oocFalse;
}
...
ooUserDefinedOperators->registerOperator("@@", sameLen);
ooItr(myClass) itrI;
char *str;
...
char pred[64];
(void) sprintf(pred, "nameField =~ \"He.*\" AND nameField
   @@ %s", str);
itrI.scan(fdb#, pred);
```
while (itrI.next()) {
    // process all objects that have a string in nameField
    // that begins with He and has the same length as str
}
Using Variable-Size Arrays

This chapter discusses persistent and transient arrays known as variable-size arrays (VArrays).

VArrays are similar to standard C++ arrays, except that they can change in size at runtime. Like standard C++ arrays, they are zero-based, with indices ranging from 0 to \( n-1 \), where \( n \) is the number of elements in the array. The contents of a VArray are also guaranteed to be allocated contiguously in virtual memory. Elements of VArrays can be of most types, not only class types.

Objectivity/DB supports VArrays through two parameterized non-persistent classes, ooVArray and ooTVArray. VArrays of class ooVArray can be either persistent or transient. VArrays of class ooTVArray are always transient.

Using VArrays of Class ooVArray

This section discusses transient and persistent VArrays of class ooVArray. For information about VArrays of class ooTVArray, see “Using VArrays of Class ooTVArray” on page 10-11.

Using a VArray within a persistent object will make the VArray itself persistent.

Declaring a VArray

Before you can use a VArray of class \( \text{className} \), you must first use the C++ declare macro to declare the parameterized class needed. The declare macro has the following syntax for VArrays of class ooVArray:

```c++
declare(ooVArray, className);
```

where

\( className \) Element class of the VArray
Transient variable-size arrays of handles are not supported for VArrays of class ooVArray. For example, the following code does not work:

typedef ooHandle(Person) PersonH;
declare(ooVArray, PersonH);
ooVArray(PersonH) sales(10);
sales.resize(20)

Use a VArray of class ooTVArray (see “Using VArrays of Class ooTVArray” on page 10-11) for transient variable size arrays of handles:

typedef ooHandle(Person) PersonH;
declare(ooTVArray, PersonH);
ooTVArray(PersonH) sales(10);
sales.resize(20)

The DDL processor automatically declares and implements ooVArray(char) and ooVArray(int32), so you should not declare or implement these types yourself. You should also not declare or implement ooVArray(int8) since this type is expanded into ooVArray(char) by the DDL processor.

Implementing VArrays

Before you can access a VArray of a particular class, you must use the C++ implement macro to generate code needed for the parameterized class in one and only one of your source files. Use this macro outside of member function and function definitions. The syntax of the implement macro is as follows:

- implement(ooVArray, className);

where

className Element class of the VArray

You must use a corresponding implement for every declare for a VArray. If you do not, your application will generate linking errors due to missing member functions of the VArray.

The implement macro must be preceded by a corresponding declare macro.
Creating a VArray

To create a VArray of class `ooVArray`, use one of the following constructors:

- `ooVArray(className) :: ooVArray(className)();`

  where

  `className` Element class of the VArray

  This constructor takes no argument and no VArray is created. The size of the array is zero, and no storage is allocated until either the `resize` or `extend` member function is used (see “Resizing VArrays” on page 10-5 and “Extending VArrays” on page 10-7).

- `ooVArray(className) :: ooVArray(className) ( uint32 initialSize );`

  where

  `className` Element class of the VArray

  `initialSize` Number of elements to initially allocate. The maximum value of this argument is limited by amount of available swap space.

  This constructor creates a VArray of `initialSize` elements of class `className`. If `initialSize` is specified as zero, no storage is allocated until either the `resize` or `extend` member function is used.

- `ooVArray(className) :: ooVArray(className) ( ooVArray(className) & arrayName );`

  where

  `className` Element class of the VArray

  `arrayName` Name of the VArray

  This constructor creates a VArray whose size is equal to the size of VArray `arrayName`. An element-by-element copy is performed from the VArray in the argument to the newly created VArray.

Objectivity/DB does not support VArrays of VArrays.

You cannot use virtual memory pointers that are pointing to other elements within elements of a VArray of class `ooVArray`. You should instead use a VArray of class `ooTVArray`. For more information, see “Using VArrays of Class `ooTVArray`” on page 10-11.
If you define a constructor for class `className`, you must also define a constructor that can be called with no arguments.

**VArray Element Operators and Member Functions**

The following operators and member functions for VArray elements are defined for the `ooVArray` class:

- `className & ooVArray(className) :: operator [] (uint32 index) const;`

  where

  - `className` Element class of the VArray
  - `index` Index of the VArray element to access

  The subscript operator returns the `index`th element of the array. This operator also performs an array bounds check to ensure integrity.

  If the object containing the VArray is opened for update, you do not need to indicate that you want to update the VArray.

  If a persistent object containing a VArray is opened for read access, then using `operator[]` to update a VArray element may not be effective.

  The system needs to know if the VArray has been updated or not. There are two ways to do this:

  - Use the `set` member function to modify a particular element of the VArray
  - Call the `update` member function

  Since VArrays require an extra level of indirection, you cannot access the first element of a VArray by dereferencing it. For example, unlike standard C++,

  ```
  *arrayName
  ```

  will not return the first element of VArray `arrayName`.

- `ooStatus ooVArray(className) :: set (uint32 index, className & newValue);`

  where

  - `index` Index of the VArray element to access
  - `className` Element class of the VArray
  - `newValue` New value to be assigned to the element, passed by reference
This member function sets the indexth element of the array to the value newValue. If the VArray is defined within a persistent class, the part of the VArray affected by the operation will be marked for update. This member function also checks array bounds to ensure integrity.

\[
\text{className} \& \text{ooVArray(className)::elem(uint32 index)};
\]

where

\begin{itemize}
  \item \text{className} \quad \text{Element class of the VArray}
  \item \text{index} \quad \text{Index of the VArray element to access}
\end{itemize}

This member function behaves like the subscript operator except that it does not check array bounds.

\[
\text{className arrayName* ooVArray(className)::head();}
\]

This member function returns the pointer to the first element of the VArray if the VArray exists. Otherwise it returns a null value.

### Updating VArrays

Use the following member function when updating a VArray:

\[
\text{ooStatus ooVArray(className)::update();}
\]

This member function has no effect if the VArray is not a member of a persistent object. If the VArray is within a persistent object, this member function causes Objectivity/DB to mark the entire VArray for update. This allows you to use the elem member function or \[\] operator on the left side of an assignment rather than using the set member function.

If the persistent object that contains the VArray is open for read access, this member function will cause Objectivity/DB to promote the object's access mode to update mode.

### Resizing VArrays

Use the following member functions when resizing a VArray:

\[
\text{uint32 ooVArray(className)::size();}
\]

This member function returns the current number of elements in the VArray.

\[
\text{ooStatus ooVArray(className)::resize(uint32 newSize);}
\]

where

\begin{itemize}
  \item \text{newSize} \quad \text{Total number of VArray elements desired}
\end{itemize}
The array is resized to contain the number of elements specified by newSize. If the new array size is smaller than the existing array size, element destructors are invoked for those elements to be truncated and then the array is truncated to the new size. If the new array size is larger than the existing array size, storage will be allocated for the additional elements and element constructors are invoked for these additional elements. In either case, the array may be relocated in virtual memory as a result of this member function.

This member function causes Objectivity/DB to mark the entire VArray for update. If the persistent object that contains the VArray is open for read access, this member function will cause Objectivity/DB to promote the object’s access mode to update mode.

For example, suppose the original size of a VArray is $n$, and the VArray is resized to a bigger size $m$.

**ooVArray(className)::resize** performs the following steps:

1. Allocates another contiguous block of space for the VArray
2. Bit-wise copies the contents in the original VArray to the new VArray
3. Deallocates the original VArray
4. Invokes the element constructor that takes no arguments on the tail of the new VArray, starting at element with index $n$ and up to the last element; that is, the constructor is invoked $m-n$ times.

If the VArray is resized to a smaller size $p$, then **ooVArray(className)::resize** performs the following steps:

1. Invokes the element destructor on the tail of the original VArray, starting at the element index $p$ and up to the last element; that is, the destructor is invoked $n-p$ times.
2. Reallocates the original VArray to get a smaller size.
Extending VArrays

Use the following member functions when extending a VArray:

- `ooStatus ooVArray<className>::extend(className & newValue);`

  where
  
  - `className` Element class of the VArray
  - `newValue` Value of the new VArray element

  This member function allocates a new element at the end of the array and sets it to the value `newValue`. This member function simplifies the addition of new elements to a VArray. Because resizing a VArray can be an expensive operation, you may want to grow the array size substantially using the `resize` member function, rather than adding one element at a time using `extend`. At the same time you should be careful not to allocate space you do not need.

  If the persistent object that contains the VArray is open for read access, this member function will cause Objectivity/DB to promote the object's access mode to update mode.

  This member function marks the entire VArray for update.

Examples

This example fills a 21-element VArray using the `set` member function.

```c++
// DDL class declaration
declare(ooVArray, uint32);
class Counter : public ooObj{
    ...
    ooVArray(uint32) vector;
}
```

// Application code that fills a 20-element VArray, each element set to twice the value of its index.
```
implement(ooVArray, uint32); // implement macro
ooHandle(Counter) countH;
```

...
countH = New(contH) Counter();
countH->vector.resize(10); // Initial size = 10

register int i;
for (i=0; i < 10; i++)
    countH->vector.set(i, i*2); // Fill with values
countH->vector.resize(20); // Resize to 20.

for (i=10; i < 20; i++)
    countH->vector.set(i, i*2); // Add more values
countH->vector.extend(20*2); // Extend by one
uint32 size = countH->vector.size();

for (i=0;i < size;i++) // Print values
    printf("%d\n", countH->vector[i]);

This example is similar to the previous example, except that it uses the update member function and directly accesses the VArray elements using the [] operator.

// DDL class declaration
declare(ooVArray, uint32);
class Counter : public ooObj{
    ooVArray(uint32) vector;
}

// Application code that changes the value of a VArray, each
// element set to ten times the value of its index.
...
ooHandle(Counter) countH;
...
// countH is a handle to a Counter Object
countH -> vector.update();
uint32 size = countH -> vector.size();

10-8  Using Objectivity/C++
register int i;
for (i=0; i < size; i++) {
    countH->vector[i] = i*10; // Fill with values
    for (i=0;i < size;i++) // Print values
        printf("%d\n", countH->vector[i]);
}

---

VArray Assignment

The following operation supports the assignment of VArrays:

- `ooVArray(className) & ooVArray(className)::operator = (ooVArray(className) & arrayName);

When copying a VArray of class `ooVArray`, you can assign a persistent or a transient VArray to a persistent VArray. Similarly, you can assign a persistent or a transient VArray to a transient VArray. During a VArray copy operation, the system automatically adjusts the VArray size and performs an element-by-element copy. This operation marks the entire VArray for update.

---

Example

The following example assigns a transient VArray to a persistent VArray, a persistent VArray to a transient VArray, and a persistent VArray to another persistent VArray.

```c
// DDL Schema Code
typdef int32 Coord;
class Point {
    public:
        Coord x, y;
    Point() { x = y = 0; }
    Point(Coord newX, Coord newY) { x = newX; y = newY; }
};
```

Using VArrays of Class `ooVArray`


```cpp
declare(ooVArray,Point);
class Polygon : public ooObj {
   public:
      ooVArray(Point) vertices;
      Polygon() { }
      Polygon(uint32 size) : vertices(size) { }
      Polygon(ooVArray(Point)& varray) : vertices(varray) { }
};

// Application Code
// Create a transient Point VArray
Point p1(0, 0), p2(10, 10), p3(0,20);
ooVArray(Point) pts(3);
pts[0] = p1;pts[1] = p2;pts[2] = p3;

// Create a polygon
ooHandle(ooDBObj) dbH;
... 
ooHandle(Polygon) polyH;
polyH = new(dbH) Polygon;

// Assign a transient VArray to a persistent VArray
polyH->vertices = pts;

// Assign a persistent VArray to a transient VArray
pts = polyH->vertices;

// Create another polygon
ooHandle(Polygon) pH = new(dbH) Polygon(pts);

// Assign a persistent VArray to a persistent VArray
polyH->points = pH->points;
```
Using VArrays of Class \texttt{ooTVArray}

VArrays of class \texttt{ooVArray} can be either persistent or transient. \texttt{ooTVArray} is a parameterized class that supports another kind of transient VArray. This class does not support persistent VArrays.

The member functions defined in \texttt{ooTVArray} are the same as those defined in \texttt{ooVArray}. VArrays of class \texttt{ooTVArray} work in the same way as \texttt{ooVArray} except the semantics of \texttt{resize}, \texttt{extend}, and \texttt{VArray copy} are different. These differences are described in the following sections.

Within elements of a VArray of class \texttt{ooVArray}, you cannot use virtual memory pointers that are pointing to other elements. If there are virtual memory pointers in each element of a VArray of class \texttt{ooVArray}, and these pointers are pointing to other elements in the VArray, then growing the VArray invalidates the original virtual memory pointer.

The class \texttt{ooTVArray} solves this problem for VArrays if the following two conditions are satisfied:

\begin{itemize}
  \item The element constructor that takes no arguments will establish the pointer linkage
  \item The element destructor will unlink the pointers
\end{itemize}

\section*{Resizing \texttt{ooTVArray} VArrays}

VArrays of class \texttt{ooVArray} and \texttt{ooTVArray} behave differently when a VArray is resized to a smaller size or a larger size.

For example, assuming the original size of a transient VArray is $n$ and the VArray is resized to $m$:

\begin{verbatim}
\texttt{ooTVArray(className)::resize()}
\end{verbatim}

performs the following steps to shrink or grow the original VArray:

\begin{enumerate}
  \item Allocates another chunk of heap space for the transient VArray
  \item Invokes the element constructor on each element of the new transient VArray; that is, the constructor is invoked \textit{m} times
  \item Copies the contents in the original transient VArray element-by-element to the new VArray
  \item Invokes the element destructor on each element of the original transient VArray; that is, the destructor is invoked \textit{n} times
  \item Deallocates the original transient VArray
\end{enumerate}
The resize operation for ooTVArray runs slower than that of ooVArray. This could be significant if the original array size is large.

**Extending ooTVArray VArrays**

The extend member function calls resize first and then fills the contents of the last element. Since the semantics of resize for ooVArray and ooTVArray are different, extend is also different.

**Assigning ooTVArray VArrays**

The assignment operator operator= calls resize first, and then copies the VArray element-by-element. Since the semantics of resize are different for ooVArray and ooTVArray, operator= is also different.

**VArrays of Handles**

You can create transient VArrays from class ooVArray, and use them to store arrays of object references, but not handles. To create a transient VArray of handles, you must use the ooTVArray class. Use of ooTVArray allows the system to correctly trace each active handle after resizing.

__________

**Example**

typedef ooHandle(Net) NetH;
declare(ooTVArray, NetH);
ooTVArray(NetH) nets;
Locking and Concurrency

The locking and concurrency control mechanisms of Objectivity/DB are closely related to opening and closing objects. To fully understand locking and concurrency, you should also read the “Opening and Closing Objects” chapter.

Objectivity/DB is a concurrent multiprocess database management system. It manages concurrent access to persistent objects through the use of locks. A lock guarantees an application will have the requested access rights (read or update) to a persistent object for the remainder of the transaction in which the lock is obtained.

The Objectivity/DB lock manager automatically obtains locks needed by your application by coordinating with the lock server lock requests between processes. If necessary, your application can also explicitly request locks.

Where syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle.
Automatic Locking Rules

Objectivity/DB provides automatic locking of objects within a transaction. These locks are maintained by Objectivity/DB until the process commits or aborts the transaction, at which time all locks obtained during the transaction are released. Locks are automatically obtained by Objectivity/DB under the following circumstances:

Creating an object When an object is created, the process that creates the object is automatically given an update lock on the object. If the object is a basic object, the process that creates the object is also given an update lock on the container in which the object is created.

Opening an object If you implicitly or explicitly open an object, Objectivity/DB will attempt to obtain an appropriate lock based on the access mode in which the object is being opened. If the object is a basic object, the process that creates the object is also given an appropriate lock on the container in which the object resides.

Locking a container When a container is locked, all basic objects within a container are also locked. These implicit locks are removed when the lock on the container is removed at the end of the transaction.

Lock propagation Locks will be propagated to objects that are connected through associations with lock propagation behavior.

Member functions Objectivity/DB member functions that modify objects obtain update locks automatically on the objects they are modifying.

Table 11-1 and Table 11-2 show typical sequences of opening objects and the locks that are placed on the objects when they are opened.
Table 11-1 shows a sequence of opening objects assuming the federated database is opened in Update mode.

**Table 11-1: Automatic Locking for Federated Database in Update Mode**

<table>
<thead>
<tr>
<th>Step</th>
<th>Opening Sequence</th>
<th>Automatic Locks Obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Open federated database for update</td>
<td>Update lock on federated database</td>
</tr>
<tr>
<td>2</td>
<td>Open database for read</td>
<td>Read lock on database update lock on federated database</td>
</tr>
<tr>
<td>3</td>
<td>Open basic object for read</td>
<td>Read lock on container Read lock on database Update lock on federated database</td>
</tr>
<tr>
<td>4</td>
<td>Open basic object for update</td>
<td>Promotion of read lock to update lock on container Promotion of read lock to update lock on database Update lock on federated database</td>
</tr>
</tbody>
</table>

Table 11-2 shows a sequence of opening objects assuming the federated database is opened in read mode.

**Table 11-2: Automatic Locking for Federated Database in Read Mode**

<table>
<thead>
<tr>
<th>Step</th>
<th>Opening Sequence</th>
<th>Locks Obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Open federated database for read</td>
<td>Read lock on federated database</td>
</tr>
<tr>
<td>2</td>
<td>Open database for read</td>
<td>Read lock on database Read lock on federated database</td>
</tr>
<tr>
<td>3</td>
<td>Open basic object for read</td>
<td>Read lock on container Read lock on database Read lock on federated database</td>
</tr>
<tr>
<td>4</td>
<td>Open basic object for update</td>
<td>Error. Federated database is open for read, not update.</td>
</tr>
</tbody>
</table>
Concurrency Rules

If multiple processes attempt to open the same persistent object, the following access rules are enforced by the locks automatically granted by Objectivity/DB. These locks ensure the integrity of all transactions.

Database and Federated Database Concurrency

When you open federated databases and databases for read or update access using their open member functions, the default locking mode is to share the accesses with other processes. Therefore, any number of processes may have the same federated database or database open for either read or update access.

Container and Basic Object Concurrency

Opening a basic object locks the basic object and the container in which it resides. Similarly, opening a container locks the container and all basic objects it contains. Therefore, for the current version of Objectivity/DB:

- the smallest granularity for locking is a container
- any number of processes may open the same container or basic object in read mode if it has not already been opened in update mode by another process
- you cannot open a container or basic object in update mode as long as any other process has the same object already open
- if a process opens a container or basic object in update mode, no other process is allowed to open the same object

The last two rules do not apply if your application uses the Multiple Readers, One Writer (MROW) concurrency mechanism. See "Multiple Readers, One Writer (MROW)" on page 11-8 for more information.
Example

Suppose an application inspects basic objects within a container and, based on their current values, conditionally updates them. If the application opens the container for update access, no other process is allowed to read any objects in the container, even though there is a chance that the application will not update any of the objects.

Opening the container for read access increases concurrency by allowing other processes to open basic objects within the container for read access until the application actually opens a basic object for update access. However, if another process opens the container for read access, the first process cannot update the container until the second process commits or aborts its transaction unless MROW is in effect (see “Multiple Readers, One Writer (MROW)” on page 11-8).

Concurrency and Iterators

An iterator allows you to traverse a group of objects. Objectivity/DB performs a special locking procedure to improve concurrency of certain iterator-based transactions. For more information, see “Iteration and Concurrency” on page 9-12.

Handling Lock Conflicts

When you explicitly open an object using the open member function, Objectivity/DB automatically attempts to grant your process an appropriate lock based on the access mode you specified. However, due to locks granted to other processes, Objectivity/DB may not be able to immediately grant you a lock. You can configure Objectivity/DB to respond to this situation in one of two ways:

- immediately give up on the operation and return to the application with an error condition.
- wait for the resource that is locked (lock waiting).

You can specify the lock waiting behavior by calling the ooSetLockWait function any time after calling ooInit. If you do not call this function, Objectivity/DB returns immediately with an error condition if it cannot obtain a lock.
Activating Lock Waiting

You can queue requests for any lockable object in the lock server using the lock waiting feature. This allows a transaction to wait for a resource that is locked by another transaction. Once the object is unlocked, the waiting transaction is granted a lock on the object.

You can set the lock waiting option at any point in your application for the current transaction (if any) and all subsequent transactions in that process using the ooSetLockWait function.

```cpp
void ooSetLockWait(int32 waitOption = oocNoWait);
```

where `waitOption` can take one of the following values:

- `oocNoWait`: Do not wait for locks. This is the default value.
- `oocWait`: Wait forever for locks.
- `n`: Wait for `n` seconds, where `n` must be greater than or equal to 1, and less than or equal to 14400. If `n = 0`, then it is treated as `oocNoWait`. If `n` is less than 0 or greater than 14400, then it is treated as `oocWait`.

The wait option setting remains in effect for all transactions in the current process until it is changed by another call to `ooSetLockWait`.

You can override the current value of the lock waiting option for a single transaction by using the `waitOption` argument of the `ooTrans::start` member function. Conversely, if you call `ooSetLockWait` within a transaction, it overrides any value you gave for the lock waiting option when you invoked `ooTrans::start` to start the transaction.

Setting Lock Waiting Within a Transaction

You can set the lock waiting option within a transaction using the member function `ooTrans::start` and specifying the `waitOption` parameter. This overrides the wait option set by `ooSetLockWait` for the current transaction only. For more information about `ooTrans::start`, see “Starting a Transaction” on page 2-29.

If you do not specify a value for `waitOption` when you invoke the `ooTrans::start` member function, then the current value of `waitOption` remains in effect during the transaction. In this case, if your application has not used the function `ooSetLockWait` to set the lock waiting option, then the default value of the lock waiting option (`oocNoWait`) remains in effect before the transaction begins and after the transaction ends.

`waitOption` parameter of `ooTrans::start` can take one of the following values:
Handling Lock Conflicts

**oocTransNoWait**  
Wait option is not specified. Use the current lock waiting setting. This is the default value.

**oocNoWait**  
Do not wait for locks.

**oocWait**  
Wait forever for locks.

**n**  
Wait for \( n \) seconds, where \( n \) must be greater than or equal to 1, and less than or equal to 14400. If \( n = 0 \), then it is treated as oocNoWait. If \( n \) is less than 0 or greater than 14400, then it is treated as oocWait.

**Turning Off Lock Waiting**

If you have set lock waiting in your application by calling ooSetLockWait with a `waitOption` of oocWait or \( n \), you can turn it off for subsequent transactions by calling ooSetLockWait again, this time with a `waitOption` of oocNoWait.

**Deadlock Detection**

A deadlock is a circular condition where one or more transactions are queued, and each is waiting for a lock that will never become available. For example, a deadlock is created under the following conditions:

- Transaction 1 is waiting for transaction 2, and
- Transaction 2 is waiting for transaction 3, and
- Transaction 3 is waiting for transaction 1.

If a deadlock condition is detected, Objectivity/DB issues an error message. Whenever infinite lock waiting is requested (\( \text{waitOption} = \text{oocWait} \)), Objectivity/DB checks to see if queueing the request would result in a deadlock situation. If so, an error is returned to the requesting application. When finite lock waiting is requested (\( \text{waitOption} = n \) seconds), no deadlock checking is done. In this case, Objectivity/DB assumes that any deadlock condition that occurs will be broken when lock waiting times out.
Multiple Readers, One Writer (MROW)

Objectivity/DB provides a concurrency mechanism that allows a container in a transaction to have multiple readers and one writer. Therefore, one process can update a container while other processes are reading its original contents. This mechanism, known as MROW, operates at the transaction level and improves concurrency. For example, under MROW, if a process opens a container or basic object in update mode, multiple processes can still open the last-committed version of the same object for read access.

MROW is particularly useful for operations on containers that are infrequently updated, such as libraries. MROW makes it easier for update transactions to co-exist with read transactions.

You can activate or deactivate MROW for a transaction using the \texttt{ooTrans::start} member function. To refresh your view of a container within a transaction, you can use the member function \texttt{openMode} to get the open mode of the container, and the member function \texttt{isUpdated} to determine whether the container has already been updated and committed by another process.

\begin{itemize}
\item \textbf{Warning}
\end{itemize}

\begin{itemize}
\item Under MROW, concurrent transactions may not be serializable if any transaction accesses basic objects in more than one container. Two or more transactions are serializable if the result of executing them concurrently is the same as executing them in a non-overlapping order.
\item MROW is designed to allow multiple transactions to read a container that is being updated by another transaction. The updating transaction should not be an MROW transaction because of the possibility that referential integrity may be compromised.
\end{itemize}
Activating or Deactivating MROW for a Transaction

You can activate or deactivate MROW for a transaction using the member function `ooTrans::start`. By default, MROW is not active. To activate MROW, specify `oocMROW` as its first argument. To deactivate it, specify `oocNoMROW`. For more information about `ooTrans::start`, see “Starting a Transaction” on page 2-29.

Example

```
ooTrans trans;
trans.start(oocMROW);
```

Checking Whether an MROW Container Is Updated

When you open a container for reading in a transaction that uses MROW mode, you can use the following member function to check whether the container has already been updated and committed by another process. The checking is performed in the middle of a transaction.

❑ `ooBoolean ooRefHandle(ooContObj)::isUpdated() const;`

The `isUpdated` member function returns `oocTrue` only if the container has been updated by another process since being opened by the current process for an MROW read operation. It returns `oocFalse` in all other cases.
Refreshing the View of an MROW Container

If a container has been modified during an MROW transaction, you may want to refresh the view of the container for other objects. To do this, you can use the following member function:

```c++
ooStatus ooRefHandle(ooContObj)::refreshOpen(
    const ooMode openMode,
    ooBoolean *pIsUpdated) const;
```

This member function signals an error if any other handle is open in the container to be refreshed.

Example

```c++
ooTrans trans;
ooHandle(ooDBObj) dbH;
ooHandle(myContainer) contH;
...
trans.start(oocMROW);
...
contH.openMode(dbH, "myContainer", oocRead);
...
ooBoolean* beenUpdated = new ooBoolean;
contH.refreshOpen(oocRead, beenUpdated);
if (*beenUpdated == oocTrue) {
    // Container refreshed with new contents
}
```
Summary of MROW and Non-MROW Concurrency

The following information summarizes concurrency characteristics for non-MROW and MROW applications:

- An object cannot be updated by two applications simultaneously, regardless of their MROW status.
- An application cannot update an object that is being read by a non-MROW application.
- A non-MROW application cannot read an object that is being updated by another application.
- An object can be updated by an application and read by an MROW application simultaneously.

Table 11-3 summarizes the interaction of MROW and non-MROW transactions in the same container. In the table cells where an interaction is not allowed, the first transaction succeeds and subsequent transactions fail. Where an interaction is allowed, both transactions succeed.

<table>
<thead>
<tr>
<th></th>
<th>Subsequent transaction is MROW</th>
<th>Subsequent transaction is Non-MROW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>First transaction is</strong></td>
<td><strong>READ</strong></td>
<td><strong>UPDATE</strong></td>
</tr>
<tr>
<td><strong>MROW</strong></td>
<td>Allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td><strong>UPDATE</strong></td>
<td>Allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td><strong>First transaction is</strong></td>
<td><strong>READ</strong></td>
<td><strong>UPDATE</strong></td>
</tr>
<tr>
<td><strong>Non-MROW</strong></td>
<td>Allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td><strong>UPDATE</strong></td>
<td>Allowed</td>
<td>Not allowed</td>
</tr>
</tbody>
</table>
Warning

Updating transactions should not use MROW because the update may introduce federated database inconsistencies.

Explicit Locking

When a persistent object is opened, Objectivity/DB automatically determines what locks are necessary to ensure data integrity and grants them to the requesting process, based on the requested access mode. In general, the automatic locking provided by Objectivity/DB produces a level of federated database concurrency that is sufficient for most applications.

Automatic locking obtains access rights to resources as they are needed by the application. However, some applications may need to reserve access to all required resources in advance. Reasons for doing so might be to secure required access rights to the necessary persistent objects before beginning an operation, or to prevent other processes from modifying objects critical to the operation.

An application needing to reserve access to all necessary objects in advance can open them all before starting to work. However, this causes Objectivity/DB to bring them all into virtual memory, possibly long before they are required. Alternatively, Objectivity/DB allows you to explicitly lock persistent objects, letting you reserve access to persistent objects in advance without bringing them into virtual memory. Explicit locks are held until the end of the transaction in which they are obtained.
Example

Suppose an application needs to calculate a value based upon the state of many objects at a specific point in time. Although the application cannot check all of the necessary objects simultaneously, it can achieve the same effect by freezing the state of the objects and then checking them in sequence. Explicit locking effectively freezes the objects, because no other process can modify them as long as they are locked.

Two-Phase Locking

Objectivity/DB uses two-phase locking unless you do explicit locking. Two-phase locking guarantees that all transactions leave the federated database in a consistent state. If you use explicit locking, it is your responsibility to do so in a manner ensuring federated database consistency.

See the following for an in depth discussion of locking and federated database consistency:


Lock Modes

Locks are granted on a per-process level and, once granted, are held for the remainder of the transaction. There are two kinds of locks—read locks and update locks.

Read Lock

If a process is granted a read lock on a persistent object, any other process may also obtain a read lock on the same object (except as described in “Multiple Readers, One Writer (MROW)” on page 11-8). As long as an object is locked in read mode by at least one process, no other process is granted an update lock for it.
This mode is indicated in the programming interface by the constant \texttt{oocLockRead}, of type \texttt{ooLockMode}.

\textbf{Update Lock}

If a process is granted an update lock on a persistent object, no other process may obtain a lock (read or update) on the same object (except as described in “Multiple Readers, One Writer (MROW)” on page 11-8).

This mode is indicated in the programming interface by the constant \texttt{oocLockUpdate}, of type \texttt{ooLockMode}.

\textbf{Locking Granularity and Concurrency}

When performing explicit locking, you should consider how locking granularity affects concurrency. The following sections describe locking granularity from fine grain to large grain.

\textbf{Basic Object}

The finest level of granularity is the basic object. While providing maximum concurrency, locking at this level also requires the most work on the part of applications. Every time an application needs to read or modify an object, it must acquire the appropriate lock on the object.

Locking a basic object also locks the container in which it is located. Therefore, the smallest granularity for concurrency is effectively the container.

\textbf{Composite Object}

The next level of granularity is composite objects. When a composite object is locked, any object associated to it, along an association link with lock propagation enabled, is also locked.

Locking a composite object also locks all the containers in which its component basic objects reside.

\textbf{Container}

The next level of granularity is containers. When a container is locked, any basic objects it contains are automatically locked at the same time.
Database

An application may explicitly lock a database, which also locks everything it contains (that is, containers and the basic objects in them). While this greatly simplifies application programming, it may prevent multiple users from accessing a federated database simultaneously.

Federated Database

An application may also explicitly lock a federated database, which locks all the database, containers, and basic objects it contains.

Obtaining an Explicit Lock

You may explicitly lock a persistent object by invoking the following member function:

```cpp
ooStatus ooRefHandle(className)::lock(
    const ooLockMode lockMode) const;
```

where

- `className` is `ooObj`, `ooContObj`, `ooDBObj`, or `ooFDObj`, or a derived class.
- `lockMode` is the kind of lock to obtain on the object, either `oocLockRead` or `oocLockUpdate`.

The `lock` member function obtains the appropriate lock. If the object is a container (or database), all objects in the container (or database) are also locked.

For containers, the `lock` member function also propagates the lock to any objects associated through links that have lock propagation enabled. This call returns `oocSuccess` only if all of the eligible objects can be locked in the specified mode. See the “Data Definition Language” chapter of Using Objectivity/ C++ Data Definition Language for details on specifying lock propagation using the DDL. For additional information about composite objects, see “Composite Objects and Propagating Operations” on page 7-11.
Use the following member function to obtain the appropriate lock on the object, without propagation semantics:

\[\text{ooStatus} \text{ ooRefHandle}(\text{className})::\text{lockNoProp} (\text{const ooLockMode lockMode}) \text{ const;}\]

where

- **className** is `ooObj`, `ooContObj`, or a derived class.
- **lockMode** is the kind of lock to obtain on the object, either `oocLockRead` or `oocLockUpdate`.

See the “Object Reference and Handle Classes” appendix for detailed syntax and parameter information on all of the above member functions.

Locking a basic object also locks the container in which it is located and all other basic objects in the container.

---

**Example**

This example demonstrates how implicit and explicit locking works. Assume there are two processes, A and B, working on the same object in the federated database, and that Objectivity/DB is configured to immediately return an error if a lock cannot be obtained. The following diagram shows in sequence how each action affects the locks on the object, and the result of the action.

<table>
<thead>
<tr>
<th>Step</th>
<th>Request</th>
<th>Resulting Locks On Object</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A:lock(oocLockRead)</td>
<td>A:Read</td>
<td>okay</td>
</tr>
<tr>
<td>2</td>
<td>B:open(oocRead)</td>
<td>A:Read B:Read</td>
<td>okay</td>
</tr>
<tr>
<td>3</td>
<td>A:open(oocUpdate)</td>
<td>A:Update B:Read</td>
<td>error</td>
</tr>
<tr>
<td>4</td>
<td>B:close</td>
<td>A:Read B:Read</td>
<td>okay</td>
</tr>
<tr>
<td>5</td>
<td>B:commit</td>
<td>A:Read B:Read</td>
<td>okay</td>
</tr>
<tr>
<td>6</td>
<td>A:open(oocUpdate)</td>
<td>A:Update</td>
<td>okay</td>
</tr>
<tr>
<td>7</td>
<td>A:commit</td>
<td></td>
<td>okay</td>
</tr>
</tbody>
</table>
Disabling Locking

1. Process A explicitly requests and receives a read lock on the object.
2. Process B opens the object for read access, which implicitly requests a read lock. At this point both process A and B have read lock on the object.
3. Process A attempts to open the object for update, which implicitly requests an update lock. Since process B still holds a read lock on the object, the lock request for process A is denied, and the open request returns with a return code other than oocSuccess.
4. Process B closes the object. Note that the lock owned by B is not released.
5. Process B commits the transaction. This removes the read lock owned by process B.
6. Process A again attempts to open the object for update. This time there is no lock conflict, so process A is granted an update lock, and the open request returns with the return code oocSuccess.
7. Process A commits the transaction. This removes the update lock owned by process A, and there are no locks on the object.

Disabling Locking

The situation may arise where your application is guaranteed exclusive access to a federated database and requires maximum performance. Under these circumstances you may wish to disable the locking facilities of Objectivity/DB for your application.

Disabling locking has the following effects:

◆ Removes the runtime overhead associated with managing federated database locks, increasing application performance.
◆ Removes the concurrent access protection afforded by locking. If another process has access to the same data as your application, unpredictable results may occur, including corruption of your data.

To disable the Objectivity/DB locking facilities for your application, use the following system-defined function after the call to ooInit, but before starting a transaction:

```c
ooStatus ooNoLock();
```
Warning

Use of `ooNoLock` should not be taken lightly. If there is even a remote chance that another application will have access to the same data as your application, you should not disable locking.

For most applications, the benefits (data integrity, concurrent access, and so on) of locking far outweigh the slight performance gain obtained by disabling locking.

Example

```c++
ooTrans transaction;
...
ooInit();
ooNoLock();   // disables locking
trans.start();
...
```
Checking Objects Out and In

Objectivity/DB provides checkout and checkin facilities, which allow you to access persistent objects for an extended period of time (beyond process boundaries).

Using checkout

To check out an object, use the following member function

```cpp
ooStatus ooRefHandle(className)::checkout(
    const ooMode mode) const;
```

where

- `className` ooObj, ooContObj, or a derived class
- `mode` Access mode in which to check out the persistent object, either `oocUpdate` or `oocRead`

Checking out an object is like placing a persistent lock on the object. The lock is persistent since it is not removed until the user who issued the checkout issues the corresponding checkin of the object.

Any process with the same user ID as the one that issued the checkout has free access to the object and may obtain either read or update access to it.

Checking out a basic object will also check out the container in which it resides, and therefore all other basic objects in that container.

---

Example

```cpp
ooHandle(ooObj) objH;
...
// objH is set to reference an object
objH.checkout(oocUpdate);
```
Using checkin

To check in a previously checked out object, use the following member function:

❑ ooStatus ooRefHandle(className)::checkin() const;

where

className ooObj, ooContObj, or a derived class

This member function removes the persistent lock placed on the object by a previous call to the checkout member function. If the object was checked out in update mode, the changes to the object become visible to other user processes.

Example

ooHandle(ooObj) objH;
...
// objH is set to reference an object which is checked out.
objH.checkin();
Using the String Classes

The Objectivity/DB string classes support the persistent storage of C++ strings. Objects of this class behave like C++ strings and can be passed as arguments requiring the type `char *` during a function or member function call.

The string classes provide string class member functions and operators to facilitate the use of strings. These strings make more efficient use of space and improve performance.

There are two kinds of string classes supported by Objectivity/DB:

- **Class `ooVString`** uses a VArray of characters to implement the C++ string
- **Class `ooString(N)`** is a parameterized string class that contains a VArray of characters and a fixed character array. You can use this string class to optimize disk space usage and runtime performance. When you use this class, your application specifies the length of the fixed character array. If the C++ string can be stored in the fixed character array, it is stored there and the VArray of characters is not used. On the other hand, if the C++ string cannot be stored in the fixed character array, the VArray of characters is used.

Choosing a String Class

You can choose which string class to use in a persistent class. Your choice depends on your knowledge of the lengths of the strings in the class.

- If the lengths of the strings to be used in a string class are not known or if these lengths vary widely, you should use the `ooVString` class
- If most of the strings in a class are about the same length, it is more efficient to use the `ooString(N)` class

Although both `ooVString` and `ooString(N)` are described in the context of persistent storage, they also support transient objects. Objectivity/DB determines when to allocate persistent storage and when to allocate heap storage.
Class ooVString

The class ooVString uses a VArray of characters to implement a C++ string. In this interface, ooChar is the same type as char and ooVArray(ooChar) is an optimized version of ooVArray(char). The class ooVArray(ooChar) is predefined by Objectivity/DB.

An ooVString class can be used as a C++ string in a persistent class. The class declaration contains member functions and operators for constructing a string, for accessing the string, and for doing assignment, concatenation, and comparison.

ooVString Constructors

The ooVString class has three constructors. The default constructor does not create a string.

❑ ooVString::ooVString();

A second constructor creates a string containing a copy of characters pointed to by p. If p is null, then no string is constructed.

❑ ooVString::ooVString(const char * p);

The third constructor creates a string containing a copy of characters in s. If the length in s is 0, then no string is constructed.

❑ ooVString::ooVString(const ooVString & s);

ooVString Member Functions

The ooVString class defines three member functions you can use to determine and resize string lengths and to obtain a pointer to the first character in a string.

length

The length member function returns the length of the string. If the string has not been allocated or is empty, the value 0 is returned.

❑ uint32 ooVString::length() const;
resize

The resize member function allows you to resize the string to a new length of newLength. The actual number of bytes allocated is newLength + 1 since an extra byte is reserved for the terminating null character. If newLength is 0, then the effect is to deallocate the string.

❑ ooStatus ooVString::resize(const uint32 newLength);

head

The head member function returns a pointer to the first character of the string. If the string does not exist, then a null pointer is returned.

❑ char *ooVString::head() const;

⚠️ Warning

The head member function returns a null pointer if the string does not exist. Using this pointer may cause errors in your application.

ooVString Operators

The ooVString class defines six operators you can use to manipulate strings.

Assignment

The assignment operator, operator =, replaces the characters in the string with the characters pointed to by p. If p is null, then the effect is to delete the string.

❑ ooVString &ooVString::operator=(const char *p);

You can assign a string of ooVString to another string of this class by using this operator because there is a conversion operator from ooVString to const char *.
Example

```cpp
ooVString s1, s2 = "test assignment";
s1 = s2;
```

**Concatenate**

The concatenate operator, `operator +=`, allows you to concatenate strings. The characters pointed by `p` are added at the end of the string. If `p` is null, then no concatenation of characters is performed.

- `ooVString &ooVString::operator +=(const char *p);`

You can concatenate one `ooVString` to another string by using this operator because there is a conversion operator from `ooVString` to `const char *`.

Example

```cpp
ooVString s1 = "concatenate", s2 = "second string";
s1 += " ";
s1 += s2;
```

**Conversion**

The conversion operator allows you to convert an `ooVString` string to an object of type `const char *`. If the string does not exist, then the conversion results in the value null.

- `operator const char *() const;`
Index

The index operator, operator [], returns the value of the indexed character, index. Objectivity/DB checks whether the index is within the allowable range of the string.

- char &ooVString::operator [](const uint32 index);

Equal Comparison

The equal comparison operator, operator ==, compares two strings to determine if they are equal. This operator returns the non-zero value ooTrue if the string is equal to the string pointed by p or s. It returns ooFalse otherwise. If either string does not exist, ooFalse is returned. If p is null or the pointer to the first character of s is 0, then ooFalse is returned as well.

- ooBoolean ooVString::operator ==(const char * p) const;
- ooBoolean ooVString::operator ==(const ooVString & s) const;

Not Equal Comparison

The not equal operator, operator ! =, compares two strings to determine if they are not equal. This operator returns the non-zero value ooTrue if the string is not equal to the string pointed by p or s. It returns ooFalse otherwise. If either string does not exist, ooTrue is returned. If p is null or the pointer to the first character of s is 0, then ooTrue is returned.

- ooBoolean ooVString::operator !=(const char * p) const;
- ooBoolean ooVString::operator !=(const ooVString & s) const;
The following example shows how to declare a string of class `ooVString` in a persistent class and how to use the string in a sample code segment.

```cpp
// Define a persistent class Person in a DDL file, for example
// person.ddl
class Person : public ooObj {
  public:
    Person() { }
    Person(char * _name, char * _address, uint32 _id) :
      name(_name), address(_address), id(_id) { }
  ooVString name;
  ooVString address;
  uint32 id;
};

void search(const char *); // Program in a C++ source file
#include <stdio.h>
#include "person.h"

ooHandle(Person) personH;

// Create a Person basic object with name and idNumber
personH = new Person("John", 0, 231876549);

// Add the address for the person
personH->address = "124 Park Ave., Palo Alto, CA 95444";

// Concatenate the last name to the name already stored
personH->name += " Smith";
```
// Access the second character in the name
cchar c = personH->name[2];

// Get the length of the name
uint32 length = personH->name.length();

// Pass name as a C++ string. A VString behaves like char *
search(personH->name);

// Compare the name
if (personH->name == "Larry Johnson")
    printf("Larry Johnson is found\n");

// Create another person
ooHandle(Person) manH;
manH = new(personH) Person("Ken Smith", 0, 227549990);

// Ken Smith has the same address as John Smith
manH->address = personH->address;

// Print Ken Smith's address
// Sometimes an explicit cast of ooVString to char * is needed.
printf("Ken Smith: %s\n", (const char *) manH->address);

// Compare the name
if (personH->name != manH->name)
    printf("This person is not John Smith\n");

// Resize a string
char * abc = "abcdefghijklmnopqrstuvwxyz";
uint32 length = strlen(abc);
personH->name.resize(length);
memcpy(personH->name.head(), abc, length + 1);
Class ooString(N)

// but the same thing could be accomplished via
// personH->name = abc; check if the name is null or not
if (! personH->name)
    printf("Error: name is null
");

The parameterized class ooString(N) contains a VArray of characters and a fixed character array whose length is the integer N, where N > 0. If the size of a C++ string is not greater than N, the string is stored in the fixed array. Otherwise, it is stored in the VArray and the fixed array is not used.

Like the ooVString class, an ooString(N) class can be used as a C++ string in a persistent class. Operators and member functions exist for constructing a string, for getting the string element and the string length, and for doing assignment, concatenation, and comparison.

You should choose a value for N (the length of the fixed character array in the class) so that a high percentage (for example, 90%) of the strings in the class have a length less than N. It is preferable that N be an even number. When choosing N, you should take into account the terminating null needed by C++ strings.

The use of ooString(N) allows you to avoid the overhead of VArrays when operating on strings whose size you can predict, and still have the flexibility to use VArrays if an occasional large string occurs. For example, if you are defining a class that contains mostly strings of less than 8 bytes, you might want to use the ooString(8) class. This class provides maximum efficiency for most of your strings (avoiding VArray overhead when the VArray is not needed) and uses VArray for the occasional occurrence of strings of length greater than 7.

When you use the ooString(N) class, the fixed character array is allocated whether it is used or not. If ooString(N) is used and the number N is not properly chosen, then the fixed part of ooString(N) could be too big to be fully utilized or be too small to store the string in most cases. In either case, a lot of storage space is wasted. You should perform an analysis of usage patterns before selecting N.
To use the ooString(N) class, you must define a string whose length for the fixed part is \( N \), where \( N > 0 \). To do this, add the following declarations to your code:

- declare(ooString, N) // add to DDL or header file
- implement(ooString, N) // add to source code file

`declare(ooString, N)` should normally be in a schema (DDL) file or a header file, while `implement(ooString, N)` should be in exactly one C++ source file. For multi-platform portability, there should be no space after the comma since the C++ preprocessor behaves differently on different machines.

After you have declared the class ooString(N), you can use this class as a data member in any class.

---

**Example**

The following macro calls create a new class called ooString(8) with a length of 8 for the fixed character array. The first call creates the class declaration and the second creates the non-inline member functions for ooString(8).

```
declare(ooString, 8)
```
```
implement(ooString, 8)
```

---

**ooString(N) Constructors**

The ooString(N) class has four constructors. The default constructor creates an empty string.

- `ooString(N)::ooString(N)();`

The copy constructor creates a string containing a copy of characters in ooString s. If the length of s is 0, an empty string is constructed.

- `ooString(N)::ooString(N)(const ooString(N) &s);`

The third constructor creates a string containing a copy of characters pointed to by p. If p is a null pointer, an empty string is constructed.

- `ooString(N)::ooString(N)(const char * p);`
ooString(N) provides a fourth constructor that creates a string containing a copy of characters in ooVString s. If the length of s is 0, an empty string is constructed.

- ooString::ooString(N)(const ooVString & s);

### ooString(N) Member Functions

The ooString(N) class defines three member functions that you can use to determine and resize string lengths and to obtain pointer to the first character in a string.

#### length

The length member function returns the length of the string. If the string is empty, the value 0 is returned.

- uint32 ooString(N)::length() const;

#### resize

The resize member function resizes the string to a new length of newLength. The actual number of bytes allocated is newLength + 1 since an extra byte is reserved for the terminating null character, 0. If newLength is 0, the effect is to delete the string and then create an empty one. If newLength is less than N, then only the fixed portion of the string is used.

- ooStatus ooString(N)::resize(const uint32 newLength);

#### head

The head member function returns a pointer to the first character of the string.

- char *ooString(N)::head() const;
**Warning**

The `head` member function returns a null pointer if the string does not exist. Using this pointer may cause errors in your application.

### ooString(N) Operators

The `ooString(N)` class defines six operators you can use to manipulate strings.

#### Assignment

The assignment operator, `operator =`, replaces the characters in the string with the characters pointed to by `s`. If `s` is null, the effect is to delete the string and then create an empty one.

```cpp
ooString(N) &ooString(N)::operator =(const char *s);
```

You can assign a string of class `ooString(N)` or `ooVString` to another string of class `ooString(N)` by using this operator because there is a conversion operator from `ooString(N)` or `ooVString` to `const char *`.

#### Example

```cpp
ooString(16) s1, s2 = "test assignment";
s1 = s2;
```
**Concatenate**

The concatenate operator, `operator +=`, allows you to concatenate strings. The characters pointed by `s` are added at the end of the string. If `s` is null, then no concatenation of characters is performed.

- `ooString(N) &ooString(N)::operator +=(const char * s);`

You can concatenate a string of class `ooString(N)` or `ooVString` to another string of class `ooString(N)` by using this operator because there is a conversion operator from `ooString(N)` or `ooVString` to `const char *`.

**Example**

```cpp
 ooString(16) s1 = "concatenate", s2 = "second string";
 s1 += " ";
 s1 += s2;
```

**Conversion**

The conversion operator allows you to convert a string of class `ooString(N)` to a string of type `const char *`

- `operator const char *() const;`

The conversion operator allows you to convert a string of class `ooString(N)` to a string of type `ooVString`. The syntax of this operator is:

- `operator const ooVString() const;`

**Index**

The index operator, `operator []`, returns the lvalue of the indexed character, `index`. Objectivity/DB automatically checks whether the index is within the allowable range of the string.

- `char &ooString(N)::operator [](const uint32 index) const;`
Equal Operator

The comparison operator, \texttt{operator ==}, compares two strings for equality. It returns the non-zero value \texttt{oocTrue} if the string is equal to the string pointed to by \texttt{p} or \texttt{s}. It returns \texttt{oocFalse} otherwise. If \texttt{p} is null or the pointer to the first character of the string is 0, then \texttt{oocFalse} is returned.

- \texttt{ooBoolean ooString(N)::operator == (const char * p) const;}
- \texttt{ooBoolean ooString(N)::operator == (const ooString(N) & s) const;}
- \texttt{ooBoolean ooString(N)::operator == (const ooVString & s) const;}

Not Equal Operator

The not equal operator, \texttt{operator !==}, compares two strings and tests for the inequality of the strings. It returns the non-zero value \texttt{oocTrue} if the string is not equal to the string pointed to by \texttt{p} or \texttt{s}. It returns \texttt{oocFalse} otherwise. If \texttt{p} is null or the pointer to the first character of the string is 0, then \texttt{oocTrue} is returned.

- \texttt{ooBoolean ooString(N)::operator != (const char * p) const;}
- \texttt{ooBoolean ooString(N)::operator != (const ooString(N) & s) const;}
- \texttt{ooBoolean ooString(N)::operator != (const ooVString & s) const;}
Example

The following example shows how to declare a string of class `ooString(N)` in a persistent class and how to use the string in a sample code segment.

```cpp
// Declarations in a DDL file, person.ddl
declare(ooString,8)
declare(ooString,24)

class Person : public ooObj {
    public:
        Person() { }
        Person(char * _name, char * _address, uint32 _id) :
            name(_name), address(_address), idNumber(_id) { }
        ooString(8) name;
        ooString(24) address;
        uint32 idNumber;
};

// Program in a C++ source file
#include <stdio.h>
#include "person.h"

implement(ooString,8)
implement(ooString,24)
ooHandle(ooDBObj) dbH;
...
// Assume dbH is a valid handle to a database.
// Create a Person basic object with name and idNumber.
ooHandle(Person) personH;
personH = new(dbH) Person("John", 0, 231876549);

// Add the address for the person.
personH->address = "124 Park Ave., Palo Alto, CA 95444";
```
// Concatenate the last name.
personH->name += " Smith";

// Get the length of the name.
uint32 length = personH->name.length();

// Compare the name.
if (personH->name == "Larry Johnson")
    printf("Larry Johnson is found\n");

// Compare the name.
ooVString who("Larry Johnson");
if (personH->name != who)
    printf("This person is not Larry Johnson\n");

// Resize a string.
char * abc = "abcdefghijklmnopqrstuvwxyz";
uint32 length = strlen(abc);
personH->name.resize(length);
memcpy(personH->name.head(), abc, length + 1);

// but the same thing would be accomplished via
// person->name = abc; check if the name is null or not.
if (!personH->name)
    printf("Error: name is null\n");
Class `ooString(N)`
Using the Map Dictionary Classes

The Objectivity/DB dictionary classes provide a specialized dictionary feature that can decrease the time it takes to create object names and use these names to locate objects.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`.

The dictionary classes provide a persistent hash table, where each element in the table consists of an object name and object identifier (OID). The key into the table is a C++ string (the name of the object). There is no limit on the length of this string. The hash table is dynamic because the number of hashing bins (buckets) can grow when a certain threshold is reached. All parameters that control the hash table are programmable by your application during the construction of the table. These programmable parameters are:

- Initial number of bins
- Maximum average density of elements in a bin
- Growth factor

Thus, the hash table can be constructed to provide optimal performance according to different application runtime requirements.

Map Dictionary Classes

Objectivity/DB defines three classes that support the dictionary of names and OIDs. These classes are:

- `class ooMap : public ooObj { ... }
- `class ooMapElem : public ooObj { ... }
- `class ooMapItr : public ooItr(ooMapElem) { ... }"
The class \texttt{ooMap} defines the hashing bins, and the class \texttt{ooMapElem} defines each element in the hash table. Each element is basically a naming object that stores a C++ string, an OID, and a linkage to another element.

The class \texttt{ooMapElem} is transparent to applications in most cases. Objects of this class cannot be directly created by applications. The class \texttt{ooMap} provides the interface to create a new element, remove an element, and lookup elements by names.

The class \texttt{ooMapItr} defines an iterator for the hash table so that each element can be returned.

\section*{Including the Map Dictionary Header File}

To use the map dictionary classes, you must include the \texttt{ooMap.h} header file in your application.

\section*{Class \texttt{ooMap}}

The class \texttt{ooMap} defines a constructor and member functions that allow you to build a hash table, manipulate the table and its elements, and obtain information about the table.

\subsection*{ooMap Constructor}

The \texttt{ooMap} constructor constructs a new \texttt{ooMap} class.

\begin{verbatim}
    ooMap::ooMap(
            const uint32 nBins = oocMapInitHashBinSize,
            const uint32 maxAvgDensity = oocMapMaxAvgDensity,
            const uint32 percentGrowth = oocMapPercentGrow);
\end{verbatim}

where

- \texttt{nBins} Initial number of bins in the hash table. The default value is 11.
- \texttt{maxAvgDensity} Parameter to indicate the average number of elements per bin allowed before resizing of the table. For example, if the total number of elements in the table is greater than or equal to \texttt{nBins * maxAvgDensity}, the table is resized. The default value of this parameter is 5.

\section*{13-2 Using Objectivity/C++}
percentGrowth

Parameter (expressed as percent) used to resize the table. The default value is 100. The table growth factor is computed by dividing this parameter by 100 and then adding 1. The table growth factor is used to multiply the original bin size to get the new bin size. The system rounds the number of bins after rehashing to a prime number.
ooMap Member Functions

Checking for a String Name
The `isMember` member function checks if any element in the hash table contains the name `name`.

❑ `ooBoolean ooMap::isMember(const char * name);`

where

`name` Name contained in an element

Looking Up a String Name
The following member function looks up the string `name` in the hash table, and returns the associated object identifier (OID). If `name` is not found or if there is an error during the lookup operation, a null OID is returned.

❑ `ooRef(ooObj) ooMap::lookup(const char * name);`

where

`name` Name contained in an element

Another form of the `lookup` member function looks up the string `name` in the hash table and returns the associated object reference or handle. If `name` is not found or if there is an error during the lookup operation, a zero value is returned and the handle or object reference is set to 0. The third argument specifies the mode in which you want to open the object.

❑ `ooStatus ooMap::lookup(`
  const char * `name`,
  ooRef(ooObj) & `objR`,
  const ooMode `openMode`=ooRead);

❑ `ooStatus ooMap::lookup(`
  const char * `name`,
  ooHandle(ooObj) & `objH`,
  const ooMode `openMode`=ooRead);`
where

name Name contained in an element
objR Object reference of the element
objH Handle of the element
openMode Mode in which to open the object (unless oocNoOpen is specified)

Replacing an Element in the Hash Table

If the string name is found in the table, the following member function replaces the
OID associated with string name with the OID of the object referenced by objR. If
elemH or elemR is specified, it is initialized to reference the new element.

❑ ooStatus ooMap::replace(
    const char *name,
    const ooRef(ooObj) &objR,
    ooHandle(ooMapElem) &elemH = oocNullMapElemHandle);
❑ ooStatus ooMap::replace(
    const char *name,
    const ooRef(ooObj) &objR,
    ooRef(ooMapElem) &elemR);

where

name Name contained in an element
objR Object reference of the referenced element
elemH Handle that references the replaced element
elemR Object reference that references the replaced element

If the string name is not found, a new element is created that contains the string
name and the specified OID. If this operation is successful, a non-zero value is
returned. If an error occurs, a zero value is returned.

With the returned object reference or handle, you can set a unidirectional
association to this element or perform other operations if so desired.
Adding a New Element to the Hash Table

The following member function adds a new element to the hash table. The new element has the name \texttt{name} and the OID referenced by \texttt{objR}. If \texttt{elemH} or \texttt{elemR} is specified, it is initialized to reference the new element.

\begin{itemize}
  \item \texttt{ooStatus ooMap::add(const char \* name, const ooRef(ooObj) \& objR, ooHandle(ooMapElem) \& elemH = oocNullMapElemHandle);} \end{itemize}

\begin{itemize}
  \item \texttt{ooStatus ooMap::add(const char \* name, const ooRef(ooObj) \& objR, ooRef(ooMapElem) \& elemR);} \end{itemize}

\textbf{where}
\begin{itemize}
  \item \texttt{name} Name contained in an element
  \item \texttt{objR} Object reference of the referenced element
  \item \texttt{elemH} Handle that references the new element
  \item \texttt{elemR} Object reference that references the new element
\end{itemize}

If there is already an element with this name, an error is returned.

The default value of \texttt{elemH} or \texttt{elemR} is a null reference to a map element handle or object reference. If \texttt{elemH} or \texttt{elemR} is not the default value, a handle or object reference to the element just added is returned in it. With the returned object reference or handle or object reference, you can set a unidirectional association to this element or perform other operations if so desired.

Forced Add of an Element to the Hash Table

The following member function is similar to \texttt{add} except there is no checking on whether any element in the table already has the specified name. Consequently, this member function is more efficient than \texttt{add}. If \texttt{elemH} or \texttt{elemR} is specified, it is initialized to reference the new element.

\begin{itemize}
  \item \texttt{ooStatus ooMap::forceAdd(const char \* name, const ooRef(ooObj) \& objR, ooHandle(ooMapElem) \& elemH = oocNullMapElemHandle);} \end{itemize}

\begin{itemize}
  \item \texttt{ooStatus ooMap::forceAdd(const char \* name, const ooRef(ooObj) \& objR, ooRef(ooMapElem) \& elemR);} \end{itemize}

13-6 Using Objectivity/C++
Class ooMap

where

- name: Name contained in an element
- objR: Object reference of the referenced element
- elemH: Handle that references the new element
- elemR: Object reference that references the new element

If the operation is successful, a non-zero value is returned. Otherwise, a zero-value is returned.

If there is already an element with this name in the table, two elements will contain the same name after a successful `forceAdd` operation. The element that is found by `lookup` is undefined. You should only use this member function if you know there is no element with the same name.

The default value of `elemH` or `elemR` is a null reference to a map element handle or object reference. If `elemH` or `elemR` is not the default value, then a handle or object reference to the element just added is returned in it. With the returned handle or object reference, you can set a unidirectional association to this element or perform other operations if so desired.

Example

```c++
ooHandle(ooMap) mapH;
// A is a persistent class
ooHandle(A) aH, targetH;
...
// Assume mapH and targetH are valid handles
if (mapH->lookup("adder", aH)) {
    ...
}
else {
    // Add a new element
    mapH->forceAdd("adder", targetH);
}
```

Using the Map Dictionary Classes 13-7
Removing an Element from the Hash Table

The following member function deletes the element with the string name or referenced by objH from the hash table:

- `ooStatus ooMap::remove(const char * name);`
- `ooStatus ooMap::remove(const ooHandle(ooMapElem) & objH);`

If the operation is successful, an oocSuccess is returned. Otherwise, oocError is returned. remove will return oocSuccess if asked to remove a non-existing element.

If forceAdd was used to add more than one element with the same name, remove only deletes the first such element it finds and gives no indication that there are other elements with the same name.

Setting an Association to an Element

Given an element's handle, you can use the following member function to set a unidirectional or bidirectional association to the element:

- `ooStatus ooObj::add_mapElems(const ooHandle(ooMapElem) & objH);`

where

`objH` Handle that references the element

This member function creates an association to the element referenced by `objH` on the association link mapElems.

Removing an Association to an Element

You can remove a unidirectional or bidirectional association to an element using the following member function:

- `ooStatus ooObj::del_mapElems();`

This member function removes an association on the association link mapElems.

Removing a Number of Associations to an Element

You can remove a number of associations to an element using the following member function:
Class ooMap

- ooStatus ooObj::sub_mapElems(
  const ooHandle<ooMapElem> &objH,
  const uint32 number = 1);

  where
  objH Handle that references the element
  number Number of associations to remove

This member function removes the first number of associations encountered on the association link mapElems.

Clearing Hash Table Parameters

The following member function resets some statistical parameters to zero for the table. These parameters include the number of elements added, number of elements deleted, number of lookups, number of comparisons during lookup, and number of rehashes.

- void ooMap::clearParam();

Getting the Number of Elements

The following member function returns the total number of elements in the hash table:

- uint32 ooMap::nElement() const;

Getting the Number of Bins

The following member function returns the number of bins in the hash table:

- uint32 ooMap::nBin() const;

Getting the Bin Density

The following member function returns the allowable maximum average density of the table, where the average density equals the number of elements divided by the number of bins (nElements / nBin):

- uint32 ooMap::maxAvgDensity() const;
**Getting the Growth Factor**

The following member function returns the growth factor for the number of bins in the table:

\[
\text{uint32 ooMap::percentGrow() const;}
\]

**Printing Statistics**

The following member function prints out the runtime statistical information and parameters about the table in the file `outFile`. The default output file is `stdout`.

\[
\text{void ooMap::printStat(FILE* outFile = stdout) const;}
\]

where

`outFile` Name of the standard output file

The output from this member function is similar to:

Run statistics of ooMap #2-3-3-3 (Fri Aug 25 15:22:45 PDT 1992)

** Number of elements added => 0
** Number of elements removed => 0
** Number of rehashs => 0

** Current state:

<table>
<thead>
<tr>
<th></th>
<th>=&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of bins</td>
<td>23</td>
</tr>
<tr>
<td>Number of elements</td>
<td>100</td>
</tr>
<tr>
<td>Maximum average density</td>
<td>5</td>
</tr>
<tr>
<td>Percent Growth</td>
<td>100</td>
</tr>
<tr>
<td>Average length per bin</td>
<td>4.34783</td>
</tr>
<tr>
<td>Maximum length</td>
<td>7</td>
</tr>
<tr>
<td>Standard deviation of length</td>
<td>1.08783</td>
</tr>
<tr>
<td>Maintain referential integrity</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Advanced Techniques

The following functions are for advanced use of the dictionary class `ooMap`.

**Looking Up a String Name in the Hash Table**

Another form of the `lookup` member function introduced on page 13-4 looks up the string `name` in the table and returns an object reference or handle to the map element:

- `ooStatus lookup(
  const char * name,
  ooRef(ooMapElem) & elemR);`
- `ooStatus lookup(
  const char * name,
  ooHandle(ooMapElem) & elemH);`

Where

- `name` Name contained in an element
- `elemID` Object reference of the element
- `elemH` Handle of the element

If `name` is not found or if there is an error during the lookup operation, a zero value is returned.

**Resizing the Hash Table**

The following member function resizes the table with the number of bins equal to `binSize`. The default value of `binSize` is 11. If the operation is successful, a non-zero value is returned. Otherwise, a zero value is returned.

- `ooStatus ooMap::rehash(
  const uint32
  binSize = oocInitMapHashBinSize);`

Where

- `binSize` New number of bins
Normally you should not use this member function to explicitly resize the table. The resizing of the table to a bigger one is automatically done by Objectivity/DB. Only when the number of bins is very large and the total number of elements is relatively small is this member function useful for shrinking the table. This occurs when many elements are deleted after the table has grown to the peak size.

**Getting the Referential Integrity State**

The `refEnable` member function returns the flag that indicates whether or not the referential integrity between map elements and objects is to be maintained by Objectivity/DB. If the flag is `oocTrue`, Objectivity/DB will automatically maintain referential integrity; otherwise, no maintenance is done.

```cpp
ooBoolean ooMap::refEnable() const;
```

The benefit of turning off the referential integrity performed by Objectivity/DB is that there is less overhead in map operations such as adding or deleting elements. The default value of the flag is `oocTrue`, which causes Objectivity/DB to maintain the referential integrity.

**Setting the Referential Integrity Flag**

The `set_refEnable` member function sets the flag for enabling or disabling the referential integrity between map elements and objects. If the value of `refEnable` is `oocTrue`, Objectivity/DB will automatically maintain referential integrity when map elements are added or deleted; otherwise, no maintenance is done.

```cpp
ooStatus ooMap::set_refEnable(
    ooBoolean refEnable = oocTrue);
```

where

- `refEnable` Referential integrity flag

You can reset the flag only before any element is added into the table.
Warning

The member function `set_refEnable(oocFalse)` should be used only by advanced users since it turns off the support of referential integrity by the system. If you use the `set_refEnable` member function, you must make sure there are no dangling OIDs in map elements.

Getting the Hash Function Used in the Table

The `nameHashFunction` member function returns a pointer to the hash function used in the table.

- `static ooNameHashFuncPtr ooMap::nameHashFunction();`

The type `ooNameHashFuncPtr` is defined as:

- `typedef uint32 (*ooNameHashFuncPtr)(
  const char *,
  const uint32);`

The `nameHashFunction` member function is an attribute of the class `ooMap`, and is not defined on a per-instance basis.

Setting the Name Hash Function Used in the Table

If you wish to set your own name hash function rather than using the name hash function provided by Objectivity/DB, you can use the `set_nameHashFunction` function.

The hash function is not stored persistently. Since all access to the `ooMap` must use the same hash function, this member function must be invoked in every program that uses the `ooMap`.

- `static void ooMap::set_nameHashFunction(
  ooNameHashFuncPtr hashFunction);`

where

`hashFunction` Name of the name hash function
Example

```c
uint32 myNameHash(const char *name, const uint32 modulus)
{
    // This function should return a value between 0 and modulus-1
    ...}
    // Set the name hash function used in map
    // to be myNameHash

    ooMap::set_nameHashFunction(myNameHash);
```

Class `ooMapElem`

The class `ooMapElem` defines each element in the hash table. This class is transparent to applications in most cases.

`ooMapElem` Member Functions

The `name` member function returns the string `name` in the element.

- `const char * name();`

  where
  `name` Name contained in an element

The `oid` member function returns the OID in the element.

- `ooRef(ooObj) ooMapElem::oid() const;`

The `set_oid` member function sets the OID in the element.

- `ooStatus ooMapElem::set_oid(
    const ooRef(ooObj) &objR) const;`

  where
  `objR` Object reference of the object from which to set the data member OID
If you want to move an object whose OID is contained in an element, you must perform the necessary steps on the map dictionary. Remove the element before the object is moved, and then add a new element after the move is successfully completed.

**Class ooMapItr**

The class `ooMapItr` defines an iterator for the hash table so that elements can be returned one-by-one. You can initialize a map iterator by using the copy constructor or assignment operator of `ooMapItr`. Then, you can use the member function `next` to iterate through each element in the table.

**ooMapItr Constructors**

The syntax for the default constructor for `ooMapItr` is as follows:

- `ooMapItr::ooMapItr();`

The following `ooMapItr` constructor initializes the iterator for the object referenced by `mapH`:

- `ooMapItr::ooMapItr(ooHandle(ooMap) &mapH);`

where

- `mapH` Object handle

**ooMapItr Assignment Operator**

The assignment operator `operator=` initializes the iterator to reference the object referenced by `mapH` or `mapR`.

- `ooMapItr &ooMapItr::operator =(ooHandle(ooMap) &mapH);`
- `ooMapItr &ooMapItr::operator =(ooRef(ooMap) &mapR);`

where

- `mapH` Object handle
- `mapR` Object reference
ooMapItr next Member Function

The ooMapItr iteration member function next sets the iterator to reference the next element of the current iteration. This member function is inherited from class ooItr and retrieves map elements in a non-update mode. To access the target object of a map element, use the oid member function, which returns a type-independent OID of the object. You must then assign this OID to a handle or to an object reference that is cast to the appropriate type.

❑ ooBoolean ooMapItr::next();

You cannot add or delete elements during the map iteration. Only read mode is allowed during the iteration. You can modify the objects the map is indexing (your objects). You should not, however, delete objects if refEnable is set by set_refEnable.

Examples

Two examples of using the next member function are shown below. The first uses the name member function and the second uses the oid member function to retrieve the object in the iteration.

ooHandle(ooMap) mapH;
...

// mapH is a valid handle

ooMapItr mapI = mapH;
while(mapI.next()) {
    printf("name = %s\n", mapI->name());
}

This example illustrates how to use the class `ooMap` for fast name creation and lookup.

```c++
// Define a persistent class Person in a DDL file, person.ddl

class Person : public ooObj {
    public:
        Person() { }
        Person(char *_address, uint32 _id) :
                address(_address), id(_id) { }
        ooVString address;
        uint32 id;
    }

    // Program in a C++ source file

#include <stdio.h>
#include <ooMap.h>
#include "person.h"

    ooHandle(ooDBObj) dbH;
    ...

    // dbH is a valid handle
```
// Create a new ooMap, and use default values for the map
ooHandle(ooMap) mapH;
mapH = new(dbH) ooMap;

// Create a Person object with address and id
ooHandle(Person) personH, manH;
personH = new(mapH) Person("124 Park Ave., Palo Alto, CA 95444", 231876549);

// Add one element in the table
mapH->add("John Smith", personH);

// Create another person
personH = new(personH) Person("51st St., Menlo Park, CA 94025", 123456789);

// Add another element in the table
mapH->add("Peter Pan", personH);

// Lookup object by name in the table, and return the result in
// a handle
ooHandle(Person) manH;
if (!mapH->lookup("John Smith", manH))
    printf("Error in lookup of John Smith\n");
manH.print();

// Lookup object by name in the table, and return the result in
// an OID
ooId id;
if (!mapH->lookup("John Smith", id))
    printf("Error in lookup of John Smith\n");

// Print out the statistics of the table
mapH->printStat();
// Delete one element in the table
mapH->remove("John Smith");

// Create a third person
personH = new(personH) Person(
   "200 Park Ave., San Jose, CA 95132", 257633421);

// Replace one element in the table
mapH->replace("Peter Pan", personH);

// Print the number of elements and bins in the table
printf("Number of elements = %d\n", mapH->nElement());
printf("Number of bins     = %d\n", mapH->nBin());

// Clear statistical parameters
mapH->clearParam();

// Iterate through the map
ooMapItr mapI;
mapI = mapH;
while (mapI.next()) {
    // Print the contents of each element
    printf("name = %s, OID = ", mapI->name());
    mapI->oid().print();
}

---

Using the Map Dictionary Classes  13-19
Referential Integrity

The OID stored in an element of the hash table can become a dangling OID if its object is deleted and the element is not deleted accordingly. To maintain referential integrity, an association is defined in the system class ooObj. Whenever a new element is added in the table, an association to the element is added in the object whose OID is the same as the one in the element.

The association is defined as:

class ooObj {
    …
    ooHandle(ooMapElem) mapElems[] : prop(delete);
};

When an object is deleted, the association attribute prop(delete) causes the corresponding element in the hash table to be deleted.

The other benefit of having this association is that the name in an element of the hash table can be found very quickly for an object if it exists. You can traverse the association to get all the names in hash tables for a given object.

Example

This example shows how to get the names in the ooMap for an object.

    // A is a persistent class
    ooHandle(A) aH;
    …

    // aH is a valid handle

    // Get all the names in the hash tables for the object
    ooItr(ooMapElem) elemI;
    aH->mapElems(elemI);
    while (elemI.next()) {
        printf("Name is %s\n", elemI->name());
    };

13-20 Using Objectivity/C++
Using Indexes

Indexes provide faster access to basic objects or containers in a given scope. The scope of an index can be a container, a database, or a federated database. You can index objects that are distributed across containers in a database or across databases in a federated database, which allows you to index objects that you have grouped for optimal accessibility. This chapter provides indexing guidelines, and explains how to create and drop indexes for a class, and how to use the indexes you create to access objects. For index performance tuning information, see the “Monitoring and Tuning Performance” chapter.

Overview

An index sorts objects of a particular class according to the value in one or more members (fields) of the class. The objects are sorted in ascending order in a B-tree. Each field that is used for the index is called a key field. One or more key fields make up the key description. You first build a key description, then create indexes using that description in one or more scopes. You can then create many indexes for a single scope, using a different key description for each index.

For example, if you want fast access to employee objects by the value of the employeeNumber field, you can define employeeNumber as the key field, then define a key description to use that key field. The key description allows you to create an index on a scope of employee objects that sorts the objects in ascending order by the value of the employeeNumber field. An application can access a specific employee object that is indexed in this manner much faster than without indexing.
In this chapter, parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form:

```c++
const ooHandle(className) &
```

are interchangeable with:

```c++
const ooRef(className) &
```

## Building Applications That Use Indexes

To build applications that use indexes, you must include the `ooIndex.h` header file in your application.

The following guidelines will help you to most effectively use indexes. With indexes, you can:

- Have any number of indexes over the objects of a given class in a scope as long as you create each one from a different key description
- Have the same key field in as many key descriptions as you wish. The key field must be defined on the same class as the key description.
- Have as many key fields in a key description as you wish as long as they refer to members of the class or its base classes
- Create a unique key description. If you index a group of objects with a unique key, there can be no objects with the same key field values (that is, no duplicates) in the given scope.
- Refer to any field in a class (C++ `private`, `protected`, or `public`) as long as the data type of the field is one of the system-defined types:

```c++
uint8
uint16
int16
int32
uint32
float32
float64
char[]
ooString
ooVString
```
Creating an Index

ooVArray (ooChar) (treated as a null-terminated string)

- Refer to members of an embedded struct of any of the above types, but not to the struct itself
- Index on base class fields using standard C++ base class field notation
- Index all objects in a version genealogy. If two versions of an object have the same key field values, and if there is a unique index using those fields, indexing will raise a duplicate object error. Any other situation involving multiple versions in a single index is legal.
- Create different sorts by defining different key descriptions with the key fields in different orders. The key fields provide a sort order for the objects in the scope. That is, the index stores the key-to-object mapping in the order defined by the order of the key fields.
- Keep track of the key description in some way (for example, by scope name or Object Identifier (OID)) in order to be able to retrieve it later to create new indexes or drop old indexes.

Creating an Index

To create one or more indexes for a class, perform the following steps:

1. Create a key description object of the class ooKeyDesc, making sure to specify uniqueness if that is what you want, and place the key description in the scope you have chosen. See “Creating a Key Description” on page 14-3.
2. For each key field that you want to define, create an object of the class ooKeyField. See “Creating Key Fields” on page 14-5.
3. Add each ooKeyField object to the ooKeyDesc object in the order you want to sort the indexed objects. See “Adding a Key Field to the Key Description” on page 14-9.
4. Use the createIndex member function on the ooKeyDesc object to create an index from the key description in a scope. See “Creating an Index from the Key Description” on page 14-10.

Creating a Key Description

The class ooKeyDesc represents the key description object. The class constructor specifies the class you want to index and whether to make indexes unique on the key. A single key description object can be used to create several indexes, if you want the same index structure on objects in different scopes.
Creating an Index

- ooKeyDesc::ooKeyDesc(
  const ooTypeNumber typeN,
  const ooBoolean unique = oocFalse);

where

- **typeN**: Type number of the class you want to index. See “Obtaining a Class Type Number” on page 20-2. Note that all subclasses of the specified class will also be indexed. Container, database, and federated database wide indexes can be created on objects of type ooContObj and on all of its derived types.

- **unique**: Specifies whether to make an index unique. The default value is `oocFalse`. A key description for a non-unique index allows multiple objects to have the same key field value.

---

**Examples**

To create a key description for a non-unique index for the Computer class:

```cpp
ooHandle(ooContObj) descContH; // place to put description
ooHandle(ooKeyDesc) keyH;
...
// Assume descContH refers to the container object.
keyH = new(descContH) ooKeyDesc(ooTypeN(Computer));
```

This creates a new key description object in the description container.

To create a key description for a unique index, add `oocTrue` to the constructor arguments:

```cpp
keyH = new(descContH) ooKeyDesc(ooTypeN(Computer), oocTrue);
```

After adding the key fields, you can use the key description to create indexes on objects of class Computer and its derived classes in any scope in your federated database.
Placing a Key Description Object

Objectivity/DB stores key description objects in containers. Before creating the key description, you have to decide which container will contain the key description object.

The key description associates itself to the indexes in the scope. For optimal performance, use the following guidelines when deciding where to place the key description object.

- If all the containers are in a single database, you might place the key description in the default container for that database by using the database handle as a clustering directive.
- If the indexes span several databases, you can put the key description anywhere in the scope.
- If you intend to create only one index in a specific container, put the key description in that container by using the container handle as a clustering directive.

When you are going to create several indexes or you are going to create indexes at different points in your program, you will want to keep track of the key description for later retrieval. You can do this with a scope name or by storing the key description handle somewhere in your program.

Creating Key Fields

Once you have created a key description, you can create the key fields and add them to the key description. Key fields must be defined on the same class as their key description. The first key field added is used as the primary key for sorting the indexed objects, the second key field is the seconding key, and so on. The ooKeyField constructor specifies the class and member on which you want to index.

\[
\text{ooKeyField::ooKeyField(}
\begin{align*}
&\text{const ooTypeNumber typeN,}\\
&\text{const char *memberName,}\\
&\text{ooBoolean fixed = oocFalse,}\\
&\text{uint32 maxstrlen = 24);}
\end{align*}
\]

where

- **typeN** Type number of the class containing the member for the key field. This class must match the class specified when creating the key description.
- **memberName** Name of the member in the class

Using Indexes 14-5
**fixed**

Specifies whether the strings in the index are to be of fixed or variable length. The value can be either `oocTrue` or `oocFalse`; the default is `oocFalse`.

**maxstrlen**

Specifies what the maximum length of a string stored in the index can be. This length is always reserved for the string. If `fixed` is set to `oocTrue`, strings are limited to a length of `maxstrlen`. If `fixed` is set to `oocFalse`, strings of any length can be indexed, but the first `maxstrlen` bytes are stored directly in the index key.

You usually provide the key description handle as the clustering directive for each field, since the field objects will always associate to the description object.

**Derived Classes**

To create a key description for an index on a class that inherits from other classes, it is good practice to fully specify member names. If a member name is ambiguous, (the same member name is defined in both the base class and the derived class), fully specifying the member name ensures that the correct member is included in the index.

**Example**

This example shows when you can refer to a base class field by its local name in a key description, and when you have to fully specify the name using C++ notation.

Given a class `Point` that inherits first from class `A` as follows:

```
Class A {int a};
class Point: public A {
    int i;
};
```

If you wanted to create an `ooKeyDesc` on `A::a`, you could specify it as

```
 ooHandle(ooKeyField) keyFieldH;
 ooHandle(ooKeyDesc) keyDescH;
 ooHandle(ooContObj) contH;

 keyDescH = new(contH) ooKeyDesc(ooTypeN(Point), oocTrue);
```
keyFieldH = new(keyDescH) ooKeyField(ooTypeN(Point),"a");

Given a class Point that inherits first from class A and then from class B as follows:
Class A {int i};
Class B {int i};
Class Point: public A, public B {
    int x;
};

To access the field i in B, you need to specify an ooKeyDesc on B::i.

ooHandle(ooKeyField) keyFieldH;
ooHandle(ooKeyDesc) keyDescH;
ooHandle(ooContObj) contH;

keyDescH = new(contH) ooKeyDesc(ooTypeN(Point), oocTrue);
keyFieldH = new(keyDescH) ooKeyField(ooTypeN(Point), "B::i");

Fixed and Variable Length Strings

When you are creating a key on a string type, ooKeyField allows you to specify whether the keys are fixed or variable maximum length. Wherever appropriate, use char[] types, since keys on variable length strings with no maximum length may incur an extra page read per indexed object during process time. Indexes do not restrict the size of a string and will be able to support any string the federated database can support.

Fixed Maximum Length
Set the fixed variable of the ooKeyField constructor to oocTrue to specify that no strings greater than a length of maxstrlen will be indexed. The index key allocates a fixed length of maxstrlen bytes for the entire string.

Variable Maximum Length
Set the fixed variable of the ooKeyField constructor to oocFalse (the default value) to specify that any length string will be indexed. The index key allocates maxstrlen bytes for the first part of the string. The default value is 24. If the entire string does not fit in this space, Objectivity/DB stores a separate copy of it elsewhere.
Examples

These examples show how to specify a key description of a fixed maximum length and a variable maximum length.

```cpp
class Person {
    ooString(8) name;
}

ooHandle(ooKeyField) keyFieldH;
ooHandle(ooKeyDesc) keyDescH;
ooHandle(ooContObj) contH;

keyDescH = new(contH) ooKeyDesc(ooTypeN(Person), oocTrue);
keyFieldH = new(keyDescH) ooKeyField (ooTypeN(Person), "name" oocTrue, 8);  // specifies a fixed maximum length

class Person {
    ooString(256) name;
}

ooHandle(ooKeyField) keyFieldH;
ooHandle(ooKeyDesc) keyDescH;
ooHandle(ooContObj) contH;

keyDescH = new(contH) ooKeyDesc(ooTypeN(Person), oocTrue);
keyFieldH = new(keyDescH) ooKeyField (ooTypeN(Person), "name" oocFalse, 24);  // specifies a variable maximum length
```
Adding a Key Field to the Key Description

Once you have created a key field, you add it to the key description with the addField member function.

```cpp
ooStatus ooKeyDesc::addField(
    const ooHandle<ooKeyField>& fieldH);
```

where

`fieldH` Handle for the key field.

The return value is oocSuccess if the addition succeeded and oocError otherwise. The addField member function adds the field after the last one you added.

Examples

This example builds on the example on page 14-4 in which a key description was created for the Computer class. This example defines a key field for the key on the productName data member. It supplies a clustering directive using the handle created in the previous example, which refers to the key description.

```cpp
ooHandle<ooKeyField> fieldH;
fieldH =
    new(keyDescH) ooKeyField(ooTypeN(Computer), "productName");
```

This example adds an existing or newly-created field to a key description using the addField member function.

```cpp
ooHandle<ooKeyDesc> keyDescH;
ooHandle<ooKeyField> fieldH;
...
// keyDescH and fieldH are appropriately initialized
...
keyDescH->addField(fieldH);
```
Creating an Index from the Key Description

Once you have created the key description and added all the key fields to it, you can create indexes from the description using the `createIndex` member function on the key description object. You specify the scope of an index by passing a handle of the scope object to `createIndex`. The index is then created over all the objects of a given class in specified scope.

The syntax for `createIndex` is as follows:

```cpp
ooStatus ooKeyDesc::createIndex(
    const ooHandle(ooSystemObj)& scopeH);
```

where

- `scopeH` - Handle indicating the index scope. The scope can be a container, database, or federated database.
- `ooSystemObj` - Either `ooContObj`, `ooDBObj`, or `ooFDObj`

You can create indexes that reference different scopes from the same key description. Although you can only index basic objects from container indexes, you can index basic objects or containers from database and federated database indexes.

If the call to `createIndex` fails, Objectivity/DB aborts the transaction, after issuing messages that state that the index creation has failed and that the transaction is being aborted.
Creating an Index

Using Indexes 14-11

Examples

Having created the key description and key field for indexing Computer objects on their product names, you create an index for all of the Computer objects (including derived classes) in a specific scope.

```cpp
ooHandle(ooContObj) computerContH; // a Computer container
...
// computerContH refers to a Computer container
keyDescH->createIndex(computerContH);
```

You can also create an index for all of the computer objects (including derived classes) in a database scope.

```cpp
ooHandle(ooDBObj) computerDBH; // a database
...
// computerDBH refers to a database
keyDescH->createIndex(computerDBH);
```

Verifying an Index Is Unique

An index is unique if it was created using a key description whose unique parameter is set to oocTrue (see “Creating a Key Description” on page 14-3). This forces all indexed objects to have unique key field values.

You can use the member function `ooKeyDesc::isUnique` to verify whether or not an index was created to be unique. This member function returns oocTrue if the target index is unique and oocFalse if it is not.

The syntax for `isUnique` is as follows:

- `ooBoolean isUnique();`
Dropping an Index

To drop an index from a class, perform the following steps:

1. Use the dropIndex member function on the ooKeyDesc object to delete an index. See “Dropping an Index from the Key Description”.
2. Use the removeIndexes member function on the ooKeyDesc object to delete all the indexes created from a key description. See “Deleting All Indexes from a Key Description”.

Dropping an Index from the Key Description

At some point you may want to remove an index. For example, you might want to recreate the index with different characteristics, or you might want to reclaim the space used by the index.

To drop a single index, use the dropIndex member function.

```cpp
ooStatus ooKeyDesc::dropIndex(
    const ooHandle(ooSystemObj)& scopeH);
```

where

- `scopeH` Handle indicating the index scope. The scope can be a container, database, or federated database.
- `ooSystemObj` Either `ooContObj`, `ooDBObj`, or `ooFDObj`

Example

To remove the index created in the last example:

```cpp
keyDescH->dropIndex(computerContH);
```
Deleting All Indexes from a Key Description

If you want to drop all the indexes created from a particular key description, you can remove them all at once. If you no longer need the key description, you can delete the key description object, which propagates the delete to all its index objects. If you want to keep the key description but remove all the indexes, you can use the `removeIndexes` member function on the key description:

```cpp
ooStatus ooKeyDesc::removeIndexes();
```

Examples

To remove all of the indexes from the `computerKey` description, including the one in `computerContH`:

```cpp
keyDescH->removeIndexes();
```

You can also delete the key description object:

```cpp
ooDelete(keyDescH);
```

This deletes the key description, any indexes created from it, and the key field objects used by the key description.
Using the Indexes You Create

Once you have created indexes, you can use lookup keys for fast and efficient queries. This section describes how to set index modes and use lookup keys. Lookup keys are a low-level indexing feature. For most object lookups, you should use the predicate query feature. For more information, see “Using Predicate Query” on page 9-18.

Concurrency of Index Operations

Objectivity/DB allows concurrent multiuser access to both database- and federated database-level indexes. This allows multiple transactions to use an index to simultaneously access objects. However, at any given instant, only one transaction can update an index or its objects. These updates can occur during the transaction, before the transaction is committed.

Multiple users can access container-level indexes for read-only access. However, write access to a container-level index or its objects is restricted to a single user during a transaction. This is because container-level indexes rely on the container-level locking granularity of Objectivity/DB.

Consistency of Index Operations

Objectivity/DB ensures that any objects that are returned by an index are valid objects. This is accomplished by locking the corresponding objects in read mode (by default) or in update mode until the end of the transaction.

If your application requires a lower level of consistency but a higher concurrency of operations on a given index, then you can utilize the MROW feature. For MROW transactions, objects that are returned by an index exist in the current application’s version of the container.

For more information on using the MROW feature, see “Multiple Readers, One Writer (MROW)” on page 11-8.
Setting Index Modes

Index modes allow you to specify when the indexes on objects are updated when the key field values change. There are three index modes: sensitive, insensitive, and explicit update. You set your index mode using the enumerated type `ooIndexMode` as an argument to the `ooTrans::start` member function, specifying the `indexMode` parameter. For more information about `ooTrans::start`, see “Starting a Transaction” on page 2-29.

The `indexMode` parameter of `ooTrans::start` specifies one of three index modes—`oocInsensitive`, `oocSensitive`, and `oocExplicitUpdate`.

- `oocInsensitive`: Default and specifies that automatic updating of indexes occurs at commit time.
- `oocSensitive`: Updates to indexed fields are immediately and automatically reflected in the corresponding indexes. Immediate updates ensure that you do not have to wait until commit for changes to be reflected in the index. Scans of an index will be consistent. If you are going to use indexes during a predicate scan, this mode should be employed. See “Using Indexes” on page 9-21. The updates, if any, of the index will be done when `scan` is called and before the index entries are returned, or at commit time if no scans are done.
- `oocExplicitUpdate`: Gives you explicit control over changes to indexed objects during a transaction. Using `oocExplicitUpdate` will improve performance in certain update-intensive applications, where database- and federated database-level indexes are used, but where the updated members are not indexed. You must call the `ooUpdateIndexes` function when using this mode. See “Explicitly Updating Indexes” on page 14-16. A warning will be issued if `ooUpdateIndexes` is called when the index update mode is not `oocExplicitUpdate`. 

Using Indexes 14-15
Explicitly Updating Indexes

You can use explicit update to improve index performance when your application is update intensive and utilizes database- and federated database-level indexes. Because Objectivity/DB is not aware you have created or updated an object (for instance, when initial data values for your object’s member functions have been assigned), you must call `ooUpdateIndexes` to update the data after you have created the object and initialized all its indexed fields, or after you have updated that object.

To explicitly update indexes, call the function `ooUpdateIndexes` after your application has created or modified an object. `ooUpdateIndexes` finds all the indexes that index the object just created or modified. The syntax for `ooUpdateIndexes` is as follows:

```c
ooStatus ooUpdateIndexes(ooHandle(ooObj) &pHandle);
```

where

- `pHandle` Handle to the modified or new object of type `ooObj`, `ooContObj`, or any derived types of `ooObj` or `ooContObj`.

Examples

This example shows how to insert a new object into all applicable indexes. We assume that `setH` is a handle to the new object’s container.

```c
// DDL schema definition of the class Point
class Point: public ooObj {
 Publication:
     char name[32];
     int32 xCoord;
     int32 yCoord;
}
```
The following example modifies a `Point` object that has already been inserted into indexes by setting its `XCoord` value to 30. We assume `point2H` is a valid handle to the `Point` object to be updated.

```c
// Application code
// Update the object
point2H->xCoord = 30;
// Call ooUpdateIndexes to update the appropriate indexes
ooUpdateIndexes (point2H);
```

**Lookup Keys**

Lookup keys are used to retrieve objects from an index. A lookup key is a transient collection of lookup key fields, each of which contains a relational operator (such as equal or greater than) and a comparison value, as shown in Table 14-1.

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td><code>ooEqualLookupField</code></td>
</tr>
<tr>
<td>&gt;</td>
<td><code>ooGreaterThanLookupField</code></td>
</tr>
<tr>
<td>&lt;</td>
<td><code>ooLessThanLookupField</code></td>
</tr>
<tr>
<td>&gt;=</td>
<td><code>ooGreaterThanEqualLookupField</code></td>
</tr>
<tr>
<td>&lt;=</td>
<td><code>ooLessThanEqualLookupField</code></td>
</tr>
</tbody>
</table>
To search a scope using a lookup key, perform the following steps:

1. Create a lookup key object. See “Creating a Lookup Key Object” on page 14-18.
2. Create the lookup key fields with the appropriate lookup field derived classes and values, and add the lookup key fields to the lookup key. See “Creating the Lookup Key Fields” on page 14-20.
3. Select a scope to search. See “Selecting a Scope” on page 14-22.
4. Declare an iterator of type `ooItr(className)`, and initialize the iterator using the scope and lookup key by calling the iterator `scan` member function. See “Initializing an Iterator to Use the Index and Lookup Key” on page 14-23.
5. Iterate in the usual way through objects that satisfy the conditions that the lookup key specifies. See “Iterating Over Indexes” on page 14-24.

**Creating a Lookup Key Object**

First, you need to create the lookup key object, using the following syntax:

```cpp
ooLookupKey::ooLookupKey(
    const ooTypeNumber typeN,
    const uint32 number);
```

where

- `typeN`: Type number of the class of objects. See “Obtaining a Class Type Number” on page 20-2. For the lookup key object to be valid and usable, `typeN` must match exactly with `typeN` that was used in creating the index, as described in “Creating a Key Description” on page 14-3.
- `number`: Number of fields you intend to add to the lookup key.

You can only create lookup keys within an operative transaction. These keys become invalid after the transaction is committed.

---

**Example**

To create a lookup key that you can use to search for `Computer` objects by name, you would create a lookup key object this way:

```cpp
ooLookupKey lookupKey(ooTypeN(Computer), 1);
```
Lookup Key Field Constraints
The number of fields in the lookup key is fixed once you have created the key object. You cannot add more fields to the key than specified by number, but you can add fewer fields.

You can include a field more than once. For example, you can define a search range for a field value, $23 < \text{value} < 76$.

Optimizing Lookups
Object lookup using indexes works faster when the:

- Search on the index involves a relatively few number of keys. The more comparisons you specify, the more work is done in the lookup.
- Lookup key contains a specification for the key fields that must include the most major key field, and may also include other minor key fields listed in order beginning with the most major, as specified in the index key. If more than one key field is specified, only the last one can be something other than $=$.

Example

Suppose you are given a description of an index key with each field of type integer and the key fields in order plantNumber, deptNumber, groupNumber, projLeaderId. If you build a search key containing all four fields with equality relational conditions in the order of the key fields, the lookup will be much faster. For example, the following lookup key would make good use of an index:

- $\text{plantNumber} = 1$
- $\text{deptNumber} = 6$
- $\text{groupNumber} > 15$
In addition, you can specify a range to a key field more than once. The following search key looks for groups with group numbers ranging from 15 to 30:

- plantNumber = 1
- deptNumber = 6
- groupNumber >= 15
- groupNumber <= 30

---

**Creating the Lookup Key Fields**

You create the lookup key fields and add them to the lookup key. If more than one field is added to the lookup key, each object returned from the lookup will satisfy the constraints of all the fields in the lookup key.

The constructors for the various derived classes of the abstract class ooLookupFieldBase all take the following arguments:

```cpp
className::className(
    const ooTypeNumber typeN,
    const char* memberName,
    const void* valuePtr);
```

where

- **className**: Derived class name (ooEqualLookupField, ooGreaterThanLookupField, ooLessThanLookupField, ooGreaterThanEqualLookupField, or ooLessThanEqualLookupField).
- **typeN**: Type number of the class containing the member to which the key field applies.
- **memberName**: Name of the member in the class.
- **valuePtr**: Address of the data value to use in the relational comparison. The type of this value comes from the schema, which stores the type associated with the member name you build into the key field object. Using data of any other type may have unpredictable results.
You first create a transient object that is an instance of a class derived from \texttt{ooLookupFieldBase} (for example, \texttt{ooLessThanLookupField}). Then you add the lookup key field to the lookup key using the \texttt{addField} member function on class \texttt{ooLookupKey}:

\begin{verbatim}
  ooStatus ooLookupKey::addField(
      const ooLookupFieldBase& field);
\end{verbatim}

where

\begin{itemize}
  \item \texttt{field} \quad \text{Lookup field to add to the lookup key.}
\end{itemize}

When using the \texttt{ooLookupKey::addField} member function for indexing operations, the field to be added must remain in scope during the iteration sequence. To get optimal performance from the lookup, add the fields in the same order they appear in the key description, as discussed in “Creating Key Fields” on page 14-5.

**Example**

To create a lookup key field that compares the \texttt{Computer} name to the name \texttt{PEGASUS} and add it to the key created in the example on page 14-18:

\begin{verbatim}
  ooEqualLookupField lookupField(ooTypeN(Computer),
      "productName", "PEGASUS"),
  lookupKey.addField(lookupField);
\end{verbatim}
Selecting a Scope

Once you have defined the lookup key and its fields, you can search any container, database, or federated database within your defined scope using the lookup key. You can only use the lookup key for scopes that have at least one index with the key fields in your lookup key. If you initialize an iterator on a scope without such indexes, you will not retrieve any objects, since there is no index through which to retrieve them. This is not an error condition; you just get a null iterator.

For performance reasons, you should call the `scan` function with a lookup key argument only for scopes that you know have relevant indexes. Lookup key fields need to match (or be a subset of) the key fields of at least one of the indexes. You need to keep track of this in your application, or you can use the `anyIndex` member function on the `ooLookupKey` object to find out whether a scope has any index that you can search.

The `anyIndex` member function will accept container, database, and federated database handles. This member function returns `oocTrue` if there is an index within the defined scope that references at least one of the lookup key fields. Indexes can only be used if the lookup key fields used match the first $n$ keys in the index specification. $n$ is a number from one to the number of fields in the index. If $n$ is less than the number of key fields in the lookup key, you have to filter the output for the remaining fields.

For example, the following table shows `anyIndex` returns values for various combinations for index fields ($x$, $y$, and $z$) and corresponding lookup keys. Note that for the last entry in this table, an index can be used to search for $x$, but you must filter the output for $z$.

<table>
<thead>
<tr>
<th>Index Fields</th>
<th>Lookup Key Fields</th>
<th>anyIndex Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x$</td>
<td><code>oocTrue</code></td>
</tr>
<tr>
<td>$x$</td>
<td>$y$</td>
<td><code>oocFalse</code></td>
</tr>
<tr>
<td>$x$, $y$</td>
<td>$x$</td>
<td><code>oocTrue</code></td>
</tr>
<tr>
<td>$x$, $y$</td>
<td>$x$, $y$</td>
<td><code>oocTrue</code></td>
</tr>
<tr>
<td>$x$, $y$</td>
<td>$y$</td>
<td><code>oocFalse</code></td>
</tr>
<tr>
<td>$x$, $y$, $z$</td>
<td>$x$</td>
<td><code>oocTrue</code></td>
</tr>
<tr>
<td>$x$, $y$, $z$</td>
<td>$x$, $y$</td>
<td><code>oocTrue</code></td>
</tr>
</tbody>
</table>
If an index is used by an iterator and the index does not reference all of the lookup key fields, the iterator applies only to those lookup key fields that are referenced by the index, and it is up to you to apply those non-referenced key fields against the object returned by the iterator.

\[
\text{ooBoolean ooLookupKey::anyIndex(}
\begin{align*}
&\text{const ooHandle(ooSystemObj)& scopeH) const;}
\end{align*}
\]

where

- **scopeH** Handle indicating the scope you want to test for a relevant index. The scope can be a container, database, or federated database.
- **ooSystemObj** Either ooContObj, ooDBObj, or ooFDObj

If this member function returns `oocTrue`, then you can use `scan` on an iterator to look up objects using an index.

**Initializing an Iterator to Use the Index and Lookup Key**

Once you have created the lookup key, the next step is to initialize an iterator using the lookup key and the scope for the index. You can use the member function `scan` of `ooItr(className)` to initialize the iterator.

The syntax for `ooItr(className)` is as follows:

\[
\text{ooStatus ooItr(className)::scan(}
\begin{align*}
&\text{const ooHandle(ooSystemObj)& scopeH,}
&\text{const ooLookupKey& lookupKey,}
&\text{const ooMode openMode = oocNoOpen);}\end{align*}
\]

where

- **className** Class of objects you are looking up
- **scopeH** Handle indicating the scope. The scope can be a container, database, or federated database.
- **ooSystemObj** Either ooContObj, ooDBObj, or ooFDObj
- **lookupKey** Object iterator
openMode

Optional parameter that specifies the mode in which Objectivity/DB opens each object when you call the next member function on the iterator.

The member function returns oocSuccess if it correctly initializes the iterator. An error will be issued if any other handle is used for the scope.

Iterating Over Indexes

Once initialized, you use the iterator in the standard way to iterate through the indexed objects that satisfy the conditions that the lookup key specifies. Iterating over an index returns objects in ascending order.

To iterate over an index using an ooLookupKey and an ooLookupField, both of these objects need to be in scope at the time of the iteration. The iteration will not work if the above objects have been destructed.

Examples

The following DDL schema file defines the class Point:

class Point : public ooObj {
    Public:
        char name[32];
        int32 xCoord;
        int32 yCoord;
}

The following application code creates a unique key description object, defines an index key by creating key fields and adding them to the key, and creates an index for class Point using the new operator in the same container that contains the key description:

    ooHandle(ooKeyField) keyFieldH;
    ooHandle(ooKeyDesc) keyDescH;
    ooHandle(ooContObj) contH;
// Assume contH is set to a valid container
// Create the key description object in the container contH.
keyDescH = new(contH) ooKeyDesc(ooTypeN(Point), oocTrue);

// Define the index key
keyFieldH = new(keyDescH) ooKeyField(ooTypeN(Point), "xCoord");
keyDescH->addField(keyFieldH);
keyFieldH = new(keyDescH) ooKeyField(ooTypeN(Point), "yCoord");
keyDescH->addField(keyFieldH);

// Create the actual index.
ooStatus status = keyDescH->createIndex(contH);

If you want to iterate explicitly over those points with x coordinate equal to 100 and y coordinate greater than 50, you must use a lookup key specifying those conditions. The following application code creates a lookup key and initializes an iterator over a container that will iterate over those basic objects that match the conditions:

// Scan another container for points satisfying a condition.
// Iterate through the points and print each one.
ooHandle(ooContObj) contH;
ooItr(ooPoint) pointI;
const int xPoint = 100; // x == 100
const int yMinPoint = 50; // y > 50

// Assume contH is set to a valid handle value.
// Create the lookup key object.
ooLookupKey lookupKey(ooTypeN(Point), 2);

// Create the lookup key fields and add them to the key.
ooEqualLookupField xLookupField(ooTypeN(Point), "xCoord", &xPoint);
lookupKey.addField(xLookupField);
ooGreaterThanLookupField yLookupField(ooTypeN(Point), "yCoord", &yMinPoint);
lookupKey.addField(yLookupField);
// Initialize the iterator to look up the objects.
status = pointI.scan(contH, lookupKey);
if (status){
    // The iterator is now available for use.
    while (pointI.next())
    {
        cout << "Found point at (" << pointI.xCoord << ", " << pointI.yCoord << ")." << endl;
    }
}

**Using Indexes with International String Collations**

Objectivity/DB uses the standard C function `strcoll` when comparing strings for index ordering, enabling you to use international collations with Objectivity/DB indexes. To use international collations, you must use the proper locale table configuration, and call the `setlocale` function with the proper arguments. For more information, refer to your operating system documentation.
This chapter describes how to use keyed objects. In this chapter, where the syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form const ooHandle(className) & are interchangeable with const ooRef(className) &.

Objectivity/DB provides a keyed objects feature that allows you to quickly locate objects within a hashed container based on data member values. The key used for the object lookup can be any integer or string field in a basic object. Keyed objects are used for stable data that needs to be looked up quickly.

Figure 15-1 illustrates the use of keyed objects. In this figure, a field size is identified as a key for the class Rectangle. Here, basic objects rect1 and rect2 were created as keyed objects with their size field as a key. Using this key, rect1 and rect2 can be quickly located using a search algorithm. Basic object rect3, however, was not created as a keyed object. Even though rect3 is of class Rectangle it cannot be looked up using its size field as a key.
Keyed Object Procedure Summary

To create and use a keyed object perform the following steps:

1. Identify the persistent class (`ooObj` or a user-defined derived class) to be used to create the keyed objects.

2. Decide which field in the class will serve as the key. You may define only one key field per class.

3. Use the `ooGetMemberOffset` and `ooGetMemberSize` macros to initialize a key structure with information about the key field selected in the previous step, including the value of the key for the object. See "Initializing a Key Structure" on page 15-3.

4. Create a keyed object using the `ooNewKey` macro. `ooNewKey` takes as parameters the class name, constructor parameter list, the handle to the hashed container to contain the keyed objects, and the key structure that describes the key. Objects that are not created using `ooNewKey` cannot be located by key. See “Creating a Keyed Object” on page 15-5.

5. Access a keyed object using the key structure you defined in step 3.
6. Manipulate a keyed object. Once you retrieve a keyed object using a key, you can manipulate in the same way as any other persistent object. However, if you change its key value, you will no longer be able to locate the object by its key.

**Initializing a Key Structure**

Key structures provide Objectivity/DB with information about a particular key field, including what value of the field you wish to search for. The key structure is defined as:

```c
typedef enum {
    oocUint16, // 16 bit unsigned
    oocUint32, // 32 bit unsigned
    oocInt16, // 16 bit signed
    oocInt32, // 32 bit signed
    oocString, // string type (char*)
    oocCharArray // character array
} ooKeyType;

typedef struct {
    ooKeyType type; // key type
    uint32 offset; // offset of key from start of object
    uint32 size; // length of key field in bytes
    ooVoidPtr value; // pointer to key value
} ooKey;
```

For a key of type `oocString`, there is a null terminator in the fixed array of characters and the `size` field is ignored. No such assumption is made for a key of type `oocCharArray`.

The following macros are available to help you initialize a key structure:

The `ooGetMemberOffset` macro returns the byte offset of the member field `memberName` from the start of class `className`. The data member `memberName` must be public.

- `uint32 ooGetMemberOffset(
    className,
    memberName);`
The `ooGetMemberSize` macro returns the size in number of bytes of the member field `memberName` in class `className`. The data member `memberName` must be public.

```c
uint32 ooGetMemberSize(
    className,
    memberName);
```

The key value need not be unique for each keyed object created. However, if multiple keyed objects of the same type are created with the same key value, only one of the objects can be located using the keyed object lookup facility.

---

**Example**

This example initializes a key structure to create and look up keyed objects of class `Rectangle`. The member field `rectName` is used as the key field.

```c
ooKey nameKey;
char keyValue[80];
...
// Initialize the rectName key structure
nameKey.type = oocString;
nameKey.offset = ooGetMemberOffset(Rectangle, rectName);
nameKey.size = ooGetMemberSize(Rectangle, rectName);
nameKey.value = keyValue;
strcpy(keyValue,"rect1");
...
```
Creating a Keyed Object

Use the following macro to create a keyed object:

\[
\text{ooHandle}(\text{className}) \ \text{ooNewKey}(
\begin{array}{l}
\text{className}, \\
(\text{initializer}), \\
\text{const} \ \text{ooHandle}(\text{ooObj}) & \text{contH}, \\
\text{ooKey} \ \text{keyStruct});
\end{array}
\]

where

- **className**: Class of the keyed object to create. `className` must be `ooObj` or a user-defined derived class.
- **initializer**: List of arguments passed to the constructor for class `className`.
- **contH**: Handle specifying the container in which the keyed object is created. This must be a handle to a hashed container. If the handle passed is that of a database, the default container is used.
- **keyStruct**: Key structure containing information about the key field `ooNewKey` supplies the key field for the keyed object.

You cannot create keyed objects within a non-hashed container.

---

**Example**

This example creates a keyed object of class `Rectangle`, using the `rectName` field as the key. The keyed object is created in the container referenced by the handle `contH`.

```c
ooKey nameKey;
ooHandle(Rectangle) rectH;
ooHandle(ooContObj) contH;
char keyValue[80];
...
// contH is initialized to a valid handle
...
```
Locating a Keyed Object

Locating a Keyed Object

To locate a keyed object:

1. Initialize a key structure with information about the key field, including the value you wish to match. See "Initializing a Key Structure" on page 15-3.
2. Use the lookupObj member function using the key structure initialized in the previous step. This member function is invoked on an uninitialized handle for the class that is to be searched.

Objectivity/DB will only search objects of the class that were created using the ooNewKey macro with the specified key field.

To search for a keyed object with a particular key value, use the following member function:

```c
ooStatus ooRefHandle(className) :: lookupObj ( 
    const ooHandle(ooObj)& scopeH, 
    const ooKey& keyStruct, 
    const ooMode openMode = oocRead);
```

where

- **className** ooObj or ooContObj, or a derived class
- **scopeH** Handle specifying the container in which to search. This container must be hashed. If the handle passed is that of a database, the default container is used.
- **keyStruct** Key structure containing information about the key field, including the key field value to match

Locating a Keyed Object

// Initialize the key structure.
nameKey.type = oocString;
nameKey.offset = ooGetMemberOffset(Rectangle, rectName);
nameKey.size = ooGetMemberSize(Rectangle, rectName);
nameKey.value = keyValue;

... // Create a keyed object with key value rect1
strcpy(keyValue, "rect1");
rectH = ooNewKey(Rectangle, (), contH, nameKey);
...
**openMode**

Optional parameter which specifies the mode in which to open the object. May be either the constant `oocRead` or the constant `oocUpdate`. If not specified, the default is `oocRead`.

---

### Example

This example looks up the keyed object of class `Rectangle` that has a value of 12 for key field `lowerLeftX`.

```c
ooKey llxKey;
ooHandle(Rectangle) rectH;
ooHandle(ooContObj) contH
int32 keyValue;
...
// contH is initialized to reference the container that holds the keyed objects
...
// Initialize the lowerLeft key structure.
llxKey.type = oocInt32;
llxKey.offset = ooGetMemberOffset(Rectangle, lowerLeftX);
llxKey.size = ooGetMemberSize(Rectangle, lowerLeftX);
llxKey.value = &keyValue;

// Retrieve the object using the llxKey key structure.
keyValue = 12;
rect1H.lookupObj(contH, llxKey);
```
Manipulating a Keyed Object

Once you have located a keyed object, you may treat it exactly like a non-keyed basic object. All of the member functions defined on the keyed object and its handle behave the same as basic objects. See “Invoking Member Functions” on page 4-9 for information on using object handles.

If you change the value of the key field, you will not be able to locate the object again using the key. However, you may still access it using the means available for non-keyed basic objects. You can, of course, delete the existing object and create a new one with ooNewKey.
Versioning

Versioning is a powerful feature of Objectivity/DB that allows you to maintain many copies (also called versions) of the same basic object. Only basic objects (objects of class ooObj and any user-defined classes derived from ooObj) can be versioned.

Objectivity/DB provides two interfaces to versioning, which you can mix to provide the desired functionality.

◆ An object reference and handle interface for basic versioning
  Objectivity/DB includes a set of predefined object reference and handle member functions that provide a simple interface to versioning. These member functions have specific versioning semantics that should provide the functionality necessary for most user applications.

◆ An association interface for advanced versioning
  The versioning features of Objectivity/DB are implemented through the use of system-defined associations. You can also access these associations to create customized versioning semantics. This is a fairly sophisticated interface, requiring you to have a thorough understanding of associations and their use.

The process of creating a new version of an object is the same for both interfaces. The choice of which interface to use is driven by the versioning semantics necessary for your application.

Where syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle.
Basic Versioning Techniques

While each version copy is an independent persistent object, each is also tracked by Objectivity/DB as being a particular version of a single object. The semantics of a new version are up to your application. It is common for versions to indicate a change in function or a checkpoint in time. You can change the versioning behavior of any basic object at any time.

There are two ways in which to version an object, as shown in Figure 16-1:

- **Linear**
  - Linear versioning allows only a single new version to be created from a given object.

- **Branch**
  - Branch versioning allows an arbitrary number of versions to be created from a given object.

![Version Behavior Diagram](image_url)

**Figure 16-1** Version Behavior
Version Genealogy

The collection of, and relationship between, all versions of a basic object is called the version genealogy. You can use both linear and branch versioning within a particular version genealogy, and you can indicate a copy of the basic object as being the default version. Objectivity/DB provides a mechanism to allow you to quickly locate the default version.

Figure 16-2 shows a typical version genealogy, where version V5a is shaded to indicate that it is the default version. Note that branch versioning was used on versions V1 and V3b.

As an example of versioning, consider the design of a hypothetical automobile, the OMW (Objectivity Motor Works) DeeBee. Within OMW’s federated database is a container named DeeBee that contains all of the design components (basic objects) for this car.

Each year, the default version is updated to be the latest American design. For 1994, OMW plans to discontinue the Canadian model and market a 1994 American version derived from the 1993 American and Canadian models.

**Key to Symbols**

- ○ = New version of DeeBee car
- □ = Default version of DeeBee car
- → = Derived from

**Figure 16-3**  Versioning OMW's DeeBee Car
Enabling and Disabling Basic Versioning

Every versionable object has an associated versioning attribute of type ooVersMode, which determines its versioning behavior. The valid values are:

- **oocNoVers**: The versioning behavior of the object is disabled; that is, you cannot create a version of the object. This is the default mode for an object when it is created.
- **oocLinearVers**: The object exhibits linear versioning behavior; that is, you may create a version of this object only if no other version exists.
- **oocBranchVers**: The object exhibits branch versioning behavior; that is, you may create any number of versions of this object.

To set and query the current versioning mode of an object, use the following member functions:

- **ooStatus ooRefHandle(ooObj)::setVersStatus(const ooVersMode versMode) const;**
- **ooVersMode ooRefHandle(ooObj)::getVersStatus() const;**

where

- **versMode** is the versioning mode (the constant oocNoVers, oocLinearVers, or oocBranchVers).

---

Example

This example sets the versioning behavior of the object indicated by handle rectH to linear versioning.

```cpp
ooHandle(Rectangle) rectH;
...
// rectH is set to a valid handle value
...
if (rectH.getVersStatus() == oocNoVers) {
    rectH.setVersStatus(oocLinearVers);
}
```
Creating a Version

When a version of an object is created, the contents of the old version are copied over to the new version, using a bit-wise copy. If the versioning is linear, then the versioning status of the old version is set to oocNoVers, and the versioning status of the new version is set to oocLinearVers. If the versioning is branch, then the versioning status of the old version will remain set to oocBranchVers, and the new versioning status of the new version will also be set to oocBranchVers.

Associations to the old version will be transferred over to the new version according to the link’s versioning semantics. See the “Data Definition Language” chapter of Using Objectivity/ C++ Data Definition Language for information on association versioning semantics.

Note that a new version does not automatically become the default version. See “Setting the Default Version” on page 16-13 for details on the default version.

Creating a Version Using Linear Versioning

To create a new version of an object using linear versioning:

1. Set the versioning mode of the object to oocLinearVers.
   Do this by invoking the setVersStatus member function on the handle that references the object.

2. Make sure the object is closed.
   If it is open, use the close member function on its handle to close it. If there is more than one handle pointing to the object, all must be closed.

3. Open the object in update mode.
   Do this by using the open or update member function on its handle. If a version of this object already exists, the open will fail. If not, the handle will reference the new version.
Example

This example creates a new version of the object referenced by `cellH` using linear versioning. It also uses the `update` member function to open the object for update.

```cpp
ooHandle(Cell) cellH; // Cell is a subclass of ooObj
...
// cellH is set to a valid handle value
...
cellH.setVersStatus(oocLinearVers); // enable linear versioning
cellH.close(); // make sure it is closed
cellH.update(); // create new version
cellH.setVersStatus(oocNoVers); // disable versioning of new
```

Creating a Version Using Branch Versioning

To create a new version of an object using branch versioning:

1. Set the versioning mode of the object to `oocBranchVers`.
   Do this by invoking the `setVersStatus` member function on the handle that references the object.
2. Make sure the object is closed.
   If it is open, use the `close` member function on its handle to close it.
3. Open the object in update mode.
   Do this using the `open` or `update` member function on its handle. The handle now refers to the new version.
Example

This example creates two new versions of the object referenced by cellH, using branch versioning. It also uses the update member function to open the object for update.

```c++
ooHandle(Cell) cellH, oldVersH;
...
// cellH is set to a valid handle value
...
oldVersH = cellH; // save handle to old version

cellH.setVersStatus(oocBranchVers); // enable branch versioning
oldVersH.close();
cellH.close();
cellH.update(); // create the branch version
cellH = oldVersH;
cellH.update(); // create the second branch version
```

Disabling Versioning Behavior

You may want to disable versioning of the old or new object versions immediately after creating the new version. A user intending to open a version for update can inadvertently create a new object, if versioning is enabled.

In the case of linear versioning, the versioning behavior of the old version is automatically disabled for you. Thus, to disable versioning for an object that has just been linearly versioned, set the versioning mode of the new version to oocNoVers.

Do this by invoking the setVersStatus member function on the handle of the new version.
In the case of branch versioning, you must disable versioning of both the old and the new versions. Thus, to disable versioning for an object that has just been branch versioned, do the following:

1. Set the versioning mode of the new version to `oocNoVers` by invoking the `setVersStatus` member function on the handle of the new version.
2. Obtain an object reference or handle to the old version by either saving an object reference or handle to the old version before creating the new one, or by invoking the `prevVers` member function on the new version, which returns an object reference or handle to the old version.
3. Set the versioning mode of the old version to `oocNoVers` by invoking the `setVerStatus` member function on the handle of the old version.

---

**Example**

This example creates a new version of the object referenced by `cellH`, using the `update` member function to open the object for update. It also disables versioning of the original version and new version after the new version is created.

```cpp
ooHandle(Cell) cellH, oldVersH;
...
// cellH is set to a valid handle value
...
oldVersH = cellH; // save handle to old version
cellH.setVersStatus(oocBranchVers); // enable branch versioning
oldVersH.close();
cellH.close();
cellH.update(); // create new branch version
cellH.setVersStatus(oocNoVers); // disable versioning of new
oldVersH.setVersStatus(oocNoVers); // disable versioning of old
```
Specifying Copy Semantics

When creating a new version of an object, Objectivity/DB performs a bit-wise copy of the existing object, possibly adding some associations. No copy constructor or assignment operator is invoked. Whether the semantics of creating a new version should be the same as that of a copy constructor or an object assignment depends on the application.

Using the virtual function `ooNewVersInit`, you can specify the exact semantics that fits the version copy requirements of your application. You can do this by overloading `ooNewVersInit` in your application's persistent classes. The function does nothing but return a success status.

```
virtual ooStatus ooObj::ooNewVersInit();
```

During the creation of a new version, the system will automatically invoke the virtual function `ooNewVersInit`.

---

**Example**

```
ooHandle(Part) partH;

... // Create a new version. The virtual function ooNewVersInit // for the Part object is invoked after the bit-wise copy.
partH.update();
```
Locating the Next Version

To obtain an object reference or handle to the next version (if linear versioning was used) or versions (if branch versioning was used) of an object, use the $\texttt{ooRefHandle(ooObj)::getNextVers}$ member function:

$$\texttt{ooStatus ooRefHandle(ooObj)::getNextVers(}
\texttt{ ooHandle(ooObj) \& nextVersItrH,}
\texttt{ const ooMode openMode = oocNoOpen) const;}$$

where

- $\texttt{nextVersItrH}$: Iterator initialized to traverse all versions derived from this object
- $\texttt{openMode}$: Mode in which objects of the iteration are automatically opened. Default is $\texttt{oocNoOpen}$; that is, they are not opened.

If no next versions exist, an empty iterator is returned (the first $\texttt{next}$ member function invocation will return $\texttt{oocFalse}$).

⚠️ Warning

Be careful when you use $\texttt{openMode = oocUpdate}$. If versioning is enabled on a version, you will actually create a new version instead of opening the next version for update.

✎ Example

This example initializes the iterator $\texttt{nextI}$ to traverse all versions created from the object referenced by $\texttt{objH}$.

```cpp
ooHandle(Rect) objH; // class Rect is a subclass of ooObj
ooItr(Rect) nextI;
...
// objH is set to a valid handle value
objH.getNextVers(nextI);
```
Locating the Previous Version

To obtain an object reference or handle to the previous version of an object, use the 
`ooRefHandle(ooObj)::getPrevVers` member function:

- `ooStatus ooRefHandle(ooObj)::getPrevVers(
  ooRef(ooObj)& prevId) const;`
- `ooStatus ooRefHandle(ooObj)::getPrevVers(
  ooHandle(ooObj)& prevH) const;`

where
- `prevID` Object reference set to the previous version of this object
- `prevH` Handle set to the previous version of this object

If no previous version exists, a null object reference or handle is returned.

Example

This example sets the handle `prevH` to reference the previous version of the object 
referred by handle `objH`. It also tests to see whether or not a previous version 
actually exists.

```c
ooHandle(Rect) objH, prevH; // class Rect is a subclass of ooObj
// objH is set to a valid handle value
...
objH.getPrevVers(prevH);
if (prevH == 0 {
    printf("No previous version exists\n");
} else {
    printf("There is a previous version\n");
}
```

16-12 Using Objectivity/C++
### Setting the Default Version

Objectivity/DB allows you to specify a single version within a genealogy as the default version. The semantics of the default version are up to you; the default version need not be the most recent version. For example, you may want to identify the latest useful version you want users to access.

To set the default version for a genealogy, use the following member function:

```cpp
ooStatus ooRefHandle(ooObj)::setDefaultVers() const;
```

This member function sets the referenced object to be the default version within its genealogy.

**Note**

Within a genealogy, you may access the default version only from the default version itself and from those versions created after the initial default version was designated.

**Example**

This example sets the object referenced by handle `objH` to be the default version within the genealogy of which it is a member.

```cpp
ooHandle(Rect) objH; // class Rect is a subclass of ooObj
...
// objH is set to a valid handle value
...
objH.setDefaultVers();
```
Locating the Default Version

You may obtain an object reference or handle to the default version from the default version itself and from any other version in the genealogy created after the initial default version was specified. This behavior reflects the fact that versions created before a default version existed cannot know about a default version.

To locate the default version, use the following member function on an object reference or handle that references a version within the genealogy:

- `ooStatus ooRefHandle(ooObj)::getDefaultVers(ooRef(ooObj)& defId) const;`
- `ooStatus ooRefHandle(ooObj)::getDefaultVers(ooHandle(ooObj)& defH) const;`

where

- `defId` Object reference set to the default version
- `defH` Handle set to the default version

This member function returns a null object reference or handle if no default version exists for the genealogy, or if the default version was specified after the referenced version was created.

You cannot locate the default version from a version created before the initial default version was assigned.

To locate the default from any version, either specify the original version as the default before creating any other versions, or create your own versioning semantic using the association interface to versioning (see “Advanced Versioning Techniques” on page 16-16).
Example

This example creates a genealogy by creating two versions of the object referenced by handle \texttt{vers1H}. Note that because the default version is not set until after the creation of the first version, \texttt{getDefaultVers} will return a null handle when invoked on \texttt{vers1H} which was created before the default was set.

\begin{verbatim}
ooHandle(Cell) vers1H, vers2H, vers3H, workH, defaultH;
...
    // vers1H is set to a valid handle value
...
    // Assign working variable workH.
    workH = vers1H;
    // Enable linear versioning
    workH.setVersStatus(oocLinearVers);
}
vers1H.close();
workH.close();
workH.update();    // Create new version
vers2H = workH;    // Save handle of new version
vers2H.setDefaultVers(); // Set to be the default version
workH.close();
workH.update();    // Create new version
vers3H = workH;    // Save handle of new version
// Now use \texttt{getDefaultVers} to get the default version
vers1H.getDefaultVers(defaultH);    // Returns null handle
vers2H.getDefaultVers(defaultH);    // defaultH == vers2H
vers3H.getDefaultVers(defaultH);    // defaultH == vers2H
\end{verbatim}
Advanced Versioning Techniques

This section describes advanced versioning concepts and the association interface that allows you to create customized versioning semantics. You should first read “Basic Versioning Techniques” on page 16-2, which covers the basic concepts of versioning and have a good understanding of Objectivity/DB associations and how they are used.

Genealogy Object

A version genealogy is a recorded history of the versioning of an object. The genealogy may consist of several persistent objects, but from a logical standpoint, it actually represents a single object. The genealogy object is a mechanism that allows other objects to associate with the genealogy as a whole. Every version genealogy can have an associated genealogy object.

The genealogy object:
- Serves as a multiplexer, allowing objects to associate to the genealogy as a whole (see Figure 16-4)
- Can be customized to hold information that pertains to the genealogy as a whole
- Necessary to support the default version of a genealogy. Thus, if a genealogy has a default version, it must also have a genealogy object.

From the object reference and handle interface a genealogy object is automatically created for you when you invoke the `setDefaultVers` member function. However, you may not associate to or customize a genealogy object created in this manner. In this case its sole purpose is to support the default version of the genealogy.

To use the genealogy object to its full potential you must create it manually and use the association interface to versioning.

The remainder of this section discusses how to create a custom genealogy object. For a discussion on how to set up the genealogy object so that it can access each version of the genealogy (including the default version), see “The Default Version” on page 16-29.
Figure 16-4  Genealogy Object

Key To Symbols

- Standard Association
- Genealogy Association
- Default Version Association

G = Genealogy Object
Vx = Objects In Genealogy
V5 = Default Version
= Object Not In Genealogy
Creating a Customized Genealogy Object

The genealogy object is an instance of class ooGeneObj or of a user-defined subclass of ooGeneObj. An instance of ooGeneObj is created automatically by the setDefaultVers member function.

You can also define a customized genealogy object class by creating a subclass of ooGeneObj. This allows you to define data fields and association links which other objects may then access.

Example

Assume that we have a genealogy consisting of instances of class Rect (a subclass of ooObj) that does not have a default version. We wish to create a customized genealogy object class named RectGeneObj that does the following:

- Defines a field called empId that stores the employee ID of the engineer responsible for the genealogy
- Defines a many-to-one association link to class Layer, allowing an instance of Layer to associate to one or more genealogy objects

The DDL declaration for class RectGeneObj is as follows:

```cpp
class RectGeneObj : public ooGeneObj {
public:
    uint32 empId;
    ooHandle(Layer) owningLayer <-> componentRect[];
}
```

The corresponding DDL declaration for class Layer is:

```cpp
class Layer : public ooObj {
public:
    ...
    ooHandle(RectGeneObj) componentRect[] <-> owningLayer;
}
```
The following code segment creates a genealogy object of class `RectGeneObj`, sets the `empId` field, and associates it to an instance of class `Layer`. Note that at this point the instance of `Layer` does not have access to individual versions within the genealogy, because the genealogy object has not been set up to act as a multiplexer. See “The Default Version” on page 16-29 for a discussion on how to set up the genealogy object to serve this function.

```c
ooHandle(RectGeneObj) objH;
ooHandle(ooContObj) contH;
ooHandle(Layer) layerH;
ooItr(RectGeneObj) geneI;

... // contH set to a valid container handle; layerH set
    // to a valid handle

... // Create the new genealogy object and set empId field to 20
objH = new(contH) RectGeneObj();
objH->empId = 20;
// Associate instance of Layer to genealogy object and print
// out the empId value
layerH->add_componentRect(objH);
layerH->componentRect(geneI);
geneI.next();
printf("The empId value is %i.\n", geneI->empId);
```
Versioning Associations

The versioning features of Objectivity/DB are implemented through the use of association links and member functions automatically defined by the DDL processor. The object reference and handle interface to versioning shields the application developer from knowledge of these associations, however it also enforces particular versioning semantics. By directly manipulating and managing these associations you can:

◆ Create customized versioning semantics
◆ Use customized genealogy object classes

The Objectivity/DB-defined association links to support versioning are confined to the classes `ooObj` and `ooGeneObj`, and can be grouped into two categories of those that support:

◆ Previous/next and derivatives/derived from
◆ Default version of a genealogy

The association links that support the previous, next, derivatives, and derived-from versions may be modeled as shown in Figure 16-5.

![Association Links Supporting Previous, Next, Derivatives, Derived](image_url)
The association links that support the default version may be modeled as shown in Figure 16-6:

![Figure 16-6 Association Links Supporting Default Version](image_url)

The actual DDL declarations for these association links are as follows:

**Basic objects** (ooObj and user-defined derived classes):
- ooHandle(ooObj) nextVers[ ] <-> prevVers;
- ooHandle(ooObj) prevVers <-> nextVers[ ];
- ooHandle(ooGeneObj) geneObj <-> allVers[ ] : version(copy);
- ooHandle(ooGeneObj) defaultToGeneObj <-> defaultVers;
- ooHandle(ooObj) derivatives[ ] <-> derivedFrom[ ];
- ooHandle(ooObj) derivedFrom[ ] <-> derivatives[ ];

**Genealogy Object** (ooGeneObj and user-defined subclasses):
- ooHandle(ooObj) allVers[ ] <-> geneObj;
- ooHandle(ooObj) defaultVers <-> defaultToGeneObj;
Except for `nextVers` and `prevVers` (which are automatically managed by the version creation mechanism), the precise use of these association links is up to you, depending upon the versioning semantics desired. The following is a list of the paired association links, each with a brief discussion on how they might be used.

- `ooObj::geneObj` and `ooGeneObj::allVers[]`
  Used to associate all versions to the genealogy object. Allows bidirectional access between all versions and the genealogy object.

- `ooObj::defaultToGeneObj` and `ooGeneObj::defaultVers`
  Used to associate the default version to the genealogy object. Allows access to the default from the genealogy object.

- `ooObj::derivatives[]` and `ooObj::derivedFrom[]`
  Used to associate a version with all other versions that are derived from it. This differs from previous and next in the sense that a next version is actually created from the current version, while a derivative generally indicates that a version branch is being merged.

Figure 16-7 shows an example of how these association links might be used within a simple version genealogy.
Figure 16-7  Versioning Associations and the Genealogy Object
The following is a complete list of member functions generated automatically by the DDL processor to support the association links. See the "Base Classes" appendix for details on each member function.

Association versioning member functions for basic objects (ooObj and user-defined derived classes):

- `ooObj::add_derivatives`
- `ooObj::add_derivedFrom`
- `ooObj::add_nextVers`
- `ooObj::defaultToGeneObj`
- `ooObj::del_defaultToGeneObj`
- `ooObj::del_derivatives`
- `ooObj::del_derivedFrom`
- `ooObj::del_geneObj`
- `ooObj::del_nextVers`
- `ooObj::del_prevVers`
- `ooObj::derivatives`
- `ooObj::derivedFrom`
- `ooObj::exist_defaultToGeneObj`
- `ooObj::exist_derivatives`
- `ooObj::exist_derivedFrom`
- `ooObj::exist_geneObj`
- `ooObj::exist_nextVers`
- `ooObj::exist_prevVers`
- `ooObj::geneObj`
- `ooObj::nextVers`
- `ooObj::ooNewVersInit`
- `ooObj::prevVers`
- `ooObj::set_defaultToGeneObj`
- `ooObj::set_geneObj`
Association versioning member functions for genealogy object (ooGeneObj and user-defined derived classes):

- ooGeneObj::allVers
- ooGeneObj::add_allVers
- ooGeneObj::defaultVers
- ooGeneObj::del_allVers
- ooGeneObj::del_defaultVers
- ooGeneObj::exist_allVers
- ooGeneObj::exist_defaultVers
- ooGeneObj::set_defaultVers
- ooGeneObj::sub_allVers

The remainder of this section on advanced versioning topics discusses how to use a select few of these association member functions to achieve a particular effect. However, you are free to use any of the above to achieve the desired versioning semantics.
Traversing the Version Genealogy

Finding the Next Version

To obtain an object reference or handle to the next version of an object, use the following member function, which is defined on all versionable persistent objects:

```c++
    ooStatus ooObj::nextVers(
        ooItr(className)& itr,
        const ooMode openMode = oocNoOpen) const;
```

```c++
    ooStatus ooObj::nextVers(
        ooItr(className)& itr,
        const char* predicate,
        const ooAccessMode access = oocPublic,
        const ooMode openMode = oocNoOpen) const;
```

where

- `className` Class of the object that is versioned
- `itr` Iterator initialized to traverse all versions derived from this object
- `openMode` Mode in which objects of the iteration are automatically opened. Default is `oocNoOpen`; that is, they are not opened.

Initializes the iterator `itr` with the object reference or handle of all versions derived from an object. For linear branching this `itr` will return zero or one object reference or handle. For branch versioning `itr` will return zero or more object references or handles. Objects of the iteration are automatically opened in mode `openMode`. 
Example

This example initializes the iterator nextI to traverse all versions derived from the object referenced by objH. Note that we must use the -> operator to access the nextVers member function because it is defined on the object, not the handle.

```cpp
ooHandle(Rect) objH; // class Rect is a subclass of ooObj
ooItr(Rect) nextI;
...
// objH is set to a valid handle value
...
objH->nextVers(nextI);
while (nextI.next())
{
    ((ooHandle(Rect)&) nextI)->getName();
}
```

Finding the Previous Version

To obtain an object reference or handle to the previous version of an object, you use the following member functions, defined on all versionable persistent objects:

- `ooRef(ooObj)& ooObj::prevVers(ooRef(ooObj)& objId, const ooMode openmode = oocNoOpen) const;`
- `ooHandle(ooObj)& ooObj::prevVers(ooHandle(ooObj)& objH, const ooMode openmode = oocNoOpen) const;`
- `ooHandle(ooObj) ooObj::prevVers(const ooMode openmode = oocNoOpen) const;`

where

- `objId` Object reference set to the previous version of this object
- `objH` Handle set to the previous version of this object
openmode

Access mode in which to automatically open the previous version. May be any one of the constants oocRead, oocUpdate, or oocNoOpen. Default value is oocNoOpen, that is the object is not automatically opened.

If no previous version exists, a null object reference or handle is returned.

Example

This example sets the handle prevH to reference the previous version of the object referenced by handle objH. It also tests to see whether or not a previous version actually does exist. We must use the -> operator to access the prevVers member function because it is defined on the object, not the handle.

```c++
ooHandle(Rect) objH, prevH; // class Rect is a subclass of ooObj
...
// objH is set to a valid handle value
...
objH->prevVers(prevH);
if (prevH == 0) {
    printf("No previous version exists\n");
} else {
    printf("There is a previous version\n");
}
```
The Default Version

Objectivity/DB allows you to specify a version of the object as the default version. The semantics of the default version are up to you, but an example might be to indicate the latest useful version, which you want users to access.

Setting Up a Default Version

If you wish to have a default version for a version genealogy, do the following:

1. Create a genealogy object.
   - Do this by using the `new` operator to create an object of class `ooGeneObj` or a user-defined subclass of `ooGeneObj`.

2. Associate the default version to the genealogy object.
   - Do this by invoking the `set_defaultVers` member function on the genealogy object and passing as its argument an object reference or handle to the default version. You may also achieve the same effect by invoking the `set_defaultToGeneObj` member function on the default version.
   - See the “Base Classes” appendix for details on these member functions.

3. Associate versions to the genealogy object.
   - Do this by repeatedly invoking the `add_allVers` member function on the genealogy object, each time passing as its argument an object reference or handle to one of the versions. From this point on, this association is automatically made for new versions when they are created. You may also achieve the same effect by invoking the `set_geneObj` member function on each version.
   - If you do not wish to allow access to the default version from a particular version, then do not perform this step for that particular version.
   - See the “Base Classes” appendix for details on these member functions.
Example

Assume the situation shown in Figure 16-7, where there are four versions (V1, V2a, V2b, V3) in the version genealogy and we wish to indicate that version V2b is the default. In this case the object is of class Rect.

This example shows the code needed to set this up, and assumes that the function getHandles returns handles to all of the versions (v1H, v2aH, v2bH, and v3aH). Obtaining these handles may not be trivial, and in reality will probably require you to fully traverse the version genealogy to obtain them. See “Traversing the Version Genealogy” on page 16-26 for information on version genealogy traversal.

```cpp
ooHandle(ooContObj) contH; // handle of container for all objects
ooHandle(Rect) v1H, v2aH, v2bH, v3H; // Rect is a subclass of ooObj
ooHandle(ooGeneObj) gH; // genealogy object handle
getHandles(v1H, v2aH, v2bH, v3H, contH); // initialize handles
// create the genealogy object in same container
gH = new(contH) ooGeneObj();
// associate default to the genealogy
gh->set_defaultVers(v2bH);
// associate all versions to the genealogy object
gh->add_allVers(v1H);
gH->add_allVers(v2aH);
gH->add_allVers(v2bH);
gH->add_allVers(v3H);
```
Locating the Default Version

You can locate the default version from any version within a given version genealogy or from the genealogy object.

To obtain an object reference or handle to the default version if you have an object reference or handle to the genealogy object, you invoke the `defaultVers` member function on genealogy object.

This member function returns the object reference or handle of default version. See "Setting Up a Default Version" on page 16-29 for details.

You may need to do explicit typecasting when you use versioning member functions. For example:

```c++
ooHandle(ooGeneObj) geneH;
ooHandle(Computer) computerH;// Computer is a persistent class
...
// need to do a typecast
computerH = (ooHandle(Computer)&) geneH->defaultVers();
```

To obtain an object reference or handle to the default version if you do not have an object reference or handle to the genealogy object:

1. Obtain the object reference or handle of any version within the genealogy.
2. Check to see if an association exists on the default version link. If so, then this is the default version.
   You do this by first invoking the `defaultToGeneObj` member function on the object and then checking if the returned object reference or handle is null. If it is not null, then by convention the object is the default version.
3. Obtain the object reference or handle of the genealogy object.
   You do this by invoking the `geneObj` member function on the object, which returns the genealogy object reference or handle.
4. Invoke the `defaultVers` member function on the genealogy object.
   This member function returns the object reference or handle of default version.
Example

Assume the situation shown in Figure 16-7 where there are four versions (V1, V2a, V2b, V3) in the version genealogy and version V2b is the default. In this case the object is of class Rect.

This example is a function `getDefault`, which passes back the handle of the default version in the parameter `defH`.

```cpp
void getDefault(ooHandle(Rect)& objH, ooHandle(Rect)& defH) {
    ooHandle(ooGeneObj) geneH;
    // check to see if objH references the default version
    if (!objH->defaultToGeneObj() == 0) {
        defH = objH;
        return();
    }
    // locate the genealogy object
    geneH = objH->geneObj();
    // get the default handle
    defH = geneH->defaultVers();
    return;
}
```
Changing the Default Version

Once you have initially set up a default version, you may at some time wish to change which version is the default. To do this you:

1. Obtain an object reference or handle to the genealogy object.
   You do this by invoking the `geneObj` member function on any version of the object.

2. Remove the association between the old default version and the genealogy object.
   You do this by invoking the `del_defaultVers` member function on the genealogy object.

3. Associate the new default version to the genealogy object.
   You do this by invoking the `set_defaultVers` member function on the genealogy object and passing as its argument the object reference or handle of the new default version.

Example

Assume the situation shown in Figure 16-7, where there are four versions (V1, V2a, V2b, V3) in the version genealogy and version V2b is the default. In this case the object is of class `Rect`. Now assume that we wish to make version V3 the default. Thus we want the default version associations to look like Figure 16-8.

The following function `setDefault` should do this when passed the handle of version V3.

```c
void setDefault(ooHandle(Rect) & objH)
{
    ooHandle(ooGeneObj) geneH;    // genealogy object handle
    ooHandle(Rect) oldVersH;      // old default version handle

    // locate the genealogy object
    geneH = objH->geneObj();

    // remove necessary associations
    geneH->del_defaultVers();

    // associate new default version
    geneH->set_defaultVers(objH);
}
```
// set up new associations
geneH->set_defaultVers(objH);
return;
}

Figure 16-8  Four Versions with Version V3 the Default Version
Removing Support for the Default Version

If you no longer wish to maintain a default version you are not required to do anything. However, if another application makes use of the existing default version associations, it may assume there is a default even though you no longer wish to indicate such. Thus it is a good idea to remove the association between the genealogy object and the default version when you no longer need it.

To remove support for the default version:

1. Obtain an object reference or handle to the genealogy object for the version genealogy.
   You may accomplish this by invoking the geneObj member function on any version other than the default.

2. Remove the genealogy object for the version genealogy.
   You do this by using the ooDelete macro, using the object reference or handle of the genealogy object as its parameter. Deleting the genealogy object automatically removes all the default version associations.

Example

Assume the situation shown in Figure 16-7 where there are four versions (V1, V2a, V2b, V3) in the version genealogy and version V2b is the default. In this case the object is of class Rect.

This example obtains the handle of the genealogy object and removes it. It assumes a user-defined function, getHandles, exists and obtains the version handles.

```c
ooHandle(Rect) v1H, v2aH, v2bH, v3H; // Rect is ooObj subclass
ooHandle(ooGeneObj) gH; // genealogy object handle
...
getHandles(v1H, v2aH, v2bH, v3H); // initialize handles
// get the genealogy handle
gH = v1H->geneObj();
// remove the genealogy object
ooDelete(gH);
```
Using the Genealogy Object

Once you have set up the versioning association to support a default object (see “Setting Up a Default Version” on page 16-29), you may use the genealogy object as a multiplexer over the entire genealogy. This allows other persistent objects to associate to a single object from which they may access any version of the genealogy.

Locating the Default Version Using the Genealogy Object

To locate the default version from a persistent object associated to the genealogy:

1. Obtain an object reference or handle to the genealogy object.
   You do this by invoking the `nameOfLink` member function on the associated object, where `nameOfLink` is the name of the association link to the genealogy object class.
2. Obtain an object reference or handle to the default version.
   You do this by invoking the `defaultVers` member function on the genealogy object.

Example

Going back to the example in “Creating a Customized Genealogy Object” on page 16-18, assume that we have a genealogy consisting of instances of class `Rect` and an instance of class `Layer`, which is associated to the genealogy object. The situation can be represented graphically using association instance notation, as shown in Figure 16-9. Although the associations to support the previous and next versions do exist, they are not shown in the figure because they are not important to this discussion.
Figure 16-9  Instances of Classes Rect and Layer

Example

The following code segment obtains a handle to the default version given a handle to the Layer object which is associated to the genealogy.

```cpp
ooHandle(RectGeneObj) objH;  // genealogy object handle
ooHandle(Layer) layerH;       // layer object handle
ooHandle(Rect) defaultRectH;  // default version handle
ooItr(RectGeneObj) geneI;    // iterator for genealogy
...
// layerH is set to a handle of object associated to genealogy
...
// obtain handle to genealogy object
layerH->componentRect(geneI);
geneI.next();
```

Versioning  16-37
Locating All Versions Using the Genealogy Object

To locate all versions of a persistent object associated to the genealogy:

1. Obtain an object reference or handle to the genealogy object.
   You do this by invoking the `nameOfLink` member function on the associated object, where `nameOfLink` is the name of the association link to the genealogy object class.

2. Obtain object references or handles to the versions.
   You do this by invoking the `allVers` member function on the genealogy object.

Example

Assume the same situation as the previous example. The following code segment initializes the iterator `versI` to return the handles of all versions of the genealogy.

```c++
ooHandle(RectGeneObj) objH;
ooHandle(Layer) layerH;
ooItr(ooObj) versI;
ooItr(RectGeneObj) geneI;
...
// layerH is set to handle of object associated to genealogy
...
// obtain handle to genealogy object
layerH->componentRect(geneI);
geneI.next();
// obtain handles to versions
geneI->allVers(versI, oocRead);
```
Merging a Version

Another option available when managing a version genealogy is to merge version branches. This allows you to merge one version branch into another, reducing the number of branches in the version tree. Figure 16-10 illustrates the concept of merging versions.

The version that serves as the merging point is said to be derived from the last version of the merged branch. Thus, in the example of Figure 16-10, V5a is derived from V4b, and V4b is derived from V3c. Note that V4a is still the previous version of V5a since V5a was created from V4a. Similarly V3b is the previous version of V4b.

![Figure 16-10  Merging Versions](image)

At some point you may wish to merge a version branch into another. Merging a version branch usually indicates that the last version within the branch has somehow contributed to the version designated as its successor; thus, the successor is said to be derived from the merged object. In a similar manner the successor object is said to be a derivative of the merged object.

Merging a version in Objectivity/DB requires that you manage the ooObj::derivatives and ooObj::derivedFrom association links properly.

To merge a version you indicate a successor to the last version of the branch. You do this either by invoking the add_derivatives member function on the last version of the branch to be merged, or by invoking the add_derivedFrom member function on the successor object.
To obtain the object references or handles of the versions from which a version was derived, or its derivative versions, you invoke the proper version association member function on the version.

To find the derivatives of a version, you invoke the `derivatives` member function on the version. To find the versions from which it was derived, you invoke the `derivedFrom` member function on the version.

---

**Example**

Assume we have the following genealogy of objects of class `Rect`, and we wish to merge version `V2b` back into `V3`.

![Figure 16-11  Merging V2b Back into V3](image)

The following code segment sets the derived from/derivatives association between `V2b` and `V3`. It also obtains the handle of `V3` from `V2b` via the `derivatives` member function, and the handle of `V2b` from `V3` via the `derivedFrom` member function.

```cpp
ooHandle(Rect) v2bH, v3H;
ooItr(Rect) V2bDerivatives, V3DerivedFrom;
...
// v2bH and v3H are set to reference the appropriate versions
...
// set the derived from/derivatives association
v2bH->add_derivatives(v3H);
```
// get derivatives of V2b
v2bH->derivatives(v2bDerivatives, oocRead);

// get versions V3 was derived from
v3H->derivedFrom(v3DerivedFrom, oocRead);

----------

### Naming Versions

You may choose to give each version of a genealogy a name, using the facilities provided by Objectivity/DB. For details on naming persistent objects, see “Using System Names” on page 8-1. In general, it makes sense to give each version a meaningful name within the scope of the genealogy object. By doing so, an application may access a particular version directly, once it has an object reference or handle to the genealogy object, by using the `lookupObj` member function.

----------

#### Example

Assume the genealogy used in the example for the section “Using the Genealogy Object” on page 16-36. Also assume each version was given the indicated name (V1, V2a, V2b, and V3) in the scope of the genealogy object. The following code segment obtains a handle to the version named V2b given the handle of a `Layer` object, that is associated to the genealogy object.

```cpp
ooHandle(Layer) layerH;
ooHandle(Rect) targetVersH;
ooItr(RectGeneObj) geneI;
...
// layerH is set to handle of object associated to genealogy
...
// obtain handle to genealogy object
layerH->componentRect(geneI);
geneI.next();
// obtain handle of object named V2b
geneI.lookupObj(geneI, "V2b", oocRead);
```

----------

Advanced Versioning Techniques
Altering the Version Genealogy

The precise versioning semantics for your application are really up to you. Objectivity/DB provides the basic mechanism to implement many different versioning schemes. This section discusses some options available to you and the issues involved with each.

Removing a Version

You might decide at some point to remove a version from a version genealogy. To do this, you use the `ooDelete` macro, using the object reference or handle of the version to be removed as its single argument.

Be aware that this will actually cause a break to appear in the version genealogy, where versions derived from the removed version will no longer have a previous version.

Example

Assume the situation shown in Figure 16-7, where there are four versions (V1, V2a, V2b, V3) in the version genealogy, and version V2b is the default. In this case the object is of class Rect.

Now assume that we remove version V2a, using the `ooDelete` macro, using an object reference or handle to this version. The resulting versioning associations are shown in Figure 16-12.

Note that version V3 is still a member of the versioning genealogy, but it can no longer be accessed as the next version of any other version, nor can any other version be accessed from V3 as being a previous version.
Changing Previous and Next Versions

You may also manually alter the order of previous and next versions within a version genealogy by manipulating the `prevVers` and `nextVers` associations.

Example

Assume the situation of the last example, where version V2a has been removed. Now assume that we wish to indicate that V3 is the next version of V1. This is known as collapsing the version genealogy. To do this we must set the versioning associations as shown in Figure 16-13.

The following code segment demonstrates how to fix up the versioning associations to achieve this effect. It assumes a user-defined function `getHandles` exists and obtains the version handles.

```cpp
ooHandle(Rect) v1H, v2bH, v3H; // Rect is ooObj subclass
...
getHandles(v1H, v2bH, v3aH); // initialize handles
```

**Figure 16-12  Versioning Associations after Removing V2a**
// fix up the associations
v3H->set_prevVers(v1H);

Figure 16-13  Versioning Associations for Collapsing the Version Genealogy
Using Debug Mode

Debug mode provides debugging support by making it easier for you to locate software bugs in Objectivity/DB applications.

You can use debug mode to detect common errors made using Objectivity/DB. For example, it is not unusual in early development to erroneously update an object opened for read-only, or to write past the boundary of an object. You may also want to use the verification option to detect data corruption caused either by a bug in the application or in Objectivity/DB. This is particularly useful in the early phase of a product, when the software is relatively unstable. Debug mode helps you detect problems early and prevent corrupt data from being written out to the federated database.

Debug mode consists of:

- Data verification at basic object, page, and container levels
- Event tracing for various Objectivity/DB operations

Currently, debug mode does not support verification of large objects (objects larger than a page).

There will be performance degradation when using the various debugging options since the database software is doing extra verification and tracing. However, this should not be an issue since the debugging options are not expected to be turned on when an application is used in a production environment.

Activating Debug Mode

To use debug mode, you must link your application with the debug version of the Objectivity/DB library (for details, see the Installation and Platform Notes for your platform). Once this is done, you can turn the debugging options on and off without recompiling or relinking your application. To select debugging options, set
environment variables for the options you want to use. On most platforms, this is done using the `setenv` command as is shown in the following example:

---

**Example**

```bash
setenv OO_DEBUG_VERIFY_OBJECT
setenv OO_DEBUG_FILE "~/debug.trace"
```

---

### Debug Options

Debugging options supported by debug mode are described in the following sections.

#### Data Verification

Debug mode provides data verification at the basic object, page, and container level, so you can detect problems earlier and generate trace information, such as the before-image and after-image of a page. This helps you track down the cause of the problem. Verification also prevents corrupt data from being written out to disk, thus avoiding permanent damage to the federated database.

#### Basic Object Verification

If you activate basic object verification, Objectivity/DB maintains the before-image of objects read from the disk and verifies that read-only objects have not been modified at the end of the transaction. This allows detection of writing-on-an-object-opened-for-read bugs common in many applications. It also helps find writing-past-an-object bugs that destroy adjacent objects or system-maintained information. Random writes by wild pointers may also be detected if they modify objects that have not been opened for update.

When a problem is detected, the before-image and the modified page containing the basic object are written to a trace file. To activate basic object verification, use the `OO_DEBUG_VERIFY_OBJECT` option.

- `OO_DEBUG_VERIFY_OBJECT`
Page Verification

If you activate page verification, Objectivity/DB verifies the consistency of each page after reading it from disk and before writing it back. It checks each page after reading and reports any problems before the application or Objectivity/DB has a chance to operate on the (inconsistent) data. Checking a page before writing it prevents corruption of the federated database. To activate page verification, use the OO_DEBUG_VERIFY_PAGE option.

- OO_DEBUG_VERIFY_PAGE

The following are examples of the checks performed:

- free space + space used == page size
- offsets of basic objects lie within the page
- types and sizes of basic objects are valid
- basic objects do not overlap each other
- page header information, such as container ID and logical page number, are valid

Container Verification

If you activate container verification, Objectivity/DB will verify the page allocation information for a container when it is opened and closed. This helps detect internal Objectivity/DB bugs, such as allocating the same physical page more than once. To activate container verification, use the OO_DEBUG_VERIFY_CONTAINER option.

- OO_DEBUG_VERIFY_CONTAINER

Event Trace

When you are debugging a problem, it is useful to know the series of events that have occurred. For example, if you experience a problem in which Objectivity/DB returns the object-does-not-exist error and you are able to reproduce the problem, you can turn on event tracing to log all object operations. You can then determine if the object has been deleted (a bug in the application), or if it has somehow disappeared (possible Objectivity/DB bug). Generating a trace of the events can help you narrow down the cause of the problem.
Basic Object Events

You can use the `OO_DEBUG_TRACE_OBJECT` option to trace basic object operations such as create, open, close, delete, and resize.

- `OO_DEBUG_TRACE_OBJECT`

Page Events

You can use the `OO_DEBUG_TRACE_PAGE` option to trace page operations such as page read, write, and allocate.

- `OO_DEBUG_TRACE_PAGE`

Container Events

You can use the `OO_DEBUG_TRACE_CONTAINER` option to trace container operations such as create, open, close, and delete.

- `OO_DEBUG_TRACE_CONTAINER`

Database Events

You can use the `OO_DEBUG_TRACE_DATABASE` option to trace database operations such as create, open, close, and delete.

- `OO_DEBUG_TRACE_DATABASE`

Trace Output File

By default, debug traces are directed to the standard error. If you set the `OO_DEBUG_FILE` option, debug traces are directed to a specified file.

- `OO_DEBUG_FILE debugFile`

where

- `debugFile` is the file for debug traces.
Signal Handling

This chapter describes how to work with system signals.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`.

System Signal Handling Issues

Objectivity/DB installs signal handlers to perform any necessary cleanup when Objectivity/DB applications receive signals on UNIX systems. The following signals are currently caught: `SIGINT`, `SIGQUIT`, `SIGILL`, `SIGABRT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGHUP`, `SIGPIPE`, `SIGTERM`, `SIGEMT`, and `SIGTRAP`.

Objectivity/DB’s signal handlers restore the previously installed handlers and reissue the signal received upon performing their cleanup activities. Because of this behavior, if you wish to install your own signal handlers you should do one of the following:

- Install your signal handlers before calling `ooInit`, or
- Make sure you chain any signal handlers installed after calling `ooInit`. Chaining refers to saving the current signal handlers before installing new ones, then restoring the saved signal handlers from within the newly installed ones and re-issuing the signal received.

If the application installs its own signal handler before calling `ooInit`, the Objectivity/DB signal handler is called first, followed by the application’s signal handler.

If the application installs its own signal handler after calling `ooInit`, only the application’s signal handler is called. If the application wants to use Objectivity/DB signal handler, it needs to be called explicitly from within the application’s signal handler.
If an application wishes to ignore signals, the calls to `signal` and `sigvec` should be made after calling `ooInit`. No cleanup will be performed by Objectivity/DB.

If you do not want the Objectivity/DB signal handlers to be invoked, you can prevent them from being installed during system initialization by calling `ooInit` with the `installSigHandler` argument set to `oocFalse`.

SIGKILL cannot be caught. Consequently, no cleanup is performed by Objectivity/DB upon receiving SIGKILL.

**Signal Handling Example**

The following example sets up a signal handler for some of the signals caught by Objectivity/DB. Since the user-defined signal handler, `mySignalHandler`, is installed after the federated database is opened, it gets called instead of the default Objectivity/DB signal handler. Within `mySignalHandler`, a message is printed and then the Objectivity/DB signal handler is reinstated using:

`signal(the_sig, oldFunc)`

where

`oldFunc` Previous signal handler for that signal

Then the signal is regenerated so that the previous signal handler gets called.

```c
#include <iostream.h>
#include <signal.h>
#include <unistd.h>     // for getpid()
#include "objects.h"

void mySignalHandler(int the_sig);
void setup_signals();

//old signal function pointers
void (*oldhup)(int);  //old function for SIGHUP
void (*oldup)(int);   //old function for SIGINT
void (*oldquit)(int); //old function for SIGQUIT
void (*oldill)(int);  //old function for SIGILL
void (*oldabrt)(int); //old function for SIGABRT
```
void (*oldfpe)(int);   //old function for SIGFPE
void (*oldbus)(int);   //old function for SIGBUS
void (*oldsegv)(int);  //old function for SIGSEGV
void (*oldpipe)(int);  //old function for SIGPIPE
void (*oldterm)(int);  //old function for SIGTERM

// User-defined signal handler mySignalHandler
void mySignalHandler(int the_sig)
{
    switch (the_sig)
    {
    case SIGHUP:
        cout << "The SIGHUP signal received! " << the_sig << endl;
        signal(the_sig, oldhup);   //restore previous function
        break;
    case SIGINT:
        cout << "The SIGINT signal received! " << the_sig << endl;
        signal(the_sig, oldint);   //restore previous function
        break;
    case SIGQUIT:
        cout << "The SIGQUIT signal received. " << the_sig << endl;
        signal(the_sig, oldquit);   //restore previous function
        break;
    case SIGILL:
        cout << "The SIGILL signal received. " << the_sig << endl;
        signal(the_sig, oldill);   //restore previous function
        break;
    case SIGABRT:
        cout << "The SIGABRT signal received. " << the_sig << endl;
        signal(the_sig, oldabrt);   //restore previous function
        break;
    case SIGFPE:
        cout << "The SIGFPE signal received. " << the_sig << endl;
        signal(the_sig, oldfpe);   //restore previous function
        break;
    case SIGBUS:
cout<<" The SIGBUS signal received. "<<the_sig<<endl;
signal(the_sig,oldbus); //restore previous function
break;
case SIGSEGV:
  cout<<" The SIGSEGV signal received. "<<the_sig<<endl;
  signal(the_sig,oldsegv); //restore previous function
  break;
case SIGPIPE:
  cout<<" The SIGPIPE signal received. "<<the_sig<<endl;
  signal(the_sig,oldpipe); //restore previous function
  break;
case SIGTERM:
  cout<<" The SIGTERM signal received. "<<the_sig<<endl;
  signal(the_sig,oldterm); //restore previous function
  break;
default:
  cout<<" An unknown signal received. "<<the_sig<<endl;
  signal(the_sig,SIG_DFL); //restore previous function
  break;
}
sigsetmask(0); // restore the default mask

cout<<"Old signal handler restored."<<endl;

int pid;
pid = getpid(); //get the process ID
cout<<"The process ID is "<<pid<<endl;

cout<<"Re-sending the signal now!"<<endl;

kill(pid,the_sig); //regenerate the signal

// setup_signals
void setup_signals()
```c
{
    // set signal handler to be mySignalHandler and save pointer
    // to the previous signal handler for the signal
    oldhup = signal(SIGHUP, mySignalHandler); // hangup
    oldint = signal(SIGINT, mySignalHandler); // reg new
        // handler, Ctrl-C
    oldquit = signal(SIGQUIT, mySignalHandler); // Ctrl-\n
    oldill = signal(SIGILL, mySignalHandler); // illegal
        // instruction
    oldabrt = signal(SIGABRT, mySignalHandler); // abort
    oldfpe = signal(SIGFPE, mySignalHandler); // arithmetic
        // expression
    oldbus = signal(SIGBUS, mySignalHandler); // bus error
    oldsegv = signal(SIGSEGV, mySignalHandler); // segmentation
        // violation
    oldpipe = signal(SIGPIPE, mySignalHandler); // Write on pipe
        // with noone to
        // read
    oldterm = signal(SIGTERM, mySignalHandler); // software
        // termination
        // signal
}

// main program
int main()
```
{ 

  ooTrans trans;
  ooHandle(ooFDObj) fddb;
  ooHandle(ooDBObj) db;
  ooHandle(ObjectA) objectA;

  ooInit();
  trans.start(oocMROW, 10);

  fddb.open("EXAMPLE", oocUpdate);

  if (!db.exist(fddb, "TestA"))
  {
    db = new ooDBObj("TestA");
  }
  else
  {
    db.open(fddb, "TestA", oocUpdate);
  }

  cout<<"After creating TestA database"<<endl;

  objectA = new(db) ObjectA(0); // create new object in
                               // TestA DB's default
                               // container

  cout<<"After adding objects to the default container
       in TestA"<<endl;

  // trap on all the signals handled by Objectivity/DB

  setup_signals();

  cout<<"Enter a char when you want to continue:"<<endl;
  char answer[10];
cin.getline(answer, sizeof(answer)-1);

cout<<"bye! this is the end of the program"<<endl;

return 0;
}
Error Handling

This chapter describes how to use the Objectivity/DB error handling facility. Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`.

Using the Error Handling Facility

Objectivity/DB provides a basic error handling facility that your applications interface with and, optionally, may adopt.

The general Objectivity/DB error handling mechanism is as follows:

- A single error handling function called the error handler is registered with Objectivity/DB at all times. Applications are free to register their own error handler at any time, although a default error handler is provided.
- When an error is detected by an application, it raises an error signal, which sets global error flags and invokes the currently registered error handler.
- An error condition is generally made known to a calling function by returning a zero value.
- Upon detecting an error condition from a called function, a calling function may obtain more detailed information about the condition by inspecting the global error flags.
- If your application uses multiple threads through the concurrent transaction feature, note that error handlers are thread-specific. To use error handlers with concurrent transactions, you should register a new error handler for each thread your application uses. For more information about concurrent transactions, see “Concurrent Transaction Support” on page 2-39.

This mechanism is very flexible and allows you to handle almost any type of error condition at the point it is detected or anytime after.
The \texttt{ooStatus} Type

The majority of Objectivity/DB member functions and functions return a status code of type \texttt{ooStatus}, which is defined as compatible with type \texttt{int32}.

By convention, if a function returns a non-zero value, you do not need to take any action. If it returns a zero value, you may wish to take some corrective action. Objectivity/DB defines the constant \texttt{oocSuccess} as a non-zero value and \texttt{oocError} as zero. The value returned by \texttt{ooStatus} for a function is \texttt{oocSuccess} if successful and \texttt{oocError} if unsuccessful. See “Error Flags” on page 19-5 for more information.

Error Identifier Structure

The following data structure is used to store error numbers and messages:

\begin{verbatim}
typedef struct {
    uint32 errorN;     // unique error number
    char *message;     // printf format string for error
} ooError;
\end{verbatim}

where

\begin{itemize}
    \item \texttt{errorN} \hspace{1cm} Unique error number. Error numbers 0 through 999999 are reserved for Objectivity/DB system error codes. Within your application, make sure that you assign modules unique error numbers greater than 999999.
    \item \texttt{message} \hspace{1cm} Identical to a \texttt{printf} format string, allowing specification of a variable number of arguments. These arguments provide the context portion of an error message.
\end{itemize}

\textbf{Example}

Assume that the following error handler code is used for a geometry manager module named \texttt{geo}. This code associates an error number (1000000) with the message “Cannot open file \texttt{filename}. Aborting processing”, and initializes the error identifier variable named \texttt{geoCannotOpen}. The string \texttt{%s} for the file name is filled with the appropriate string at runtime.
Error Message File

You should declare and initialize all error identifier structures in an error message header file, and include this file in your application source file. We recommend that you group error identifiers into files on a per source module basis. Figure 19-1 illustrates how a typical error message file is used.

![Figure 19-1  The Error Message Header File](image)

You should give a symbolic name that begins with the module prefix to each structure in a message file. Within the application source, error messages are referred to by these identifiers.
Example

To create an error message file for the geo geometry manager, you must do the following:
◆ Decide on a symbolic name prefix.
◆ Decide on a range of error numbers for this module.

If you use the prefix geo and the error number range from 1000000 to 1000100, the resulting error message file might look like this:

```c
ooError geoCannotOpen  = { 1000000, "Cannot open file %s.
                        Aborting processing" };  
ooError geoNoSuchFile  = { 1000001, "No such file - %s"  };  
ooError geoNoSuchDir   = { 1000002, "No such directory - %s" };  
ooError geoIOError     = { 1000003, "System I/O Error"   };  
ooError geoRatioTooBig = { 1000004, "Ratio %d/%d is too big" };  
```

Error Levels

Errors are assigned a level according to their severity. These error levels are used by the error handler to help determine how to handle a given error condition.

A function that raises the error signal must specify a proposed error level for the error condition. This proposed level may subsequently be overridden by the error handler that is currently registered.

The possible error levels (defined by the system-defined type `ooErrorLevel`) are:

- **Warning** (`ooWarning`): An abnormal event has occurred. No action should be taken beyond notifying the user of the condition.
- **Nonfatal User Error** (`ooUserError`): A non-fatal user error is an error detected by the programming interface that can be directly attributable to the application programmer. Inconsistent user data passed to the...
programming interface, for example, would typically trigger a non-fatal user error.

Nonfatal System Error (oocSystemError)
A non-fatal system error is an error detected by Objectivity/DB during its operations. These errors might actually have been caused by user errors that slipped through consistency checks in the programming interface. Without in depth analysis, it is often difficult to tell whether a non-fatal system error is attributable to the user or Objectivity/DB.

Fatal Internal Error (oocFatalError)
A fatal internal error is the most severe type of error that can occur in the system. This is signalled when Objectivity/DB detects an unrecoverable internal inconsistency that might have already caused data corruption. When a fatal internal error is signalled, the only thing an application should do is abort the transaction and shut down as quickly as possible.

Error Flags
The following global variables are defined by Objectivity/DB:

- ooErrorLevel oovLastErrorLevel;
- ooError* oovLastError;
- uint32 oovNError;

where

- oovLastErrorLevel Error level of the last event
- oovLastError Pointer to the error for the last event
- oovNError Count of the total number of errors (not including warnings) that have occurred so far

Your application obtains access to these error flags through the Objectivity/DB header file oo.h, which the DDL Processor automatically includes in the schema source code file.
Your application should never directly modify the error flags except to clear their values. They are actually set indirectly by your application when you signal an error condition. See “Raising the Error Signal” on page 19-7 for more information on how the error flags are set.

By convention the values in Table 19-1 indicate a clear (non-error) state:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Non-Error Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>oovLastErrorLevel</td>
<td>oocNoError</td>
</tr>
<tr>
<td>oovLastError</td>
<td>zero</td>
</tr>
</tbody>
</table>

You can clear these variables by hand, or you can call the ooResetError function, which clears them for you. The syntax of this function is as follows:

```c
void ooResetError();
```

### Example

The following code shows how a C++ application might use the error flags to handle an error from function `calcRect`. Because `calcRect` returns a code of type `ooStatus`, it may be assumed that if zero was returned, then an error has been signaled and the error flags are set.

```c
ooStatus calcRect(void); // function prototype for calcRect
ooStatus rc; // local variable for return code
static int retryCount = 0;
...
if ((rc = calcRect()) == 0) {
    // if the error is not gmneRetry, return the return code
    if (oovLastError->errorN != gmneRetry.errorN)
        return rc;
    // if the error is gmneRetry, clear the Objectivity/DB error flags, increment a counter and return oocSuccess
```
Signaling an Error

When your application detects an error, it should take the following actions to indicate the error condition exists:

1. Raise the error signal by using the ooSignal function.
2. Indicate whether or not an error condition exists to the calling function. By convention, the constant oocSuccess indicates that no error occurred, and any other value indicates an error condition.

Raising the Error Signal

You may raise the error signal by calling the following Objectivity/DB function:

```c
ooStatus ooSignal (const ooErrorLevel errorLevel,
                   const ooError& errorID,
                   const ooHandle(ooObj)* contextObj...);
```

where

- `errorLevel` Proposed error level for this event.
- `errorID` Symbolic name of the error identifier. These are defined in the error message file.
- `contextObj` Pointer to the handle of an object that may help the system error handler to pinpoint the context of the error. If not used, this parameter should be set to zero.
- ... Variable number of arguments required by the format string stored in the message part of the error identifier. These arguments provide context data that are printed as part of the error message.

These functions set the system-defined global variables, invoke the most recently registered error handler, and return the result from the error handler. The variable oovLastErrorLevel is set to `errorLevel`, oovLastError is set to the last error number and error message, and oovNError is incremented.
Example

The following code shows how a C++ application might signal an error condition. In this case the error condition is file open failure.

```c
FILE* fptr; // file pointer
char fileName[80]; // name of the file to open
...
fptr = fopen(fileName, "w");
if (fptr == 0) {
    return(ooSignal(oocUserError, geoCannotOpen, 0, fileName));
...}
```

This example uses the error identifier `geoCannotOpen`, defined in the `ooerror` example on page 19-2. This error identifier expects a single argument, the name of the file that could not be opened. Thus `fileName` is passed as the last parameter to `ooSignal`.

Informing the Calling Function

You indicate an error condition to the calling function by returning a zero value. By convention, a non-zero return value means that the function completed error-free. A return zero value means that an error has occurred.

In most cases your application should use a return code of type `ooStatus`, using a non-zero value to indicate that no error occurred or zero to indicate an error condition.
Detecting an Error Condition

Checking the Function Return Code

By convention, if a function with type ooStatus returns a zero value, an error has occurred. It is up to your code to handle the condition in a manner appropriate to your application. See “Error Flags” on page 19-5 for an example.

As an error handler may have already been invoked by a function that returns an error code, some cooperation may be necessary between software layers to establish exactly where a particular error condition will be handled (by the error handler or by some local error handling code.)

Checking the Error Flags

Situations may arise where it is not desirable to indicate an error condition using the function return code. In such cases you may wish to adopt the convention that an error condition exists if any of the Objectivity/DB Error Flags are set to anything but their non-error value. If so, you should make sure that the code that actually handles the error clears these variables.

See “Error Flags” on page 19-5 for information about these variables.

The Error Handler

Role of the Error Handler

The error handler is a function registered with Objectivity/DB that is invoked when an application raises the error signal by calling the ooSignal function. Its purpose is to handle the signaled error condition.

The default error handler provided by Objectivity/DB calls the message handler (see “The Message Handler” on page 19-13) to print an error message and returns a value according to the error level. A non-zero value is returned for a warning message. A zero value is returned for a non-fatal error. A fatal error generates a core dump. You may define a custom error handler for your application and register it with Objectivity/DB at any time.
Writing an Error Handler

You may write and register your own error handler at any time but it must conform to the following calling interface and return a value according to the error level:

```c
ooStatus errHandler(
    const ooErrorLevel errorLevel,
    const ooError& errorID,
    const ooHandle(ooObj)* contextObj,
    char* errorMsg
)
```

where

- `errHandler` Name of your error handler function
- `errorLevel` Proposed error level for this event
- `errorID` Symbolic name of the error identifier. These are defined in the error message file.
- `contextObj` Pointer to the handle of an object that may help the system error handler to pinpoint the context of the error. If not used, this parameter should be set to the null pointer `((type*) 0)`.
- `errorMsg` String created by running `vsnprintf` over the message part of the error identifier and its arguments.
Example

This example shows a typical user-defined C++ error handler, which ignores the error unless it is fatal.

```c++
 ooStatus ignoreErrHandler(
   ooErrorLevel errorLevel,
   ooError& errorID,
   ooHandle(ooObj) *contextObj,
   char *errorMsg) {
   switch(errorLevel) {
     case oocWarning:
       return(oocSuccess);
     case oocUserError:
     case oocSystemError:
       return(0);
     case oocFatalError:
       abort();
     default:
       return(0);
   }
}
```
Registering an Error Handler

You may write your own error handler and register it with the system, replacing
the one last installed. When an error signal is raised, the most recently registered
error handler is invoked.

You register an error handler using the following system-defined function:

\[
\text{ooErrorHandlerPtr ooRegErrorHandler(}
\quad \text{ooErrorHandlerPtr handlerName);}
\]

where

\[
\text{handlerName} \quad \text{Name of the error handler function to register}
\]

This function registers a new error handler and returns a pointer to the previously
registered error handler function.

Note that the interface to your error handler must conform to the specification
shown in “Writing an Error Handler” on page 19-10.

You may also obtain a pointer to the currently registered error handler by using
the following system-defined function:

\[
\text{ooErrorHandlerPtr ooGetErrorHandler();}
\]

Example

In this example, the application saves the old error handler in the variable
\text{oldEHPtr}, registers a new one named \text{myErrorHandler}, performs processing,
and then restores the original error handler.

\[
\begin{align*}
\text{ooErrorHandlerPtr oldEHPtr;} & \quad \text{// pointer to old error handler} \\
\text{...} & \\
\text{oldEHPtr = ooRegErrorHandler(myErrorHandler);} & \quad \text{// save old error handler and register new one} \\
\text{ooRegErrorHandler(oldEHPtr);} & \quad \text{// restore old one} \\
\text{...} & \\
\end{align*}
\]
The Message Handler

The default error handler calls the message handler to actually write the error message to an output device.

While a default message handler is supplied by Objectivity/DB, you can write your own and replace the currently registered message handler.

Your message handler function must conform to the following calling interface:

❑ void MsgHandler(char* message);

where

MsgHandler Name of your message handler function
message String containing message to display

The default message handler prints the error message to stderr.

Example

This example shows a simple C++ message handler named dlgBoxMsgHandler that calls a user-defined graphics function to display the message in a dialogue box.

void dlgBoxMsgHandler(char* message)
{
    // display the message on the graphics terminal in a dialogue box
    winMsgDialog(displayHandle, message);
}

Registering a Message Handler

For applications where you do not wish to direct error and status messages to stderr, you can register your own message handler. For example, this might be desirable in an application running in a graphical environment where status and error information is displayed in dialogue boxes.

You use the following system-defined function to register your own message handler:

❑ ooMsgHandlerPtr ooRegMsgHandler(ooMsgHandlerPtr msgHandler);
where

msgHandler  
Name of the Message Handler function to install

This function registers a message handler and returns a pointer to the previously
registered message handler function. You may also obtain a pointer to the
currently registered message handler by using the following system-defined
function:

- ooMsgHandlerPtr ooGetMsgHandler();

Example

The following code suppresses all error messages printed to stderr while
processing within a particular function. The code saves the old message handler
pointer ooMsgHandlerPtr in the variable oldMHPtr, registers the message
handler dlgBoxMsgHandler (defined in the message handler example on page
19-13), performs processing, and then restores the original message handler.

... 

ooMsgHandlerPtr oldMHPtr;       // pointer to old message
handler
...                              // save old error handler
oldMHPtr = ooRegMsgHandler(dlgBoxMsgHandler);       // register new one
... 

ooRegMsgHandler(oldMHPtr);      // restore old one
...

19-14  Using Objectivity/C++
Using Informational Functions

This chapter describes the member functions, functions, and macros you can use to obtain object information.

Where syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle.

Printing an Object’s OID

You can use the following member function to print a persistent object’s object identifier (OID).

```cpp
void ooRefHandle(ooObj)::print(
    FILE *fp = stdout) const;
```

Locating the Default Container

When you create a new database, Objectivity/DB also creates a default container of class ooDefaultContObj in the new database. If you then create a basic object using the database as the clustering directive, the new basic object is placed in the database’s default container.

To obtain a handle or object reference to the default container, use the following member functions:

```cpp
ooHandle(ooContObj) ooRefHandle(ooDBObj)::getDefaultContObj(
    const ooMode openMode = oocNoOpen) const;
```

```cpp
ooHandle(ooContObj)&
    ooRefHandle(ooDBObj)::getDefaultContObj(
        ooHandle(ooContObj)& contH,
        const ooMode openMode = oocNoOpen) const;
```
Obtaining a Class Type Number

Every persistent class has a system-defined type number associated with it. This type number uniquely identifies the class to Objectivity/DB.

To obtain the type number of a class, use the following macro, which returns the type number of the class specified by `className`:

```cpp
ooTypeNumber ooTypeN(const char* className);
```
where

className

is the name of the desired persistent class.

You can also obtain the type number of a persistent object by using one of the following member functions.

The ooGetTypeN virtual object member function, defined for all basic object and container classes, returns the type number of the object’s class.

❑ virtual ooTypeNumber ooObj::ooGetTypeN() const;

The typeN member function, defined for all handle and object reference classes, returns the class type number of the currently referenced object.

❑ ooTypeNumber ooRefHandle(className) ::typeN() const;

Note

You cannot use the macro ooTypeN as a label in a switch conditional statement because ooTypeN is expanded into a variable name. For example, the following statement causes a compilation error:

```
    ooItr(Foo) fooI;
    ...
    while (fooI.next()) {
        switch(fooI.typeN()) {    
            case ooTypeN(Apple):
                ...
            case ooTypeN(Orange):
                ...
            default:
                ...
        }
    }
```

Instead, you should use an if-else conditional statement, such as:

```
    ooItr(Foo) fooI;
    ooTypeNumber typeN;
    ...
```
Determining an Object's Class

while (fooI.next()) {
    typeN = fooI.typeN();
    if (typeN == ooTypeN(Apple)) {
        ...
    } else if (typeN == ooTypeN(Orange)) {
        ...
    } else {
        ...
    }
}

---

**Determining an Object's Class**

To determine if an object is of a particular class or one of its derived classes, use the following virtual object member function, which is defined by the DDL Processor for all basic object and container classes:

- virtual ooBoolean className::ooIsKindOf(const ooTypeNumber typeN) const;

This member function returns the constant oocTrue if the object belongs to the class with type number typeN or one of its derived classes, and the constant oocFalse otherwise.

You can also obtain a string containing the type name of an object's class using one of the following member functions. They perform the same function, however one is defined on persistent classes while the other is defined on handle and object reference classes:

- The ooGetTypeName virtual object member function is defined by the DDL Processor for all basic object and container classes. It returns a string containing the name of the object's class.
  - virtual char* ooObj::ooGetTypeName() const;

- The typeName member function is defined on all handle and object reference classes. It returns a string containing the class name of the currently referenced object.
  - char* ooRefHandle(className):typeName() const;
Warning

The string returned by `ooGetTypeName` and `typeName` is used by Objectivity/DB internally, so you must not modify the string in any manner. Doing so may result in unexpected program errors.

Example

In this example, the variable `isCellObj` is set to `ooTrue` because the object referenced by `objH` is of class `Layer`.

```c
ooHandle(ooObj) objH;
ooHandle(Cell) cellH; // Cell is a derived class of ooObj
ooBoolean isCellObj;
ooTypeNumber typeNum;
...
// cellH is set to a valid handle
// objH is set to a valid handle
...

isCellObj = objH->ooIsKindOf(ooTypeN(Cell));
if (isCellObj)
    objH = cellH; // Okay because Cell is derived class of ooObj
// Get the type number using the member functions.
typeNum = objH.typeN(); // handle member function
typeNum = objH->ooGetTypeN(); // object member function
// Get the type name both ways.
printf("Type name from handle is %s\n", objH.typeName());
printf("Type name from object is %s\n", objH->ooGetTypeName());
```
Getting the Number of Containers in a Database

To get the number of containers in a database, use the member function `ooRefHandle(ooDBObj)::numContObjs`. The syntax for this member function is as follows:

```cpp
unsigned long ooRefHandle(ooDBObj)::numContObjs() const;
```

Checking Virtual Table Pointers

In C++ applications, there is a virtual table for each class. Sometimes when an application uses a persistent object, Objectivity/DB cannot obtain a virtual table for it. Undesired behavior can occur if virtual functions are invoked through these objects. In general, undesired behavior is a core dump, usually with an illegal memory access or invalid memory address or similar diagnostic.

Normally, Objectivity/DB does not treat a persistent object without a virtual table as an error. If you want to identify classes that do not have virtual tables during the runtime, you can use the following function. This function is intended for advanced users only:

```cpp
void ooCheckVTablePointer(
    const ooBoolean checkVptr = oocTrue);
```

If this function is called with the argument of `oocTrue`, the system will issue a warning message whenever a persistent object is accessed whose persistent class does not have a virtual table pointer. Otherwise, no warning will be issued.

The following example demonstrates how this situation can occur:

Example

Assume that you created the following DDL schema file and application file in 1992:

```cpp
//file: myDDL1992.ddl
class A : public ooObj {
    virtual void displayObject();
};
class B : public A {
```
virtual void displayObject();
};

//file: myApplication.1992
#include "myDDL1992.h"
...
 ooHandle(A) aH = << somehow find an "A" OR "B" object >>;
...
 aH->displayObject();
...

In this application, no matter whether an object of class A or class B is referenced by the handle aH, the appropriate virtual method, displayObject, is called.

Sometime in the following year you create another DDL schema file that relies on the one from 1992, resulting in the following file:

// file: myDDL1993.ddl
#include "myDDL1992.h"

class C : public B {
    virtual void displayObject();
};

Then you use some application created in 1993 to place objects in the federated database, including some objects of class C. Your 1993 applications can read and write those objects, and use their virtual methods.

Now you go back and try to run your 1992 application. You do so without rebuilding it and, in particular, without relinking it with the file myDDL1993.o that results from compiling myDDL1993.C.

If the 1992 application traverses an association, or follows an object reference, or does a scan operation, and expects to find an object of class A, it might (depending on the specific data) find an object of class B or class C instead. As noted above, it can handle objects of classes A and B. But what happens when it finds an object of class C? Since the 1992 application did not know that C objects could exist, it has never built up the virtual methods that are needed by class C and has never built the appropriate virtual table. So C’s virtual table is absent.

For this reason, the following statement in myApplication.1992:

aH->displayObject()
causes a core dump, since the runtime system attempts to use the missing virtual table to locate the appropriate implementation of displayObject.

There is no proper solution for this problem. The function ooCheckVTablePointer at least allows you to anticipate this problem and provide an escape mechanism. Since the standard ooSignal protocol is used for the warning, perhaps your application can trap and handle the warning, choosing to ignore the object in question.
Monitoring and Tuning Performance

This chapter presents basic concepts involved in monitoring and tuning Objectivity/DB applications.

Tuning Guidelines

The best starting point for tuning an application is to collect performance statistics at various times while your application is running.

Use the native resource usage accounting facilities of your operating system, such as the UNIX `time` command to determine whether your application is I/O bound or CPU bound.

Then call the Objectivity/DB function `ooRunStatus`, which will give you statistics indicating the Objectivity/DB operations occurring in your application. For detailed information about `ooRunStatus` output, see “Obtaining Runtime Statistics Using ooRunStatus” on page 21-18.

You may be able to improve your application in terms of:

- Federated database size
- Runtime speed
- Concurrency

Consider all three when tuning your application, weighting them according to your application's needs.
Tuning Concepts

You should be familiar with the features and concepts described in this section when tuning an Objectivity/DB application.

Clustering

Placing persistent objects physically near each other in secondary storage (on disk) is called clustering. From the Objectivity/DB programming interface, clustering is implied by the containment hierarchy of persistent objects; all objects in the same containing object are located as close as possible on secondary storage, within the limits of the host file system.

Your application can control clustering of an object by using a clustering directive when creating the object. For example, when creating a basic object, the object will be placed on the same page as the object used as the clustering directive if the clustering directive is an object reference or handle of a basic object, and if the page has sufficient space.

Clustering of objects that are frequently accessed at the same time improves federated database performance by reducing disk I/O.

Variable-Size Arrays

Variable-size arrays (VArrays) enable you to store arbitrary length data contiguously in memory or on disk. They are zero-based with indices ranging from 0 to n-1 (where n is the number of elements in the array).

VArrays are derived from class ooVArray. VArrays of class ooVArray can be either persistent or transient. They are persistent when declared within a persistent object.

You can create transient VArrays from class ooVArray, and use them to store arrays of object references, but not handles. You can use VArrays of class ooTVarArray to store arrays of handles.
Object References

Every object in an Objectivity/DB federated database has a unique object identifier (OID), which indicates the object's place in the containment hierarchy of the federated database. Objectivity/DB provides two object reference classes that you can use to access basic objects through their OIDs: ooRef and ooShortRef.

Class ooRef

Each instance of the ooRef class contains four 16-bit fields to represent a basic object's OID. This OID identifies the database, container, page, and slot of the basic object. It provides a single logical address for a basic object anywhere in the federated database.

<table>
<thead>
<tr>
<th>0</th>
<th>16</th>
<th>32</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Number</td>
<td>Container Number</td>
<td>Page Number</td>
<td>Slot Number</td>
</tr>
</tbody>
</table>

Class ooShortRef

Each instance of the ooShortRef class contains two 16-bit fields to represent a basic object's Object Identifier (OID). This OID identifies only the page and slot of the basic object and can refer only to a basic object within the same container.

<table>
<thead>
<tr>
<th>0</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Number</td>
<td>Slot Number</td>
</tr>
</tbody>
</table>

String Classes

Objectivity/DB string classes ooVString and ooString(N) provide member functions and operators to facilitate the use of strings in your application. These strings make more efficient use of space and improve performance; for example, using ooString(N) to store a string uses less space than storing it in a VArray of type char.

For more information on the Objectivity/DB string classes, see the “Using the String Classes” chapter.
Dictionary Classes

The Objectivity/DB dictionary classes ooMap, ooMapElem, and ooMapItr provide a specialized dictionary feature that uses a persistent hash table. This facility can decrease the time it takes to create object names and use these names to locate objects, compared with the time required when you use scope names. It may also use slightly less storage space than scope naming.

For more information on the Objectivity/DB dictionary classes and the hash table, see the “Using the Map Dictionary Classes” chapter.

Using Indexes

Objectivity/DB supports indexes, which provide fast access to containers or basic objects within a specified scope (container, database, or federated database) based on the following:

- Values of fields within the indexed objects
- Range of field values

Tuning for Database Size

This section discusses actions that may significantly affect the size of your federated database.

Minimize Use of Scope Naming

Scope naming is useful for fast random object lookup. However, it is best used only when you have a small number of objects accessed by lookup, since each scope name uses more than 61 bytes of storage. You can calculate the approximate storage needed for a scope name with the following formula:

Required storage for scope name = 61 + (2 x nameLength)

where

nameLength Length of the scope name in bytes.

Use scope names only when necessary. Using scope naming also requires a hashed container, which occupies more storage than a non-hashed container. Therefore you can save storage by using object references, indexes, or associations instead of scope names to locate objects.
Use Dictionary Classes When Naming Many Objects

If your application names, and subsequently looks up, many objects, use the Objectivity/DB dictionary classes (see “Use Dictionary Classes to Name and Locate Objects” on page 21-15). These classes allow you to name objects with minimal space overhead, as well as providing a fast lookup mechanism.

Minimize Number of Hash Pages

You can keep the number of hash pages to a minimum by using non-hashed containers whenever possible. To create a non-hashed container, set the hash parameter to zero when creating the object using the new operator. Hashed containers are necessary when using scope names or keyed objects.

If you use indexing for object lookup, you can avoid the storage overhead for hash pages, although you must take other costs into consideration. See “Use Indexes Instead of Scope Names or Keyed Objects” on page 21-12.

You can also avoid the storage overhead of hash pages by using dictionary classes to name and look up objects. See “Use Dictionary Classes to Name and Locate Objects” on page 21-15.

The default container is always hashed. Therefore unless your application needs hashing, you should avoid using the default container.

Minimize Number of Containers

Consider the number of basic objects you will be creating in the container when you set the initial number of pages. In general, you should cluster basic objects that will be accessed together (that is, in the same transaction). However, remember that every container is allocated a certain minimum number of pages (the default number is four pages for a hashed container, two pages for non-hashed). Thus if you have many containers with only a few objects in each, you may be using more storage than is necessary.

Minimize Size of Containers

If your application does not use the default container, set the initial number of pages of the default container to one when you create a database (if you set it to zero, it defaults to four). If your application does use the default container, set the initial number of pages to the number of pages your application will actually need when you create a database.
Minimize Growth Factor of Containers

When creating a container, consider its initial size and projected growth and set the growth factor accordingly instead of using the default growth factor of 10%. The goal is to avoid container fragmentation and expensive growth operations, thus improving performance without wasting space by allowing containers to grow too large.

Create Single Containers Using new Operator

Use the new operator to set the initial container size. Finding the optimal initial size of a container is important to the performance of the application if you use many scope names. Growing a container can be expensive. Sizing a container with some reserve space for future objects minimizes fragmentation (that is, improves clustering) and container extensions.

You can create multiple containers at the same time with the ooNewConts macro, which is more efficient than creating them one at a time with the new operator. If you create containers using ooNewConts, you cannot give them system names.

Set Associations on a Basic Object at the Same Time

Set as many non-inline associations for a basic object as possible at the same time so that the storage will be allocated contiguously.

Use Unidirectional Instead of Bidirectional Associations

If disk space is of paramount importance, use bidirectional associations only when necessary; that is, when the association is used by both objects. A unidirectional association does not support a back pointer, so it uses only 12 bytes of storage compared to 24 bytes for a bidirectional association. Unidirectional associations do not have automatic referential integrity maintenance.

Use Object References Instead of Unidirectional Associations

An object reference uses eight bytes of storage, four fewer bytes than an equivalent association. A short object reference uses four fewer bytes than an object reference. If you are concerned with disk space usage, consider using object references instead of associations, or use inline associations. Remember that short object references can only access objects within a given container.
Use Short Object References Instead of Object References

Object references are easier to use than short object reference because they can reference objects in other containers and databases. However, an object reference occupies eight bytes of storage, twice as much space as a short object reference. If disk space usage is a concern for your application and you must maintain a large number of references to persistent objects, you can save space by using short object references if all the persistent objects you are referencing reside in the same container.

Use Fixed Arrays or String Classes Instead of VArrays

Use a VArray only when necessary. If the size of an array is not likely to change, you can save space by defining it as a fixed array, which is inherently smaller than a VArray. You can use a transient VArray of class ooTVArray if the data does not need to be persistent. You can create transient VArrays from class ooVArray, and use them to store arrays of object references, but not handles. You can use VArrays of class ooTVArray to store arrays of handles.

The Objectivity/DB string classes also make more efficient use of space than VArrays.

Use Class ooVString for Unknown String Lengths

If the lengths of the strings to be used in a string class are not known or if these lengths vary widely, you should use the ooVString class.

Use Class ooString(N) for Predictable String Lengths

If you can predict the lengths of the strings in a class and if the range of these lengths is not great, it is more efficient to use the ooString(N) class.

When you use the ooString(N) class, you create a VArray and a fixed character array. If the C++ string can be stored in the fixed character array, it is stored there and the VArray of characters is not used. On the other hand, if the C++ string cannot be stored in the fixed character array, the VArray of characters is used.

Because the fixed character array is allocated whether it is used or not to facilitate heterogeneous operation, it is important to specify its size carefully or the array could be too big to be fully utilized or be too small to store the string in most cases. In either case, a lot of storage space is wasted. You should perform an analysis of the usage patterns before selecting the value of N.
Delete Basic Objects Efficiently

The manner in which you delete basic objects may affect the disk space used. The packing density of a container may be very low if most of the objects in it are deleted near the end of a transaction. If you do find it necessary to delete many objects, you should run the ootidy tool periodically to reduce the fragmentation in secondary storage. ootidy will eliminate empty pages but it will not move objects off of nearly-empty pages so that those pages can be deleted.

Consider putting transient basic objects (ones that you know will be deleted) in their own container for easy and space-efficient deletion. This decision is a space/runtime performance tradeoff because the transient objects will not be clustered near the other basic objects used, possibly resulting in more cache activity. Be sure to delete the container when you are finished using it. Deleting a container deletes all of the basic objects it contains.

Be Judicious in Creating Indexes

Indexes provide fast and predictable search capabilities at the cost of additional disk space and memory space to maintain the index. An index duplicates a portion of the information that already resides within the indexed objects. Each index is stored in a container, although the index itself can operate on objects over a specified scope of a container, database, or federated database. An index can account for a large percentage of the container space usage. The space required to support more than one index in a container can be quite large, so be judicious in the creation of indexes.

Example

Assume we have a container-level index, and that the indexed container contains 10,000 basic objects. Each basic object key is 5 bytes long, and the basic objects are each 30 bytes long. If all the pages of the container are full, and the page size is 8K, approximately 44 pages are required to hold all the basic objects in the container. Therefore, approximately 8 pages (about 15 percent of the total page usage in the container) are required to hold an index defined on all the basic objects.
Tuning for Runtime Speed

To achieve optimal runtime speed for your application, you must minimize I/O, networking overhead, and CPU time. The following actions may significantly improve the speed of your application by increasing the efficiency with which it can access and manipulate objects in your federated database.

Avoid Hash Overflows

In applications that use keyed objects or scope names extensively, hash overflows may occur when the hash pages allocated are insufficient for the hashing required. These overflows cause the container to be extended at run time, which is costly in terms of speed. Also, extending the container may make it non-contiguous on disk, leading to higher disk latency.

If you see more than zero hash overflows in your ooRunStatus output, you may want to adjust the initial container size when creating your containers. For user-defined containers, specify these parameters using the new operator. For the default container, specify these parameters when creating the database. To calculate a good initial container size, use the information provided in “Avoid Container (OC) Extensions” on page 21-9.

Also consider using dictionary classes for objects that are frequently accessed (see “Use Dictionary Classes to Name and Locate Objects” on page 21-15) since they allow for expansion of the hash table used for lookups.

Avoid Container (OC) Extensions

In general, you should cluster basic objects that will be accessed together. However, if you create more basic objects in a given container than will fit, Objectivity/DB extends the size of the container at runtime to accommodate the new objects. Since this extension occurs at runtime, it adversely affects runtime speed and may result in a non-contiguous container on disk. Therefore, if you see more than a few OC Extensions in your ooRunStatus output, consider making your containers larger when you create them. An object cluster (OC) is the physical implementation of a container.

The default size of a hashed container is only four pages and the default growth percentage for containers is 10% of the current size. The initial size of a container is extremely important for the speed of keyed object lookups and scope name lookups. You must use a hashed container for scope names and keyed object; the hash value is used as a hash clustering factor when you create keyed objects and use scope names.
Find the Best Cache Size

The default initial cache size is 200 pages, which can grow to a maximum of 500 pages.

The optimal cache size depends on the application, main memory size, and virtual memory space. In general, the best cache size is just large enough to hold the active working set of data that your application will be operating on most of the time.

A smaller cache size is often better than a larger one. But, if your cache size is too small, your application will be doing unnecessary swapping. Using ooRunStatus, you can evaluate your application’s cache size. The cache size may be too small if either of the following conditions is true:

- Proportion of disk reads to buffers read is high
- Number of buffers used is the same as the maximum number of buffer pages you specified with the ooInit function

To find the best cache size, keep reducing the initial number of buffer pages until ooRunStatus shows a Number of Buffers Used that is greater than the initial number of buffer pages you specified. You specify the initial and maximum number of buffer pages in the cache using the ooInit function. See “Initializing Database Services” on page 2-26 for details.

Minimize the Size of Your Database

In general, your application will run faster with a smaller federated database since I/O operations take time and a smaller federated database requires less I/O. For guidelines on minimizing the size of your federated database, see the “Tuning for Database Size” on page 21-4.
Cluster Objects Efficiently

In general, you should cluster basic objects that will be accessed together. If your application frequently uses certain associations between basic objects, you should try to place these associated objects in the same container. This clustering approach will minimize the disk reads and lock server remote procedure calls needed to access associated objects.

When creating a new basic object, specify the proper clustering directive. For example, use the object reference or handle of a container to ensure that the object is created in the right container. Use the object reference or handle of a basic object to ensure that the new basic object is created on the same page as that object if space is available, or a nearby page. If you are creating large numbers of objects, use the object reference or handle of the most recently created object as a clustering directive. By specifying proper containment, you reduce disk I/O, improving performance.

You can use the copy and move member functions to try to change the clustering of basic objects, but there is never any guarantee that an object will be placed on the same page as others.

Minimize Use of Scope Names and Keyed Objects

For improved runtime speed, look up objects through associations, object references, or using the ooMap dictionary class instead of scope names or keys when possible.

Using scope names and keyed objects can speed up your application’s performance; however, it can also slow it down because of hashing overhead required for object lookup, and the extra storage overhead needed for scope names. Objectivity/DB containers can be either hashed or non-hashed. Only hashed containers support scope naming and keyed objects.

For more information about creating containers, see “Creating Containers” on page 3-7. For more information about scope names and keyed objects, see “Using Scope Names” on page 8-6 and the “Using Keyed Objects” chapter. See “Use Dictionary Classes to Name and Locate Objects” on page 21-15 for information on using the Objectivity/DB dictionary classes for faster object lookup.
Use Indexes Instead of Scope Names or Keyed Objects

You may be able to improve runtime speed by using indexes for object lookup instead of scope names or keys. An index can locate an object quickly given its key, and may be required when object lookup must be fast.

Also see “Use Dictionary Classes to Name and Locate Objects” on page 21-15 for information on using the Objectivity/DB dictionary classes for faster object lookup.

Explicitly Updating Indexes

You can use explicit update to improve index performance when your application is update intensive and utilizes database and federated database wide indexes.

`oocExplicitUpdate` gives you explicit control over changes to indexed objects during a transaction. In certain update-intensive applications, where database and federated database wide indexes are used, but where the updated members are not indexed, using `oocExplicitUpdate` will improve performance.

Also see “Explicitly Updating Indexes” on page 14-16 for more information on this index mode.

Avoid Unnecessary Opening of Containers

Operations on basic objects within a container are very efficient. Operations on containers are much less efficient because the system name space must be accessed first. Try to avoid unnecessary opening of containers.

Use `ooNewConts` When Creating Multiple Containers

When you are creating many containers at one time, use the `ooNewConts` macro to create them instead of creating them one at a time with the `new` operator.
Find the Best Page Size

The best page size for your application will depend on many factors, including the average size of your basic objects and the amount of memory available. Basic objects clustered together are placed contiguously in the pages of their containers. These pages are read into memory when the basic objects are opened by your application. Only when all of the basic objects on a page are closed can that page be swapped back to disk to make room in the cache for another page.

Look at the pattern of object access in your application and consider the following general rules:

- If your application randomly accesses many basic objects, you maybe better off with a small page size.
- If your application spends a great deal of time on a fixed set of data, a larger page size may be better.

Try a number of different page sizes to determine the best size for your application; 2048, 4096, or 8192 bytes tend to be best for local files. For remote files, 1024 bytes works well.

You specify the page size when creating the federated database with the `oonewfd` tool. You cannot change the page size later.

Page size affects your application’s cache size and the size of each container in your application. Therefore, when altering the page size, you should also adjust the number of buffer pages in your cache and the number of initial pages in your containers accordingly. Adjust number of buffer pages using the `ooInit` function, and the number of container pages using the `new` operator. Remember that after changing the page size, you will need to rebuild your federated database; you cannot use an existing federated database with a new page size.

Set Associations as Soon as Possible

If possible, associate objects as soon as they are created. This keeps the association data near the object data, probably on the same page. Setting a bidirectional association opens the associated object for update, therefore requiring space in the cache if the objects are on different pages.
Pass Object References or Handles by Reference Instead of Value

When using object references or handles, avoid constantly opening, closing, and then opening the same object, which is time consuming.

To save time, always pass object references and handles by reference. For example, the first function call below is faster than the second:

```c++
func(ooHandle(Computer)& computerH); // faster
func(ooHandle(Computer) computerH);    // slower because transient handle is required
```

Avoid Constantly Resizing VArrays

Set the initial size of a VArray close to its final size to minimize the number of calls to the `resize` member function. However, avoid setting it larger than necessary so that you do not waste disk space.

Use the `elem` member function instead of `operator[]` to avoid bounds checking where appropriate.

Specify the Open Mode for Iterators

Specify the open mode when initializing an iterator if each target object is to be accessed in the same mode during the iteration. For example, when counting objects, use `oocNoOpen`; for reading, use `oocRead`; for updating, use `oocUpdate`.

Use `contains` Instead of `scan`

Use the `contains` member function to find all target containers in the database. The `scan` member function performs the same logical function more slowly. However, to find all objects that satisfy a particular condition or set of conditions, you must use the conditional form of `scan`.

Assign Explicit Handles When Using Object References

When you use `operator ->` on an object reference, it may cause a transient handle to be generated. If you use `operator->` several times to access object member functions or data members, you can decrease the number of transient handles created as well as the number of opens and closes of an object by assigning the implicit handle to an explicitly declared handle, and using the explicit handle instead of the object reference.
Minimize the Number of Indexed Objects

The number of disk I/Os needed to locate an object through an index is directly related to the number of levels in the index, which is directly related to the number of index objects in the index. You can help minimize the number of disk I/Os needed to locate an object by indexing at the appropriate level of the object hierarchy.

An index is used to locate objects of a particular base class and its derived classes. The class on which an index is defined is specified when the index is created. In order to reduce the size of an index (and hence to reduce the processing costs of the index), define the index on the most specific class possible.

Example

Suppose a schema contains a class called employees with two derived classes called doctors and nurses. If an application requires fast access to locate specific doctors, then the index should be defined on the doctors class. The same is true for the nurses class. If the application requires fast access to both doctors and nurses (but not within the same request), then two indexes are preferable: one on the doctors classes and one on the nurses class. Only if the application requires access to doctors and nurses at the same time should you define an index on the employees class.

Use Dictionary Classes to Name and Locate Objects

If you use the Objectivity/DB dictionary classes carefully, they can decrease the time it takes to create object names and use these names to locate objects. The hash table you construct with the ooMap member function is dynamic because the number of hashing bins can grow when a certain threshold is reached. However, the runtime speed of your application decreases each time the table is resized. When you construct a hash table: set the number of bins, the average number of elements per bin allowed before resizing, and the growth factor, to minimize the number of times the table will be resized at runtime.
Use the String Classes to Store String Data

The Objectivity/DB string classes provide string class member functions and operators to facilitate the use of strings. When used properly, these classes make more efficient use of space and improve performance.

Use Class ooVString for Unknown String Lengths

If the lengths of the strings to be used in a string class are not known or if these lengths vary widely, you should use the ooVString class.

Use Class ooString(N) for Predictable String Lengths

If you can predict the lengths of the strings in a class and if the range of these lengths is not great, it is more efficient to use the ooString(N) class.

When you use the ooString(N) class, you create a VArray and a fixed character array. If the C++ string can be stored in the fixed character array, it is stored there and the VArray of characters is not used. On the other hand, if the C++ string cannot be stored in the fixed character array, the VArray of characters is used.

Because the fixed character array is allocated whether it is used or not to facilitate heterogeneous operation, it is important to specify its size carefully or the array could be too big to be fully utilized or be too small to store the string in most cases. In either case, a lot of storage space is wasted. You should perform an analysis of the usage patterns before selecting the value of N.

Tuning for Concurrency

The following actions may significantly affect the concurrency (that is, the simultaneous access to objects by multiple processes) in your application.

Do Not Explicitly Lock Objects

Objectivity/DB provides implicit locking of objects under certain conditions. For example, when a container is locked, the basic objects within it are implicitly locked. In general, maximum concurrency is achieved by using this implicit locking. Explicit locking requested by your application can reduce the application’s concurrency dramatically if used improperly.
Consider Access Granularity

Opening or locking an object for update access at the smallest level of granularity possible increases overall concurrency. However, this approach also increases the chance of later competing with other processes for access rights.

Currently the smallest level of locking granularity is a container. By controlling the clustering of objects in containers, you can exercise some control over the granularity of locking in your application.

The smaller the number of objects in each container, the greater the concurrency, but the more potential lock requests needed to lock all objects used by a transaction. However, also consider the discussion in “Cluster Objects Efficiently” on page 21-11.

Use Unidirectional Associations

Whenever you either set or remove a bidirectional association on an object, update access privileges on the associated object are automatically obtained by Objectivity/DB. If there are many processes that wish to either add or remove a bidirectional association to a single object, all the other processes have to wait until the process with update access permission commits its transaction, thus slowing performance.

If it is not important to be able to traverse the association from both sides, then a unidirectional association may give better performance.

Make Container-Level Indexes Sharable

Objectivity/DB allows concurrent multiple user access to both database- and federated database-level indexes. This allows multiple transactions to use an index to simultaneously access objects. However, at any given instant, only one transaction can update an index.

Container-level indexes can support multiple users for read operations if the container on which the index is defined is opened in read mode (oocRead). Read operations are much faster in terms of access than write operations, because write access to a container-level index or its objects is restricted to a single user during a transaction; this happens because container-level indexes are reliant upon the container-level locking granularity of Objectivity/DB.
Obtaining and Interpreting Runtime Statistics

Objectivity/DB maintains a set of internal statistics on a process basis. By looking at these statistics, and statistics available from your operating system at runtime, you can determine what measures to take to tune your application.

Obtaining Runtime Statistics Using ooRunStatus

There are two ways to obtain the Objectivity/DB internal statistics for C++ applications:

- Call the function ooRunStatus from within your application.
- Use the dbx/oodbtools alias from within dbx and issue the oodebug command stats, which executes the ooRunStatus function.

The syntax for the ooRunStatus function is:

```
void ooRunStatus();
```

Interpreting Objectivity/DB Internal Statistics

Consider the following guidelines when reviewing Objectivity/DB internal statistics:

- The ratio of buffers read to disk reads should be as high as possible because it indicates the clustering efficiency of the program.
- The number of multiple opens should be low or 0.
- The number of forced file closes should be 0.
- The number of hash overflows, OC extensions, and pages added to OCs should be 0.
- The number of containers created or containers opened can affect performance. Container operations are slower than basic object operations.
- Try to minimize the numbers of disk reads and page writes.

Only a few of the Objectivity/DB internal statistics have a significant impact in terms of performance. The following is a list of these statistics to help you tune your application.

Number of multiple opens

This value indicates the number of objects that have been opened more than once before the object is closed. Ideally, this value should be zero. While there may be specific instances where it makes sense to open an object more
than once, in general it is a poor practice because it may lead to objects being inadvertently left open long after they are needed.

This in turn ties up space in the cache with unneeded data, which may adversely affect performance. If this value is not zero, look for multiple opens from the same object reference or handle. Also try passing object references or handles by reference instead of value.

**Number of new OCBs**

If the number of Object Control Blocks (OCBs) is larger than 256, then the process may have too many objects open simultaneously. This may have an adverse effect on performance by using more virtual memory than necessary.

**Number of associations resized**

If this value is high relative to the **Number of new associations** statistic, it may be worth setting at least one association immediately after creating a new object. This will reserve space for the association links in the same page as the object itself. When you create one association, Objectivity/DB automatically allocates enough storage for two more. When all three associations are used, Objectivity/DB adds one more. Thereafter, each time all associations are used, it applies a growth factor of 40% (Objectivity reserves the right to change this algorithm in future releases.) Also see the discussion below on “Number of SM objects resized”.

**Number of buffers used**

If this figure is significantly higher than the initial number of buffer pages allocated in the cache (determined by the `nPages` parameter to the `ooInit` function), Objectivity/DB is making a relatively expensive call to malloc to extend the cache as needed.

Consider setting a higher initial number of buffer pages using the `nPages` parameter to `ooInit`. See “Initializing Database Services” on page 2-26.
Obtaining and Interpreting Runtime Statistics

Number of forced file closes
If this figure is non-zero then the number of file descriptors allocated to Objectivity/DB is too low (determined by the nFiles parameter to the ooInit function). Opening and closing files is much slower than opening and closing objects. You should consider setting a higher number of file descriptors (greater than the number of database Files to be opened) using the nFiles parameter to ooInit.

Number of hash overflows
If this figure is high then the amount of space reserved for hashed objects should be greater increased. Increase the initial size of the container in which the hashed objects are located. Use the new operator to specify the initial size of the container. See the “Creating and Deleting Objects” chapter.

Number of times OCs extended
If this statistic is high relative to the number of object clusters (OCs) you have updated thus far, then the initial container size is too small. Either increase the initial size of the containers or increase the amount by which they are extended.

Number of SM objects resized
If this figure is approximately equal to the total number of times your application either extends a VArray or adds an association, then it may indicate that your clustering is too dense. If too many objects are placed on a page, then Objectivity/DB must allocate more storage on another page each time you extend a VArray or add an association to an object located on the page. Control the clustering of objects with the clustering directive of the new operator. This value may also be high because the number of associations is increasing.
Examine the Number of associations resized statistic. If it is high relative to the Number of new associations statistic, it may be worth creating at least one association immediately after creating a new object. This will reserve space in the same page as the object itself. In extending VArrays, keep in mind that VArrays must always be contiguous in memory and that extensions will frequently involve copying the entire VArray to a new location in memory. Therefore, you may want to consider preallocating a larger amount of storage or resizing by a larger amount less frequently to minimize this contingency. However, if your estimate is too large, your application may perform more I/O than necessary.
Object Conversion

This chapter presents object conversion tasks you can perform to change existing objects in a federated database after schema evolution. For information about schema evolution, see the “Schema Evolution” chapter in Using Objectivity/ C++ Data Definition Language.

Basics

After performing schema evolution, you may need to augment the automatic conversion of existing objects performed by Objectivity/C++ for the classes you changed. These objects are called affected objects, and the process of changing them to reflect schema changes is called object conversion. See Figure 22-1.

Objectivity/C++ supports the three object conversion modes—deferred, on-demand, and immediate. You should choose one of these modes based on the requirements of your applications and the schema evolution operations you perform.

For overview information about these modes and their relation to schema evolution operations, read the “Schema Evolution” chapter in Using Objectivity/ C++ Data Definition Language. Detailed information about how to use these three modes are described in the following sections.

This chapter also presents information on creating and running user-defined conversion functions and upgrade applications, which you may need to use to augment conversion of affected objects. See “Using Conversion Functions” on page 22-5 and “Using Immediate Mode with an Upgrade Application” on page 22-4 for more information.
Using Deferred Mode

With the **deferred** mode, Objectivity/C++ automatically converts an affected object to its most recent class representation when any user application accesses the object for the first time following the change to its class. If the transaction is an update transaction, the converted object is stored in the federated database. If it is a read-only transaction, the object will be converted for the duration of the transaction, but will not be stored in the federated database.

The deferred mode allows applications to continue to access the federated database since objects are converted one at a time. This is useful if your deployed environment cannot afford the downtime or reduced access to objects that may occur by updating all affected objects at once.

This mode is also suitable for applications that have a large number of affected objects but only small number of these objects will be accessed. With the deferred mode, you avoid the time required to convert all affected objects.
Using On-Demand Mode

With the on-demand mode, you trigger automatic object conversion from within any user application at the granularity of a container, a database, or the entire federated database. You can use this mode to selectively convert objects based on their containment.

The on-demand mode on a federated database converts all affected objects that have not already been converted in the federated database. This is useful when you want to convert all affected objects, but are not required to use an upgrade application for the conversion.

For performance reasons, updating all objects at once may be better than the deferred mode. This is particularly true for short, read-only transaction applications that may incur additional performance costs for object conversions by converting affected objects one by one.

Converting all affected objects at once also eliminates the possibility of a read-only transaction converting (but not saving) an affected object multiple times when it is accessed. This mode also purges the schema information that is used to convert the objects, which will reduce the time required to load schema information at runtime, since the older, unused representations are removed.

Procedure

Perform the following step to use on-demand conversion:

1. From within an update transaction, call the `convertObjects` member function on the object reference or handle of the object containing the affected objects.
   ◆ To convert all objects within a specific container, call the member function `ooRefHandle(ooContObj)::convertObjects`. See page K-22.
   ◆ To convert all objects within a database, call the member function `ooRefHandle(ooDBObj)::convertObjects`. See page K-30.
   ◆ To convert all objects, call the member function `ooRefHandle(ooFDObj)::convertObjects`. See page K-42.
Using Immediate Mode with an Upgrade Application

With the immediate mode, Objectivity/C++ automatically converts all affected objects in a federated database within a single transaction with the help of a single-use upgrade application. All affected objects are converted before you run any other applications against the federated database. Performing immediate conversion also purges information from the schema that is used to convert the objects.

Immediate mode (and an upgrade application) are only required for schema operations that include replacing a base class or deleting a class that has inherited associations or object references. For these changes, the DDL processor marks the classes affected by the change as protected until the upgrade application is run. If a non-upgrade application tries to access an object of one of these protected classes before the upgrade application is run, Objectivity/C++ issues an error message. You can trap this message in other applications to notify users that they need to run the upgrade application on their federated database.

Creating an Upgrade Application

If your schema evolution operations required you to run the DDL processor with the -evolve and -upgrade flags, you must create and run a separate single-use, upgrade application to convert all affected objects before running any other applications.

Upgrade applications are different from other applications because they call special upgrade interface member functions to perform the upgrade process.

Some operations, such as deleting a class that has inherited associations or object references, only require a simple upgrade application. A simple upgrade application does not require you to write any special code for object conversion (unless you need to use a conversion function to augment automatic conversion). For most simple upgrade applications, you only need to call the upgrade interface member functions.

Other schema evolution operations require you to write object conversion code in the upgrade application based on the kind of conversions you need to perform.
Upgrade Application Procedure

Perform the following steps to create and run an upgrade application. This procedure runs the application in the immediate conversion mode:

1. Create the upgrade application based on the schema evolution operations you made. If your operations only require a simple upgrade application, you only need to create the application. No special object conversion code is necessary.

2. Initiate the upgrade application by calling `ooTrans::upgrade` at the start of the upgrade application (see page H-5). This member function notifies Objectivity/C++ that an upgrade application is running. You must call this member function before calling `ooRefHandle(ooFDObj)::open` (see page K-47) and `ooTrans::start` (see page H-3). `ooTrans::upgrade` converts all affected objects.

3. Register any desired conversion functions by calling member function `ooRefHandle(ooFDObj)::setConversion` (see page K-48).

4. Initiate the upgrade process by calling the member function `ooRefHandle(ooFDObj)::upgradeObjects` (see page K-51).

5. Commit the transaction to complete the object upgrade process.

Using Conversion Functions

For any of the conversion modes, deferred, on-demand, or immediate, you can specify a user-defined object conversion function when converting objects to augment the built-in conversion of Objectivity/C++. This is useful for conversions that use existing data member values to compute new values, or change the semantics of a data member.

Objectivity/C++ provides two classes you can use to get and set data member values—`ooConvertInObject` and `ooConvertInOutObject`. Using these classes, you can get the value of an embedded data member of an existing object and set the value in the new object representation. `ooConvertInObject` provides read-only access to the data members, while `ooConvertInOutObject` provides read and update access.

You can also use a conversion function to convert a data member that is common to multiple classes by writing a simple conversion function for a class of embedded data members or a base class.

For consistency of object conversions (particularly in deferred mode), you should limit the use of the conversion function to only those objects being converted. You should not access other objects.
To use a conversion function, you must register it as part of the initialization of the federated database in all applications that may access an object of an evolved class. For an example of using conversion functions, see “Using Immediate Mode with an Upgrade Application” on page 22-4.

**Registering a Conversion Function**

You must register a conversion function at runtime before Objectivity/C++ can call it to convert objects of a specified class.

**Procedure**

Perform the following steps to register a conversion function:

1. Create a conversion function that has two parameters, a reference to an existing unconverted object, and a reference to the converted object. See `userConversionFunction` on page N-2 for a description of the required syntax. You can have no more than one conversion function for each changed persistent-capable class. If you register more than one conversion function, only the last one will be used.

2. Register the conversion function by calling the member function `ooRefHandle(ooFDObj)::setConversion` (see page K-48). Specify the class name of the objects to be converted and the name of the conversion function.

**Getting Primitive Data Member Values**

To get data member values for existing and converted objects, use member functions from the non-persistent-capable classes `ooConvertInObject` and `ooConvertInOutObject`. Both of these classes provide member functions for getting data member values.

`ooConvertInObject` does not currently support variable-length data members (VArrays), associations, or object references.
Procedure

Perform the following step to get a data member value:

1. To get the value of a primitive data member of an unconverted object, call one of the member functions of the class `ooConvertInObject` (see page R-1):
   - `ooConvertInObject::getInt8` on page R-1
   - `ooConvertInObject::getInt16` on page R-2
   - `ooConvertInObject::getInt32` on page R-2
   - `ooConvertInObject::getUint8` on page R-2
   - `ooConvertInObject::getUint16` on page R-3
   - `ooConvertInObject::getUint32` on page R-3
   - `ooConvertInObject::getFloat32` on page R-3
   - `ooConvertInObject::getFloat64` on page R-4

Setting Primitive Data Member Values

The non-persistent-capable class `ooConvertInOutObject` provides member functions for you to get and set data member values of primitive types. You can use these member functions to access an object’s state after it is converted by object conversion operations. These member functions can access variable-length data members (VArrays).

Procedure

Perform the following step to set a data member value of a converted object:

1. To set the value of a primitive data member of a converted object, call one of the following member functions of the class `ooConvertInOutObject` (see page R-5):
   - `ooConvertInOutObject::setInt8` on page R-6
   - `ooConvertInOutObject::setInt16` on page R-6
   - `ooConvertInOutObject::setInt32` on page R-6
   - `ooConvertInOutObject::setUint8` on page R-7
   - `ooConvertInOutObject::setUint16` on page R-7
   - `ooConvertInOutObject::setUint32` on page R-7
   - `ooConvertInOutObject::setFloat32` on page R-8
   - `ooConvertInOutObject::setFloat64` on page R-8
Getting the Objects that Use a Specified Data Member

You can get the objects that have a specified data member. This is useful if you want to use a conversion function to convert a class data member that is used by many classes.

Procedure

Perform the following step to get objects of a data member:

1. Call `ooConvertInObject::getOldDataMember` (see page R-4) or `ooConvertInOutObject::getNewDataMember` (see page R-5).

Getting the Objects that Have a Specified Base Class

You can get the objects that have a specified base class. This is useful if you want to use a conversion function to change a base class that is common to many classes.

Procedure

Perform the following step to get objects of a base class:

1. Call `ooConvertInObject::getOldBaseClass` (see page R-4) or `ooConvertInOutObject::getNewBaseClass` (see page R-5).

Example Upgrade Application with Conversion Functions

The following example changes a course enrollment schema shown in Figure 22-2 to the one shown in Figure 22-3. The evolved schema contains the following user-defined classes—Address, Project, Student, Course, TeachingAssist, and GradStudent. The example then shows an upgrade application used to convert existing objects to the new schema. The upgrade application augments automatic conversion through conversion functions.
Figure 22-2  Course Enrollment Example Schema (Before)

Figure 22-3  Course Enrollment Example Schema (After)
// Course enrollment schema code

// Address class
class Address {
    ooVString _streetName;
    ooVString _city;
    char state[2];
    uint8 zipCode[5];
};

// Project class
class Project: public ooObj {
    ooVString _ProjectName;
    public:
        ooRef(Student) assigned[] <-> responsibleStudents[];
        inline ooRef(Course) assignedFor;
        Project(const char *ProjectName);
};

// Course class
class Course: public ooObj {
    ooVString _CourseTitle;
    uint16 _numberCredits;
    public:
        Course(const char *title);
        ooRef(Course) assistant[] <-> assistIn[];
};

// Student class
class Student: public ooObj {
    ooVString _major;
    float32 _GPA;
    uint16 _yearAdmitted;
    Address _homeAddress;
    float32 _height;
public:
  ooRef(Project) responsibleStudents [] <-> assigned[];
  Student ();
  Student (const char *name);
  char *getAll ();
};

// Teaching assistant class
class TeachingAssist: public Student {
  ooRef(Course) assistIn [] <-> assistant [];
};

+ // Changing a base of GradStudent from Student to TeachingAssist
+ #pragma oochangebase Student -> TeachingAssist

// Graduate student class
class GradStudent: public TeachingAssist {
  float32 _undergradGPA;
  ooVString _undergraduateMajor;
};

The following is the upgrade application code to apply the above changes to existing objects.

// Course enrollment upgrade application code
#include "generated.h"
void FatalError(cont char *text){
  fprintf(stderr, "A fatal error has occurred: "%s"
    failed\n", text);
  exit(-1);
};
// Convert height in inches (from existing objects) to
// height in centimeters (in converted objects)
void
StudentConversion(const ooConvertInObject existingObject,
   ooConvertInOutObject convertedObject) {
   int i;
   float32 ht;
   ooStatus status;
   float32 theHeight [100];

   for (i = 0; i < 9; i++) {
      sprintf (theHeight, "_height[%d]", i);
      existingObject.getFloat32 (theHeight, &ht);
      theHeight = theHeight/.3937;
      convertedObject.setFloat32 (theHeight, ht);
   }
}

// Main program
main() {
   ooTrans transaction;
   ooStatus status;
   ooHandle(ooFDObj) fdH;

   status = transaction.upgrade();
   (void) ooInit ();
   transaction.start ();
   if (status != oocSuccess) {
      FatalError("Transaction upgrade");
   }

   status = fdH.open ("testFD", oocUpdate);
   if (status != oocSuccess) {
      FatalError("Open FD testFD for update");
   }
}
Performance Considerations

This section describes the effect on performance when objects are converted after schema evolution.

Redirected Objects

Adding a data member to a class increases the size of its affected objects. Depending on how the object is stored on disk, this size increase may require that the objects are stored on another page. Accessing these redirected objects through object references and associations may result in an additional I/O if the page is not currently in the cache.

To minimize the need for redirected objects, you may want to pad classes with placeholder data members if you anticipate adding new data members to this class in a future schema evolution. This eliminates the need for redirection as a side effect of converting the affected objects.

status = fdH.setConversion("Student", StudentConversion);
if (status != oocSuccess) {
    FatalError("Setting Student conversion");
}
status = fdH.setConversion("GradStudent", StudentConversion);
if (status != oocSuccess) {
    FatalError("Setting GradStudent conversion");
}
status = fdH.upgradeObjects();
if (status != oocSuccess) {
    FatalError("Upgrade Objects");
}
transaction.commit();
} // end of main
Read-Only Transactions

If a read-only transaction accesses an affected object before it is converted, the transaction will convert the object to read it, but will not save the conversion. Because of this, the transaction will experience the conversion performance overhead for the object each time it is opened. You can reduce this overhead by using the on-demand interface as broadly as possible.

Deploying to End Users

After performing schema and object conversion, you can deploy your changes to your end users as follows:

1. Prepare the following files and distribute them to your end user:
   - A copy of your upgraded federated database without any development databases
   - The upgrade application (if any) required for object conversion
   - Other applications you provide with your product that have been modified and rebuilt to use the new federated database

2. Instruct your end users to install the new federated database using ooinstallfd (see page A-29).

3. Instruct your end users to attach their existing databases to the new federated database using ooattachdb (see page A-5).

4. If you provided an upgrade application, instruct your users to run it on the new federated database.

5. Instruct your end users to run the new applications you provided and load in the upgraded federated database.

This process is illustrated in Figure 22-4.
Figure 22-4  End-User Object Conversion
Effect of Schema Changes on Indexes

Certain schema evolution operations may affect existing indexes and their key descriptors. These effects are summarized in Table 22-1. You can check for inconsistencies in a key descriptor or key field after schema evolution using the `ooKeyDesc::isConsistent` (see page F-3) and `ooKeyField::isConsistent` (see page F-5) member functions. An inconsistent key descriptor can occur when a data member is deleted that is one of the key descriptor's key fields. Changing the type of a primitive data member that used as one of the key descriptor's key fields can also lead to an inconsistent key descriptor.

Table 22-1: Effects on Indexes

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect on Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dropping an indexed field</td>
<td>Invalidates the key descriptor and drops the key descriptor's associated indexes</td>
</tr>
<tr>
<td>Changing an indexed field's type</td>
<td></td>
</tr>
<tr>
<td>Deleting a class</td>
<td>Drops corresponding index entries automatically</td>
</tr>
<tr>
<td>Dropping a base class</td>
<td></td>
</tr>
<tr>
<td>Replacing a base class</td>
<td></td>
</tr>
<tr>
<td>Adding or replacing a base class</td>
<td>May require adding index entries. You can add entries explicitly or implicitly as a side-effect of converting an object. For more information, see “Setting Index Modes” on page 14-15 and “Explicitly Updating Indexes” on page 14-16. Objectivity/C++ will automatically add index entries only if the affected object is converted in an update transaction.</td>
</tr>
</tbody>
</table>
Using the ODMG Interface

This chapter lists Objectivity/DB types and classes that conform to a subset of Release 1.1 of the Object Database Management Group (ODMG) interface, and describes the basic process for developing ODMG applications with Objectivity/DB. The ODMG interface is an object database standard that allows you to develop applications that can be shared with other systems that support this standard.

We assume you are familiar with the ODMG interface, and have access to a copy of the document The Object Database Standard: ODMG -93 Release 1.1.

Basics

Several Objectivity/DB classes are equivalent to classes in the ODMG interface. Because of these similarities, you can substitute ODMG class names in your schema and application code for their equivalent Objectivity/DB classes. Table 23-2 and Table 23-2 list names Objectivity/DB classes and types that map directly to the ODMG standard classes and types. These tables also list page references you can use to find more information about these classes.

Table 23-1: ODMG and Objectivity/DB Class Name Equivalents

<table>
<thead>
<tr>
<th>ODMG Name</th>
<th>Objectivity/DB Name</th>
<th>Refer to page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_Persistent_Object</td>
<td>ooObj</td>
<td>D-1</td>
</tr>
<tr>
<td>d_Ref&lt;userClass&gt;</td>
<td>ooRef(userClass)</td>
<td>A-8, P-2</td>
</tr>
<tr>
<td>d_Ref&lt;d_Persistent_Object&gt;</td>
<td>ooRef(ooObj)</td>
<td>K-2</td>
</tr>
<tr>
<td>d_String</td>
<td>ooVString</td>
<td>G-1</td>
</tr>
<tr>
<td>d_Transaction</td>
<td>ooTrans</td>
<td>H-1</td>
</tr>
<tr>
<td>d_Varray&lt;type&gt;</td>
<td>ooVArray(type)</td>
<td>A-10, I-1</td>
</tr>
</tbody>
</table>
### Table 23-2: ODMG and Objectivity/DB Type Name Equivalents

<table>
<thead>
<tr>
<th>ODMG Name</th>
<th>Objectivity/DB Name</th>
<th>Refer to page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>d.getBoolean</td>
<td>ooBoolean</td>
<td>A-2</td>
</tr>
<tr>
<td>d_CHAR</td>
<td>char</td>
<td>2-8 in Using Objectivity/ C++ Data Definition Language</td>
</tr>
<tr>
<td>d_DOUBLE</td>
<td>float64</td>
<td>2-8 in Using Objectivity/ C++ Data Definition Language</td>
</tr>
<tr>
<td>d_FLOAT</td>
<td>float32</td>
<td>2-8 in Using Objectivity/ C++ Data Definition Language</td>
</tr>
<tr>
<td>d_LONG</td>
<td>int32</td>
<td>2-8 in Using Objectivity/ C++ Data Definition Language</td>
</tr>
<tr>
<td>d_OCTET</td>
<td>char</td>
<td>2-8 in Using Objectivity/ C++ Data Definition Language</td>
</tr>
<tr>
<td>d_SHORT</td>
<td>int16</td>
<td>2-8 in Using Objectivity/ C++ Data Definition Language</td>
</tr>
<tr>
<td>d_ULONG</td>
<td>uint32</td>
<td>2-8 in Using Objectivity/ C++ Data Definition Language</td>
</tr>
<tr>
<td>d_USHORT</td>
<td>uint16</td>
<td>2-8 in Using Objectivity/ C++ Data Definition Language</td>
</tr>
</tbody>
</table>
Objectivity/DB also provides ODMG-specific classes and functionality, as listed in Table 23-3.

**Table 23-3: ODMG-Specific Classes**

<table>
<thead>
<tr>
<th>ODMG Name</th>
<th>Refer to page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_Ref_Any</td>
<td>K-55</td>
</tr>
<tr>
<td>d_Database</td>
<td>K-52</td>
</tr>
<tr>
<td>d_Iterator</td>
<td>J-6</td>
</tr>
<tr>
<td>d_Date</td>
<td>S-1</td>
</tr>
<tr>
<td>d_Time</td>
<td>S-11</td>
</tr>
<tr>
<td>d_Timestamp</td>
<td>S-20</td>
</tr>
<tr>
<td>d_Interval</td>
<td>S-28</td>
</tr>
</tbody>
</table>

**Terminology**

The ODMG standard refers to the highest logical storage level of an ODBMS as a *database*. In contrast, Objectivity/DB refers to the highest storage level as a *federated database* and the next level down as a *database*. To minimize confusion, between ODMG databases and Objectivity/DB databases, in this document we use the definitions listed in Table 23-4:

**Table 23-4: Objectivity/DB and ODMG Terminology**

<table>
<thead>
<tr>
<th>Objectivity/DB Term</th>
<th>Equivalent ODMG Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objectivity/DB federated database</td>
<td>ODMG database</td>
</tr>
<tr>
<td>Objectivity/DB database</td>
<td>No equivalent</td>
</tr>
</tbody>
</table>
Template Requirements

Since the ODMG interface is implemented using C++ templates, you can only use this interface if you compile your application code using a compiler with template support.

Enabling the ODMG Interface

Objectivity/DB ODMG support is disabled by default. To enable ODMG support (that is, add the ODMG types and definitions), use the -DOO_ODMG flag when running both the DDL processor and your C++ compiler.

Developing an ODMG Application

This section describes the basic process of developing an Objectivity/DB ODMG application. Objectivity/DB ODMG classes and types (except for time and date classes) are incorporated into standard Objectivity/DB header and library files.

Developing an ODMG application is similar to developing a non-ODMG Objectivity/DB C++ application. The basic steps are as follows:

1. Use the oonewfd tool to create an Objectivity/DB federated database (equivalent to an ODMG database). This task is usually performed by your database administrator. For more information, see the "Federated Database Tasks" chapter of Objectivity/DB Administration.
2. Design your database schema and create corresponding ODMG schema files containing declarations for all classes used in your application.
   To use the ODMG time and date classes, you must include installDir/arch/include/ooTime.h in your DDL schema files and source code .c files that use these classes.
3. Process your schema file using the DDL processor. See the "Processing Data Models" chapter in Using Objectivity/ C++ Data Definition Language.
4. Write the source code for your application using the ODMG classes and types supported Objectivity/DB.
   You must include the schema header file generated by the DDL processor in your application source code.
5. Use the oonewdb tool to create an Objectivity/DB database. For more information, see the "Database Tasks" chapter of Objectivity/ DB Administration.
6. Set the environment variable `OO_DB_NAME` to be the name of the Objectivity/DB database you created in step 5. This is not the same as the ODMG database you created in step 1.

Objects you create are stored in the Objectivity/DB database, which is itself stored within the ODMG database (the Objectivity/DB federated database). If you do not set this environment variable, the database `ooDefaultDB` is used.

7. Compile your C++ application source code file and the schema source file generated by the DDL processor.

8. Link your compiled code with Objectivity/DB runtime libraries and the compiled schema code. See “Application Development Overview” on page 1-6.

**Example**

This section presents a simple example that uses the Objectivity/DB ODMG interface.

---

**Example**

This example uses two persistent-capable classes—`Person` and `Phone` to create and manage a phone list. In the DDL file, class `Person` has object references to `Phone` for home and work phone numbers. `Person` also has a reference to itself for spouse information. `Phone` has a reference to `Person` for the owner of the phone number.

The application code stores a new `Person` object and `Phone` object if person and phone strings are given as command-line arguments. If a person string is passed in, the application retrieves information about the person, including phone numbers.
The application assumes:

- An ODMG database (Objectivity/DB federated database) named \texttt{odmgFDB} was created with \texttt{oonewfd}
- The DDL processor is run on \texttt{phoneList.ddl}
- An Objectivity/DB database is created with \texttt{oonewdb}
- The \texttt{OO\_DB\_NAME} environment variable is set to \texttt{objyDB} for the Objectivity/DB database

**DDL Scheme File Code**

```c++
//
// DDL schema file phoneList.ddl
//

class Phone;

//
// class Person
//
class Person : public d\_Persistent\_Object {
public:
    // data members:
    d\_String firstName;
    d\_String lastName;
    d\_Ref\_<Person> spouse;
    d\_Ref\_<Phone> home;
    d\_Ref\_<Phone> work;
    // member functions:
    Person(const char *argArray[]);
    void print() const;
};
```
// class Phone
//
class Phone : public d_Persistent_Object {
public:
    // data members:
    d_ULong number;
    d_Ref<Person> owner;
    // member functions:
    Phone(d_ULong theNumber, d_Ref<Person> theOwner);
    void print(const char *label = "phone") const;
};

Application Source Code
//
// Application code source file phoneList.C
//

#include <phoneList.h>

#include <iostream.h>

extern "C" {
    extern
    long
    atol();
}

// Constructor for Person
//
void Person::Person(const char *argArray[]) :
    firstName(argArray[2]),
    lastName(argArray[1]),
    home(new(this) Phone(atol(argArray[3]), this)),
    work(new(this) Phone(atol(argArray[4]), this))
{
    // Person::Person

Using the ODMG Interface 23-7
//
// print member function for Person
//
Person::print() const
{
    cout << "name: " << this->lastName << ", " <<
    this->firstName << endl;
    this->home->print("home");
    this->work->print("work");
    if (this->spouse) {
        cout << "(Married.)" << endl;
    }
} // Person::print

//
// Constructor for Phone
//
void Phone::Phone(d_ULong theNumber, d_Ref<Person> theOwner) :
    number(theNumber),
    owner(theOwner)
{
} // Phone::Phone

//
// print member function for Phone
//
Phone::print(const char *label) const
{
    cout << label << ": " << (this->number) / 10000
     << "-" << (this->number) % 10000 << endl;
} // Phone::print
// createPerson function
//
static void createPerson(const char *argArray[]) {

    // ODMG database (Objectivity/DB federated database)
    d_Database odmgFDB;

    // open ODMG database
    odmgFDB.open("odmgFDB.boot");
    d_Transaction trans;
    trans.begin();
    d_Ref<Person> person = new Person(argArray);
    // set name
    odmgFDB.set_object_name(person, name);
    trans.commit();
    odmgFDB.close();
} // createPerson

//
// displayPerson function
//
static void displayPerson(const char *name) {

    d_Database odmgFDB; // ODMG database

    // open ODMG database (Objectivity/DB federated database)
    odmgFDB.open("odmgFDB.boot", d_Database::read_only);
    d_Transaction trans;
    trans.begin();
    d_Ref<Person> person = odmgFDB.lookup_object(name);
    person.print();
    trans.commit();
    odmgFDB.close();
} // displayPerson
// main program
int main(unsigned int numOfArgs, const char *argArray[]) {
    if (numOfArgs > 2) {
        createPerson(argArray);
    } else {
        displayPerson(argArray[1]);
    }
    return 0;
} // main
Administration Programming Tasks

This chapter describes how to use the C++ programming interface to perform administration tasks. Use this chapter in conjunction with information presented in Objectivity/DB Administration.

Where C++ syntax and parameter information is identical for the object reference and handle classes, `ooRefHandle` is used to represent either `ooRef` or `ooHandle`.

Using the C++ Administration Interface

Most of the C++ administration interfaces described in this appendix are in the standard Objectivity/DB library. To use these functions, link your application with this library. However, if you are using the recovery functions (`ooCleanup`, `ooGetActiveTrans`, or `ooGetResourceOwners`) on UNIX or VMS, you must link your application with the Objectivity/DB administration library before linking with the standard Objectivity/DB library (see your Installation and Platform Notes). Note that, for Windows platforms, all administration interfaces are in the standard Objectivity/DB library.

Parameters are shown in this appendix as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`. 
Federated Database Tasks

Changing Federated Database Attributes

You can change the following attributes of a federated database:

- Federated database file host
- Federated database file path
- Lock server host
- Journal directory host
- Journal directory path
- Federated database identifier
- Boot file path

You cannot change the system name of the federated database or its page size. After changing federated database attributes, the old boot file remains. You must delete this file manually.

Use the `ooRefHandle(ooFDObj)::change` member function to change attributes of a federated database. See page K-39.

When using the `ooRefHandle(ooFDObj)::change` member function to modify the attributes of a federated database, you must strictly follow these steps:

1. Shut down the lock server.
2. Run in single-user mode (call `ooNoLock` in your program).
3. Start a transaction.
4. Open the federated database for update.
5. Invoke the `ooRefHandle(ooFDObj)::change` member function.
6. Commit the transaction.
7. Exit from the process.
8. Restart the lock server.

After running `ooRefHandle(ooFDObj)::change`, federated database attributes are in a new and different state. To avoid inconsistencies when accessing the federated database, you must exit from the process immediately after changing its attributes.
Examples

Assume the following handle declaration:

```c
ooHandle(ooFDObj) fdH;
...
```

// fdH is a valid handle to a federated database

The following examples show how to use the `ooHandle(ooFDObj)::change` member function to update the federated database attributes.

- This code changes the federated database identifier to 101. The lock server host name is not changed.
  ```c
  fdH.change(0, 0, 101);
  ```

- This code changes the lock server host name to `myHost`:
  ```c
  fdH.change(0, "myHost");
  ```

The following code uses the `ooHandle(ooFDObj)::change` member function to change the host name that is running the lock server process to `moon`.

```c
ooTrans trans;
ooHandle(ooFDObj) fdH;
...
ooInit();
ooNoLock();
trans.start();
fdH.open("Documentation", oocUpdate);
fdH.change(0, "moon"); // change the lock server
trans.commit();
exit(0);
```
Encoding and Decoding Schema

You can encode your schema to prevent users from viewing proprietary data structures. Conversely, you can decode a protected schema to make it available to users.

Use the member function ooRefHandle(ooFDObj)::encodeSchema to encode the contents of the schema of a federated database. See page K-44. You must use this member function exclusively; that is, you cannot invoke any other member function within the same transaction.

To invoke ooRefHandle(ooFDObj)::encodeSchema, you must do the following:
1. Start a transaction.
2. Open the federated database.
3. Invoke encodeSchema using a password. The password secures and protects the schema so that it cannot be viewed through Objectivity/DB tools.
4. Immediately commit the transaction.
5. Exit the process.

⚠️ Warning

Be sure to record your password in a safe place. If you cannot remember the password when you wish to decode the schema, Objectivity cannot help you.

Use the member function ooRefHandle(ooFDObj)::decodeSchema to decode the contents of the schema of a federated database. See page K-42. You must use this member function exclusively; that is, you cannot invoke any other member function within the same transaction.

To invoke ooRefHandle(ooFDObj)::decodeSchema, you must do the following:
1. Start a transaction.
2. Open the federated database.
3. Invoke decodeSchema using the password created when the schema was encoded.
4. Immediately commit the transaction.
5. Exit the process.
Getting Federated Database Information

You can get federated database information including its attributes and files.

Use the administration C++ programming interface to obtain information about a federated database, including checking the validity of its identifier, listing all of its attributes, and getting its lock server host name, identifier, system name, and page size individually.

The string returned by some of the following member functions is statically allocated and overwritten with each invocation. You should always make a local copy of the returned string if you intend to use it later in your application.

Checking for Valid Federated Database Identifier

Use the ooRefHandle(ooFDObj)::isValid member function to check whether or not the specified federated database identifier is valid. See page K-45.

Getting the Lock Server Host Name

Use the ooRefHandle(ooFDObj)::lockServerName member function to retrieve the name of the host running the lock server process for the federated database. See page K-46.

Getting the Federated Database Identifier

Use the ooRefHandle(ooFDObj)::number member function to retrieve the identifier of the federated database. See page K-46.

Getting the Federated Database System Name

Use the ooRefHandle(ooFDObj)::name member function to retrieve the system name of the federated database. See page K-46.

Getting the Page Size of the Federated Database

Use the ooRefHandle(ooFDObj)::pageSize member function to obtain the page size of the federated database. See page K-48.

Listing Federated Database Attributes

Use the ooRefHandle(ooFDObj)::dumpCatalog member function to print out federated database attributes from within your application. See page K-43.
Example

```c
ooHandle(ooFDObj) fdH;
FILE* fp;

fp = fopen("catalog.inf", "w");

// Print the attributes on the standard output
fdH.dumpCatalog();

// Print file names in host-local format and with labels
// in the file catalog.inf.
fdH.dumpCatalog(fp, oocHostLocal, oocTrue);

// Print the attributes in the standard output
// with file names in native format.
fdH.dumpCatalog(stdout, oocNative, oocFalse);
```

Getting Transaction Information

You can use the following tasks to get information about active and waiting transactions.

All Active Transactions

To return a list of active transactions, use the `ooGetActiveTrans` function. See page B-6.

Specific Transaction

To obtain the resource and list of transactions for which the specified transaction is waiting, use the `ooGetResourceOwners` function on a specific transaction. See page B-8.
Tidying a Federated Database

Tidying a federated database consolidates data that has become fragmented over time. When tidying a federated database, Objectivity/DB tries to obtain an update lock on each database in the federated database. If it fails to obtain the lock for a particular database it does not tidy it, and continues on to the next database.

Since the tidy operation creates temporary database files while it is running, it requires free disk equal to the size of the databases you are tidying.

To tidy a federated database, use the `ooRefHandle(ooFDObj)::tidy` member function. See page K-49. When using this member function, you must follow these steps:

1. Start a transaction.
2. Open the federated database for update.
3. Invoke the `ooRefHandle(ooFDObj)::tidy` member function
4. Immediately commit the transaction.

You cannot manipulate any database, container, or basic object within the transaction before invoking the `ooRefHandle(ooFDObj)::tidy` member function. This member function performs physical storage compaction and relocation. After calling this function, some of the cached system data is in an inconsistent state. It is important that you immediately commit the transaction so that the system data can be reinitialized to a known state.

You cannot abort a transaction after calling the `ooRefHandle(ooFDObj)::tidy` member function.

---

**Example**

The following code starts a transaction, opens a federated database for update, invokes the `tidy` member function, and then commits the transaction:

```c
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooStatus status;
...
trans.start();
status = fdH.open("Documentation", oocUpdate);
```
Database Tasks

Changing Database Attributes

You can use the \texttt{ooRefHandle(ooDBObj)::change} member function to change attributes of a database. See page K-28. If you are using the Objectivity/DB Data Replication Option, note that you can only use this member function to change databases that have one image (that is, those databases that have not been replicated). See Using Objectivity/FTO and Objectivity/DRO for more information.

Use the \texttt{ooRefHandle(ooDBObj)::change} member function if you want to rename or physically move a database file to a new location. Objectivity/DB tries to get an exclusive update lock on the corresponding database when the member function is invoked. If it cannot obtain the lock, the member function returns an error status.

If Objectivity/DB obtains the exclusive update lock, the member function \texttt{ooRefHandle(ooDBObj)::change} will logically rename or relocate the database file within the federated database. However, you must then physically move or rename the file.

For example, after invoking the \texttt{ooRefHandle(ooDBObj)::change} member function, you should rename or relocate the actual database file using the operating system command to move a file.

if (status == oocSuccess) {
    status = fdH.tidy();
    if (status == oocSuccess) {
        trans.commit();
    }
}
Example

The following code uses the `ooHandle(ooDBObj)::change` member function to set the database file host name to `myHost` and the local path name of the database file to `/mnt/john/design/adder.ecad.DB`.

```cpp
ooHandle(ooDBObj) dbH;
...
// dbH is a valid handle to a database
// Set the new host name of the database to be myHost and
// new local path name to be /mnt/john/design/adder.ecad.DB
dbH.change(0, "myHost", "/mnt/john/design/adder.ecad.DB");
```

Getting Database Information

You can obtain information about the attributes of a database. If you are using the Objectivity/DB Data Replication Option, note that you can only use these member functions to get information on databases that have one image (that is, those databases that have not been replicated). See Using Objectivity/FTO and Objectivity/ DRO for more information.

Getting the Database Host Name

Use the `ooRefHandle(ooDBObj)::hostName` member function to return a string containing the name of the network host where the database file is located. See page K-32.

Getting the Database Path Name

Use the `ooRefHandle(ooDBObj)::pathName` member function to return a string containing the full path name of the directory where the database file is located. See page K-35.
Getting the Database File Name

Use the `ooRefHandle(ooDBObj)::fileName` member function to return a string containing the full file name of the database file (path and filename). See page K-31.

Example

In this example, the handle `dbH` is set to reference the database with system name `testDb` located in the federated database with system name `testFd`. Information about the corresponding database file is printed out.

```c
ooHandle(ooDBObj) dbH;
ooHandle(ooFDObj) fdH;
...
// dbH is set to reference database with system name testDb
fdH.open("testFd", oocRead);
dbH.open(fdH, "testDb", oocRead);

// Get and print out info on database file
printf("hostname: %s\n", dbH.hostName());
printf("pathname: %s\n", dbH.pathName());
printf("filename: %s\n", dbH.fileName());
```

Moving a Database File

You can move a database file from one location to another on a network. If you are using the Objectivity/DB Data Replication Option, note that you can only use this member function to move databases that have one image (that is, those databases that have not been replicated). See Using Objectivity/FTO and Objectivity/DRO for more information.

To move a database file from one host machine to another, use `ooRefHandle(ooDBObj)::change`. See page K-28. This member function logically renames or relocates the database file within the federated database. However, you must then physically move or rename the file using the operating system command to move a file.
Tidying a Database

Tidying a database consolidates data that has become fragmented over time. The tidy operation creates a temporary database file while it is running. Therefore it requires twice as much space as the database you are tidying.

To tidy a specific database, use the `ooRefHandle(ooDBObj)::tidy` member function. See page K-35. When using this member function, you must follow these steps:

1. Start a transaction.
2. Open the database for update.
3. Invoke the `ooRefHandle(ooDBObj)::tidy` member function.
4. Immediately commit the transaction.

You cannot manipulate any database, container, or basic object within the transaction before invoking the `ooRefHandle(ooDBObj)::tidy` member function. This member function performs physical storage compaction and relocation. After calling this function, some of the cached system data is in an inconsistent state, so it is important that you immediately commit the transaction so that the system data can be reinitialized to a known state.

You cannot abort a transaction after calling the `ooRefHandle(ooDBObj)::tidy` member function.

---

**Example**

The following code starts a transaction, opens a federated database for update, opens a database for update, calls the `ooHandle(ooDBObj)::tidy` member function, and then commits the transaction:

```c++
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;
...
trans.start();
fdH.open("Documentation", oocUpdate);
dbH.open(fdH, "Introduction", oocUpdate);
dbH.tidy();
```
AMS Tasks

By default, Objectivity/DB uses the Advanced Multithreaded Server (AMS) feature if it is available. Use the following information, if you wish to explicitly set up how an application uses AMS.

Setting AMS Usage in an Objectivity/C++ Application

To specify AMS usage for an application, perform the following step:

1. Use the `ooSetAMSUsage` function after `ooInit` and before any other Objectivity/DB function. See page B-15. Include one of the enumerated types specified in `ooAMSUsage` on page A-1:

- `oocAMSPreferred`: Use AMS for remote data access if it is available
- `oocAMSOnly`: Use AMS exclusively
- `oocNoAMS`: Never use AMS

The default setting is `oocAMSPreferred`. If you specify `oocAMSOnly`, but AMS is unavailable on the host that contains the data you are trying to access, an error is signalled.
Automatic and Manual Recovery Tasks

The following are automatic and manual recovery tasks you can perform using the administration programming interface.

Enabling Automatic Recovery from Application Failures

You can set up an application to open federated databases with automatic recovery enabled (see the steps described in the following subsections). When such an application opens a federated database, Objectivity/DB rolls back any incomplete local transactions for that federated database. (Local transactions are transactions that were started by applications running on the same client host as your recovery-enabled application.) For more information on recovery tasks, see the “Automatic and Manual Recovery” chapter in Objectivity/DB Administration.

To enable automatic recovery from C++ application failures:

1. Call the `ooRefHandle(ooFDObj)::open` member function with the `recover` parameter set to `oocTrue` when opening a federated database for the first time in an application. See page K-47. For performance reasons, you may want to do this only one time in your application.

Creating a Recovery Application

You can create your own recovery tool by writing a C++ database application that either:

- Opens a federated database and activates the automatic recovery option. See “Enabling Automatic Recovery from Application Failures” above.
- Calls the administrative recovery functions `ooGetResourceOwners`, `ooGetActiveTrans`, and `ooCleanup`. See pages B-8, B-6, and B-2.
WARNING

Do not call the ooInit function, or any other non-recovery Objectivity/DB functions, from within a recovery application process. If you need to call a non-recovery function, create a separate execution environment using a system call.

Follow these guidelines when creating a recovery tool using the C++ interface:

1. Design your recovery application using one or more of the recovery interface functions. Do not call ooInit, or any other non-recovery Objectivity/DB functions within this application. Also note that ooCleanup does not clean up a transaction if the process owning that transaction is still active, or if another process is running that has the same process identifier.

2. Include the header file ooRecover.h in the application code. See “ooRecover.h Data Structures” on page 24-14.

3. Link your application with the administration library before linking with other Objectivity/DB libraries. See Installation and Platform Notes for information on linking with Objectivity/DB libraries.

**ooRecover.h Data Structures**

The data structures defined in ooRecover.h are as follows:

```c
// transaction identifier type
typedef unsigned long ooTransId;

// transaction information
typedef struct ooTransInfo
{
    ooTransId tid;          // transaction identifier
    char host[32];          // host name
    char arch[32];          // architecture type
    unsigned int uid;       // user identifier
    unsigned int pid;       // process identifier
} ooTransInfo;
```

**24-14 Using Objectivity/C++**
/ resource information
typedef struct ooResource
{
    char type[8];               // resource type
    char mode[8];              // lock mode
    unsigned int fdId;         // federated database identifier
    unsigned int dbId;         // database identifier
    unsigned int ocId;         // lock server identifier
    unsigned int pgId;         // page identifier
} ooResource;

Example Recovery Application
The following examples gets a list of all active transactions and recovers them. It does not check to see if a given user is active, and therefore, may clean up a currently running transaction. Because of this, you must be sure that the federated database is not currently being used before running this application.

Example

#include <stdio.h>
#include "ooRecover.h"

void main()
{
    char *bootfilepath = "EXAMPLE";  // path to the boot file

    ooTransInfo *activeTrans;
    ooStatus stat;
/ get the List of active transactions
ooGetActiveTrans(&activeTrans, // pointer to list
    &bootfilepath, // federated database
    0, // list transactions on all nodes
    0); // list transactions by all users

// for all active transactions
while(activeTrans->tid != oocInValidTransId) {
    // cleanup up the transaction
    stat = ooCleanup(&bootfilepath, // federated database
        activeTrans->tid, // transaction to clean
        1, // clean up transactions from any host
        0, // use the lock server
        0, // no reset lock needed
        0); // fail if another cleanup is running

    // Verify that cleanup worked
    if (stat)
        printf("Cleaned up transaction id %d \n",
            activeTrans->tid);
    else
        printf("Could NOT cleanup transaction id %d \n",
            activeTrans->tid);

    activeTrans++;
}

//
Recovery Application Error Conditions

Specific error numbers for a given recovery function are shown in Table 24-5. Table 24-6 lists all recovery interface error numbers and messages.

### Table 24-5: Recovery Function Error Numbers

<table>
<thead>
<tr>
<th>Recovery Function</th>
<th>Error Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ooGetResourceOwners</td>
<td>201, 202, 208, 212</td>
</tr>
<tr>
<td>ooGetActiveTrans</td>
<td>201, 202, 203, 204, 205, 206, 208, 209, 210, 211, 212, 213, 214, 215</td>
</tr>
<tr>
<td>ooCleanup</td>
<td>201, 202, 203, 204, 205, 206, 208, 209, 210, 211, 212, 213, 214, 215</td>
</tr>
</tbody>
</table>

### Table 24-6: Recovery Interface Error Messages

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>Fail to initialize the database service</td>
</tr>
<tr>
<td>202</td>
<td>Fail to allocate more heap space</td>
</tr>
<tr>
<td>203</td>
<td>Fail to find the specified transaction</td>
</tr>
<tr>
<td>204</td>
<td>Transaction is started by a remote host</td>
</tr>
<tr>
<td>205</td>
<td>Fail to recover the transaction</td>
</tr>
<tr>
<td>206</td>
<td>Fail to get/reset recovery lock on the transaction</td>
</tr>
<tr>
<td>208</td>
<td>Fail to read the federated database boot file information</td>
</tr>
<tr>
<td>210</td>
<td>Process owning the transaction is active</td>
</tr>
<tr>
<td>211</td>
<td>No privilege to check the transaction process status</td>
</tr>
<tr>
<td>212</td>
<td>Fail to connect to the lock server</td>
</tr>
<tr>
<td>213</td>
<td>Standalone mode fails, the lock server is active</td>
</tr>
</tbody>
</table>
### Table 24-6: Recovery Interface Error Messages (Continued)

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>214</td>
<td>Fail to read the journal file header</td>
</tr>
<tr>
<td>215</td>
<td>Fail to read the directory containing journal files</td>
</tr>
<tr>
<td>216</td>
<td>Unknown recovery error</td>
</tr>
</tbody>
</table>
Using the Cursor Class

This chapter describes the cursor class provided with Objectivity/C++.

Many GUI-driven applications use buffers to reference objects displayed in tables or scroll menus. The Objectivity/DB cursor class implements such a buffer as a parameterized class and allows you to:

◆ Maintain an arbitrary set of instances of a class className or its subclasses
◆ Populate this set of objects using Objectivity/DB iterators
◆ Iterate forward and backward through all objects contained in this set
◆ Pin a subset of objects within this set in memory to achieve maximum access performance
◆ Add a callback function for GUI operations

Architecture

A cursor object maintains a VArray of ooRef(className), referred to as the cursor VArray. You specify an initial size and a growth factor for this VArray during the construction of the cursor.

In addition, the cursor maintains a fixed-sized array of ooHandle(className) referred to as the visible area. These handles are used to pin all objects being currently displayed in the visible area of a table or a scroll menu in memory. The size of this array can be specified during the construction as well.

A cursor is able to maintain an iterator referred to as the cursor iterator, which you can use to populate the cursor array both initially and incrementally.

Alternatively, you can initialize a SQL query to populate the cursor VArray. This query may be executed on a server; only the OIDs of the objects found in the query are shipped to the client application.

The arrays maintained by the cursor can be resized and modified, allowing addition and removal of table rows or scroll menu entries.
Figure 25-1 shows the internal architecture of the cursor.
Usage

Figure 25-2 shows the life cycle of a cursor.

![Diagram of cursor life cycle](image)

**Figure 25-2  Cursor Life Cycle**

Including the Header File

To use the parameterized cursor class, you must include the `ooCursor.h` header file in your application.
Class ooCursor

Declaring a Cursor

Before you can use a cursor of class className, you must first use the C++ declare macro to declare the parameterized class needed. The declare macro has the following syntax:

- `declare(ooCursor, className);`

  where

  `className` Element class of the cursor.

Implementing Cursors

Before you can access a cursor of a particular class, you must use the C++ implement macro to generate code needed for the parameterized class in only one of your source files. Use this macro outside of member function and function definitions. The syntax of the implement macro is as follows:

- `implement(ooCursor, className)`

  where

  `className` Element class of the cursor.

ooCursor Constructors

The ooCursor constructor constructs a new cursor object:

- `ooCursor(className)::ooCursor(className)(` `const uint32 visibleSize,` `const uint32 initSize,` `const uint32 percentGrow);`

  where

  `visibleSize` Size of the visible area that contains all handles pinning a subset of the objects maintained by the cursor in memory.
  `initSize` Initial size of the cursor VArray that contains references to all objects maintained by the cursor.
  `percentGrow` Percentage growth factor for the cursor VArray.
Initializing the Cursor

The member function `ooCursor(className)::init` initializes the cursor to be populated with all objects retrieved in a scan over the object `scope` using the predicate `pred`.

```c++
virtual ooStatus init(
    const ooHandle(ooObj) &scope,
    const char *pred);
```

where

- `scope` Scope for the initialization scan query. `scope` has to be an `ooHandle(ooFDObj)`, an `ooHandle(ooDBObj)`, or an `ooHandle(ooContObj)`.
- `pred` Predicate used for the scan iteration.

The member function `ooCursor(className)::init` initializes the cursor to be populated with all objects retrieved in an iteration using `itr`.

```c++
virtual ooStatus ooCursor(className)::init(ooItr(X) &itr);
```

where

- `itr` Initialized iterator used to populate the cursor.

The iterator `itr` may not be deleted (neither explicitly nor by leaving a definition scope) until the iteration is complete.

The member function `ooCursor(className)::init` can be used to populate the cursor `VArray` with the result of a server side SQL query.

```c++
virtual ooStatus ooCursor(className)::init(
    const char *query,
    const char *server,
    const char *usr,
    const char *password);
```

where

- `query` SQL statement.
- `server` Name of a boot file of a federated database that is accessed by the query.
- `user` User name for the SQL query.
- `password` Password for the SQL query.
The SQL statement \textit{query} must have the form:

\begin{verbatim}
SELECT OID ...
\end{verbatim}

Using the server side SQL query involves running two processes against a federated database. Serializability conflicts may occur in this scenario.

For all initialization member functions, only \textit{initSize} (according to the constructor) iteration steps are performed and the cursor VArray is populated with the iteration result. The visible area is set to the first \textit{visibleSize} entries of the cursor VArray.

If the iteration has less then \textit{initSize} steps, remaining entries in both arrays are initialized with zero.

\section*{Retrieving the Cursor Size}

The member function \texttt{ooCursor(className)::getQuerySize} returns the current number of non-zero entries in the cursor VArray.

\begin{verbatim}
❑ uint32 ooCursor(className)::getQuerySize() const;
\end{verbatim}

If no entries are added or removed manually, this number is either equal to \( N \times \text{querySize} \) or the complete number of objects retrieved by the complete iteration with the cursor iterator.

\section*{Checking for Query Completeness}

You can use the member function \texttt{ooCursor(className)::isQueryComplete} to check whether the iteration with the cursor iterator, or the server side SQL query, is complete.

\begin{verbatim}
❑ ooBoolean ooCursor(className)::isQueryComplete() const;
\end{verbatim}

\section*{Retrieving Objects from a Cursor}

The member function \texttt{ooCursor(className)::getEntry} is used to retrieve the object at position \textit{index} in the visible area. If the index exceeds the size of the handle array, \texttt{NULL} is returned and an error is raised.

\begin{verbatim}
❑ ooHandle(className) ooCursor(className)::getEntry(
  const uint32 index)
\end{verbatim}

where

\begin{verbatim}
index Relative index of the fixed-size array.
\end{verbatim}
The member function `ooCursor(className)::getAbsolutEntry(const uint32 index)` is used to retrieve the object at position `index` in the cursor VArray.

- `ooHandle(className) ooCursor(className)::getAbsolutEntry(
    const uint32 index)`

  where
  
  `index`  
  Absolute index of the VArray.

  If `index` exceeds the current size of the cursor VArray, the VArray is resized and the iteration continues until the iteration is complete or the cursor VArray entry with the index `index` is initialized.

The member operator `ooCursor(className)::operator[]` is the operator version of the member function `ooCursor(className)::getAbsolutEntry`.

- `ooHandle(className) ooCursor(className)::operator[](const uint32 index);`

  where
  
  `index`  
  Absolute index of the VArray.

Resizing the Cursor

The member function `ooCursor(className)::resize` resizes the cursor VArray.

- `ooStatus ooCursor(className)::resize()`

  where
  
  `newSize`  
  New size of the VArray.

  If `newSize` is greater than the current size, the newly created array fields are initialized with zero. If `newSize` is less than the current size, the remaining object references are removed.

Appending Objects

The member function `ooCursor(className)::addEntry` can be use to append an object at the end of the cursor VArray.

- `ooStatus ooCursor(className)::addEntry(
    const ooHandle(const ooHandle(className) &newObj);`

Using the Cursor Class 25-7
Inserting Objects

where

\( \text{newObj} \) Handle that references the new element.

If necessary, the cursor VArray is resized during this operation.

**Inserting Objects**

The member function \texttt{ooCursor(className)::insEntry} can be used to insert an object referenced by \texttt{newObj} at the position \texttt{index} of the cursor VArray. If necessary, the cursor VArray is resized.

\[
\text{ooStatus ooCursor(className)::insEntry(
\quad \text{const ooHandle(const ooHandle(className) & newObj,}
\quad \text{const uint32 index);}
\]

where

\( \text{newObj} \) Handle that references the new element.

\( \text{index} \) Position in the cursor VArray, where the new object will be inserted.

**Deleting Objects**

The member function \texttt{ooCursor(className)::delEntry} can be used to delete the entry at position \texttt{index} in the cursor VArray.

\[
\text{ooStatus ooCursor(className)::delEntry(
\quad \text{const uint32 index})}
\]

where

\( \text{index} \) Position in the cursor VArray, which is deleted.

The visible area is adjusted accordingly.

You can use the member function \texttt{ooCursor(className)::delEntry} to remove a reference to the object referenced by \texttt{hnd} from the cursor VArray. If there is no entry referencing this object, the operation is ignored.
Moving the Visible Area

- ooStatus ooCursor(className)::delEntry(
  const ooHandle(className) & hnd);

where

  hnd          Handle to an object reference by an entry of the cursor VArray.

The visible area is adjusted accordingly.

Moving the Visible Area

You can use the member function ooCursor(className)::next to move forward the visible area of the cursor over the cursor VArray.

- ooBoolean ooCursor(className)::next(const uint32 step = 1);

where

  step        Number of steps to be performed.

You can use the member function ooCursor(className)::prev to move backward the visible area of the cursor over the cursor VArray.

- ooBoolean ooCursor(className)::prev(
  const uint32 step = 1);

where

  step        Number of steps to be performed.

The member function ooCursor(className)::first moves the visible area to the beginning of the cursor VArray.

- ooBoolean ooCursor(className)::first();

The member function ooCursor(className)::last moves the visible area to the end of the cursor VArray.

- ooBoolean ooCursor(className)::last();
**Sorting the Cursor**

You can use the member function `ooCursor(className)::sort` to sort the current content of the cursor `VArray`.

\[
\text{ooStatus ooCursor(className)::sort(int (*ordFunc)(const ooHandle(className) &, const ooHandle(className)));
\]

where

\*ordFunc Ordering function used for the sort.

**Registering a Callback Function**

You can use the member function `ooCursor(className)::registerCallback` to register a callback function to the cursor.

\[
\text{ooStatus ooCursor(className)::registerCallback( ooCursorFncPtr *fnc);
}\]

where

\*fnc Pointer to the callback function.

This callback function is called whenever the `next()` member function is called for the cursor iterator.

This callback function may be used to implement communication with GUI progress indicators. For example, you can use it to display the number of read objects.

The type `ooCursorFncPtr` is defined as:

\[
\text{ooBoolean (*)(uint32 itrStep)}
\]

where

\*itrStep Current number of iteration steps performed with the cursor iterator.

If the return value of the callback function is `ooFalse`, the current iteration will be stopped.
The following is an example of using the cursor class

declare(ooCursor, Person)
...
implement(ooCursor, Person)
...

ooBoolean Callback(const uint13 Num)
{
    cout << "Object No. " << Num << " read." << endl;
    cout << "Continue (Y/N)? " << endl;
    cin << Buff;

    if (Buff == 'N')
    {
        return oocFalse;
    }
    else
    {
        return oocTrue;
    }
}

ooCursor(Person) PersonCursor(10,20,10);

    // Register the callback
PersonCursor.registerCallback(Callback);

    // Initialize the cursor to scan all persons in
    // the federated database FD whose age is greater
    // than 18
PersonCursor.init(FD, "Age > 18");
/ Reset the callback

PersonCursor.registerCallback(NULL);

uint32 N = 0;

while (!PersonCursor.isQueryComplete())
{
    PersonCursor.getAbsolutEntry(N)->print();
    N++;
}
Types

This appendix describes Objectivity/C++ types, as well as parameterized types and classes.

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ooAccessMode</td>
<td>Data member access mode type for predicate queries.</td>
</tr>
</tbody>
</table>

**SYNTAX**

```cpp
enum ooAccessMode {
    oocPublic,
    oocAll
};
```

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ooAMSUsage</td>
<td>(for administration) This is an enumerated type for use with the ooSetAMSUsage function. Usage can be oocAMSPreferred (use if available; the default usage), oocAMSOnly (always used), or oocNoAMS (never used).</td>
</tr>
</tbody>
</table>

**SYNTAX**

```cpp
enum ooAMSUsage {
    oocAMSPreferred,
    oocAMSOnly,
    oocNoAMS
};
```
**ooBoolean**

**DESCRIPTION**
Boolean type.

**SYNTAX**
```c
typedef uint8 ooBoolean;
```

---

**ooDataType**

**DESCRIPTION**
The enumerated type used by user-defined relational operators.

**SYNTAX**
```c
enum ooDataType {
    oocInt32T,
    oocUint32T,
    oocFloat64T,
    oocCharPtrT,
    oocBooleanT,
    oocInvalidTypeT
};
```

where
- `oocInt32T` Objectivity/DB type `int32`
- `oocUint32T` Objectivity/DB type `uint32`
- `oocFloat64T` Objectivity/DB type `float64`
- `oocCharPtrT` Objectivity/DB type `char *`

---

**ooDowngradeMode**

**DESCRIPTION**
The enumerated type used by `ooTrans::commitAndHold` to preserve update locks or downgrade them to read locks.

**SYNTAX**
```c
enum ooDownGrade {
    oocNoDowngrade = 0,
    oocDownGradeAll
};
```
<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>DESCRIPTION</th>
<th>SYNTAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ooError</td>
<td>Error type, associates an error number with a text string.</td>
<td>struct ooError {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>uint32 errorN;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *message;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>};</td>
</tr>
<tr>
<td>ooErrorHandlerPtr</td>
<td>Error handler function pointer type.</td>
<td>typedef ooStatus (*ooErrorHandlerPtr)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ooErrorLevel,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ooError&amp;,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ooHandle(ooObj) *,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *);</td>
</tr>
<tr>
<td>ooErrorLevel</td>
<td>Error level type. Used in conjunction with the ooSignal function, it</td>
<td>enum ooErrorLevel {</td>
</tr>
<tr>
<td></td>
<td>indicates the severity of a particular error.</td>
<td>oocNoError,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oocWarning,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oocUserError,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oocSystemError,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oocFatalError</td>
</tr>
<tr>
<td></td>
<td></td>
<td>};</td>
</tr>
</tbody>
</table>
### ooFileNameFormat

**TYPE NAME**  ooFileNameFormat  
**DESCRIPTION**  Describes how to format file names for output.  
- oocNative specifies a file name format as full path names; for example, /net/mach3/usr/mnt/project/myfd.FDDB.  
- oocHostLocal specifies a file name format as hostName:localPath; for example, mach3:/mnt/fred/project/myfd.FDDB.  
**SYNTAX**  
```
enum ooFileNameFormat {  
oocNative,  
oocHostLocal  
};
```

### ooIndexMode

**TYPE NAME**  ooIndexMode  
**DESCRIPTION**  Specifies the index mode.  
- oocInsensitive specifies that updates to indexed fields are done at commit time.  
- oocSensitive specifies that updates to indexed fields are immediately and automatically reflected in the corresponding indexes.  
**SYNTAX**  
```
enum ooIndexMode {  
oocInsensitive,  
oocSensitive  
};
```

### ooKey

**TYPE NAME**  ooKey  
**DESCRIPTION**  Keyed Object key structure. Provides the Keyed Object facility with necessary key information to create and look up Keyed Objects.  
**SYNTAX**  
```
enum ooKey type;  
    uint32 offset;  
    uint32 size;  
    ooVoidPtr value  
};
```
<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>ooKeyType</th>
<th>DESCRIPTION</th>
<th>SYNTAX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Specifies the type of the key field within a class. Used with the ooKey</td>
<td>enum ooKeyType { oocUint16, oocUint32, oocInt16, oocInt32, oocFloat32, oocFloat64, oocString,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>structure.</td>
<td>oocCharArray }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ooLockMode</td>
<td>Specifies mode in which to lock an object.</td>
<td>enum ooLockMode { oocLockRead, oocLockUpdate }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ooMode</td>
<td>Specifies the access mode in which to open a persistent object.</td>
<td>enum ooMode { oocNoOpen, oocRead, oocUpdate }</td>
</tr>
<tr>
<td>TYPE NAME</td>
<td>Description</td>
<td>Syntax</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------</td>
<td>---------------------------------------------</td>
<td></td>
</tr>
</tbody>
</table>
| ooMsgHandlerPtr | Message Handler function pointer type.                | `typedef void (*ooMsgHandlerPtr)(
  char `*`);`                           |
| ooNameHashFuncPtr | Pointer to the hash function.                        | `typedef uint32 (*ooNameHashFuncPtr)(
  const char `*`,
  const uint32);`                 |
<p>| ooStatus     | General function/member function return type. Indicates whether an error occurred or not. | <code>typedef int32 ooStatus;</code>                |
|              | CONSTANTS                                             |                                             |
|              | oocSuccess                                            |                                             |
|              | oocError                                              |                                             |
|              | RETURN VALUES                                         | <code>oocSuccess</code> if successful and <code>oocError</code> if unsuccessful. |
| ooThreadId   | Type used to identify a thread to Objectivity/C++.    | <code>typedef uint32 ooThreadId;</code>               |</p>
<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>ooVoidFuncPtr</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Pointer to a function with no parameters and that returns no result.</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>typedef void (*ooVoidFuncPtr)(void);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>ooTypeNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Type number of a persistent class.</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>typedef uint32 ooTypeNumber;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>ooVersMode</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Specifies the versioning status of a versionable persistent object.</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>enum ooVersMode { oocNoVers, oocLinearVers, oocBranchVers }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>ooVoidPtr</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Pointer to void type.</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>typedef void *ooVoidPtr;</td>
</tr>
</tbody>
</table>
# Parameterized Types and Classes

This section describes the Objectivity/C++ parameterized types and classes.

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ooHandle(\textit{className})</td>
<td>Declaration for an ooHandle(\textit{className}) class. Refer to Appendix K, &quot;Object Reference and Handle Classes,&quot; and Appendix P, &quot;User-Defined Object Reference and Handle Classes,&quot; for complete descriptions and syntax of system-defined and user-defined handle classes.</td>
</tr>
<tr>
<td>ooItr(\textit{className})</td>
<td>Declaration for an ooItr(\textit{className}) class. Iterator classes contain non-persistent mechanisms that allow you to traverse collections of objects. Refer to Appendix J, &quot;Iterator Classes,&quot; and Appendix O, &quot;User-Defined Iterator Classes,&quot; for complete descriptions and syntax of system-defined and user-defined iterator classes.</td>
</tr>
<tr>
<td>ooRef(\textit{className})</td>
<td>Declaration for an ooRef(\textit{className}) class. Refer to Appendix K, &quot;Object Reference and Handle Classes,&quot; and Appendix P, &quot;User-Defined Object Reference and Handle Classes,&quot; for complete descriptions and syntax of system-defined and user-defined object reference classes.</td>
</tr>
</tbody>
</table>
### ooShortRef (className)

**TYPE NAME**  | ooShortRef (className)
--- | ---
**DESCRIPTION**  | Declaration for an ooShortRef (className) class.
 | Refer to Appendix P, "User-Defined Object Reference and Handle Classes," for complete descriptions and syntax of short object reference classes.

### ooString (N)

**TYPE NAME**  | ooString (N)
--- | ---
**DESCRIPTION**  | Declaration for string class that contains a VArray of characters and a fixed character array of length \( N \). This class is non-persistent, but an instance of the class may be persistent when it is contained within a persistent object.
 | Refer to Appendix G, "String Classes," for a complete description and syntax of this class.

### ooTVArray (type)

**TYPE NAME**  | ooTVArray (type)
--- | ---
**DESCRIPTION**  | Declaration for variable-size array (VArray) class. This non-persistent class contains objects that exist only within the scope of program execution. It may not be contained within a persistent class. The parameter `className` specifies the type or class of each element within the VArray. Member functions are defined on the class to allow dynamic resizing of the VArray. The member functions defined in ooTVarray are similar to those defined in ooVArray. The semantics of resize, extend, and VArray copy are different.
 | Refer to Appendix I, "VArray Classes," for a complete description and syntax of this class.
ooVArray(\textit{type})

\begin{table}[h]
\centering
\begin{tabular}{|l|p{\textwidth}|}
\hline
\textbf{TYPE NAME} & \textbf{ooVArray (\textit{type})} \\
\hline
\textbf{DESCRIPTION} & Declaration for variable-size array (VArray) class. This class is non-persistent, but an instance of the class may be persistent when it is contained within a persistent object. Refer to Appendix I, "VArray Classes," for a complete description and syntax of this class. \\
\hline
\end{tabular}
\end{table}
Macros and Functions

This appendix describes the Objectivity/C++ macros and functions. Parameters are shown as they are declared in the Objectivity/C++ header files. Since Objectivity/C++ provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`.

About Macros

The preprocessor replaces each occurrence of a macro with the code given in its definition in the Objectivity/C++ header files. Because the macro is replaced prior to compilation, the compiler does not generate any error messages using the macro name or do any type checking on the macro arguments. The compiler sees only the code resulting from the preprocessor's expansion of the macro.

<table>
<thead>
<tr>
<th>FUNCTION NAME</th>
<th>free</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>1. int free(ooVString &amp;varString);</td>
</tr>
<tr>
<td></td>
<td>2. int free(ooString(N) &amp;varString);</td>
</tr>
<tr>
<td></td>
<td>3. int free(ooVArray(className) &amp;varString);</td>
</tr>
<tr>
<td></td>
<td>4. int free(ooTArray(className) &amp;varString);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Deallocates storage for varString by resizing it to 0 length. Same behavior as the ANSI free function.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>Returns result of varString.resize(0);</td>
</tr>
</tbody>
</table>
**FUNCTION NAME**  ooAbort  
**SYNTAX**  ooStatus ooAbort();  
**DESCRIPTION**  Aborts the default transaction oovTopTrans, discarding any modifications. The transaction is terminated and cannot be restarted.  
**RETURN VALUES**  ooStatus (oocSuccess or oocError).

**FUNCTION NAME**  ooCheckVTablePointer  
**SYNTAX**  void ooCheckVTablePointer(  
            const ooBoolean checkVptr = oocTrue);  
**DESCRIPTION**  If this function is called with the argument of oocTrue, the system will issue a warning message whenever a persistent object is accessed whose persistent class does not have a virtual table pointer. This can occur when the type information for an object cannot be found.  
**RETURN VALUES**  ooStatus (oocSuccess or oocError).

**FUNCTION NAME**  ooCleanup  
**SYNTAX**  ooStatus ooCleanup(  
            char** ppBootFilePath,  
            ooTransId tId,  
            int ignHost,  
            int standalone,  
            int resetLock,  
            ooTransInfo* pLockOwner);
where

\textit{ppBootFilePath} \hspace{1cm} \text{Pointer to the location that contains the name of the federated database file to be checked. If \textit{ppBootFilePath} is set to zero, the federated database filename is obtained from the environment variable \texttt{OO\_FD\_BOOT} and \textit{ppBootFilePath} is set to the name obtained in \texttt{OO\_FD\_BOOT}.}

\textit{tId} \hspace{1cm} \text{Transaction identifier of the transaction to recover}

\textit{ignHost} \hspace{1cm} \text{Determines whether a transaction started on another node is recovered. If the \textit{ignHost} is zero, \texttt{ooCleanup} does not recover the transaction that started on a node other than the one from which \texttt{ooCleanup} was invoked.}

\textit{standalone} \hspace{1cm} \text{If this parameter is non-zero, ensures that \texttt{ooCleanup} makes no attempt to contact the lock server and obtain locks during the recovery process.}

\textit{resetLock} \hspace{1cm} \text{If this parameter is non-zero, resets the recovery lock on the transaction and recovers the transaction, even if the recovery lock is owned by another instance of \texttt{ooCleanup}.}

\textit{pLockOwner} \hspace{1cm} \text{Points to information about a competing \texttt{ooCleanup} process. Setting this parameter to zero, prevents \textit{pLockOwner} from being updated with the competing \texttt{ooCleanup} information.}

**DESCRIPTION** (for administration) Use the \texttt{ooCleanup} function to recover a specific transaction. Unless the \textit{ignHost} parameter is set, \texttt{ooCleanup} does not recover transactions started on a node other than the one from which it is invoked. Also, \texttt{ooCleanup} does not recover a transaction if the process owning that transaction is still active.
ooCleanup attempts to get a recovery lock on the specified transaction tId before doing the recovery. If another instance of ooCleanup owns the lock, the following will occur:

- Your execution of ooCleanup will fail with a return status of zero
- The structure pointed to by the global pointer variable oovLastError will be set to a value of 206
- The pointer pLockOwner will point to information about the competing ooCleanup process.

For more information about using oovLastError, see “Error Flags” on page 19-5.

You can prevent ooCleanup from failing when it encounters another ooCleanup process by specifying the resetLock parameter.

Do not call the ooInit function (or any other non-recovery functions) in the application process since ooCleanup already initializes the cache.

RETURN VALUES
Returns a non-zero value for ooStatus if successful.

---

FUNCTION NAME  ooCommit
SYNTAX          ooStatus  ooCommit();
DESCRIPTION      Writes any objects updated during the default transaction oovTopTrans to permanent storage. The transaction is terminated and cannot be restarted.
RETURN VALUES    ooStatus (oocSuccess or oocError).
<table>
<thead>
<tr>
<th>FUNCTION NAME</th>
<th>ooCommitAndHold</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus ooCommitAndHold();</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Writes any objects updated during the default transaction oovTopTrans to permanent storage and immediately starts another transaction.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>ooStatus (oocSuccess or oocError).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FUNCTION NAME</th>
<th>ooCurrentThread</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>extern &quot;C&quot; ooThreadId ooCurrentThread();</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Returns the thread identifier to which the context is currently set (the active thread).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MACRO NAME</th>
<th>ooDelete</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus ooDelete(const ooHandle(className) &amp;objH);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Deletes the object of class className referenced by handle objH and any other objects associated via links with delete propagation specified. This operation cannot be undone (via a transaction abort) for databases or federated databases that are removed.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>Non-zero value for ooStatus if successful.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MACRO NAME</th>
<th>ooDeleteNoProp</th>
</tr>
</thead>
</table>
| SYNTAX                 | ooStatus ooDeleteNoProp(
|                        |     const ooHandle(className) &objH); |
| DESCRIPTION            | Removes the object of class className referenced by handle objH. This operation cannot be undone (via a transaction abort) for databases or federated databases that are removed. |
| RETURN VALUES          | Non-zero value for ooStatus if successful. |
FUNCTION NAME  ooGetActiveTrans

SYNTAX  

    ooStatus ooGetActiveTrans(
        ooTransInfo** ppTrans,
        char** ppBootFilePath,
        char* pHost,
        unsigned int* pUid);

where

- **ppTrans**  Pointer to the location where a pointer to a list (array) of active transactions will be placed. The list is allocated by ooGetActiveTrans and terminated by an entry whose transaction identifier is set to oocInValidTransId. The list of transactions is valid until a subsequent call to ooGetActiveTrans.

- **ppBootFilePath**  Pointer to the boot file that contains the name of the federated database file to be checked. If ppBootFilePath is zero, the boot file path is obtained from the environment variable OO_FD_BOOT and ppBootFilePath is set to the name obtained from OO_FD_BOOT.

- **pHost**  Parameter to identify a specific node. Only transactions started on the node identified by pHost will be returned in the list, subject to filtering based on other parameters. If pHost is zero, transactions started on any node are returned.

- **pUid**  Parameter to identify a specific user. Only transactions started by the user named in pUid will be returned in the list, subject to filtering based on other parameters. If the pUid is zero, transactions started by any user are returned.

DESCRIPTION  (for administration) Use the ooGetActiveTrans function to return a list of active transactions. Do not call the ooInit function (or any other non-recovery functions) in the application process since ooGetActiveTrans already initializes the cache.

RETURN VALUES  Returns a non-zero value for ooStatus if successful.
### ooGetErrorHandler

**FUNCTION NAME**  
 ooGetErrorHandler

**SYNTAX**  
 ooErrorHandlerPtr ooGetErrorHandler();

**DESCRIPTION**  
 Returns pointer to the currently registered error handler function.

**RETURN VALUES**  
 Pointer to currently registered error handler function.

---

### ooGetMemberOffset

**MACRO NAME**  
 ooGetMemberOffset

**SYNTAX**  
 uint32 ooGetMemberOffset(className, memberName);

**DESCRIPTION**  
 Returns the offset to the member field memberName from the start of class className. Used to initialize a key structure. The data member memberName must be public.

**RETURN VALUES**  
 Offset to member field in number of bytes from start of object instance.

---

### ooGetMemberSize

**MACRO NAME**  
 ooGetMemberSize

**SYNTAX**  
 uint32 ooGetMemberSize(className, memberName);

**DESCRIPTION**  
 Returns the size of the member field memberName in the class className. Used to initialize a key structure. The data member memberName must be public.

**RETURN VALUES**  
 Size of member field in bytes.

---

### ooGetMsgHandler

**FUNCTION NAME**  
 ooGetMsgHandler

**SYNTAX**  
 ooMsgHandlerPtr ooGetMsgHandler();

**DESCRIPTION**  
 Returns pointer to the currently registered message handler function.

**RETURN VALUES**  
 Pointer to currently registered message handler function.
FUNCTION NAME: ooGetResourceOwners

SYNTAX:

```c
ooStatus ooGetResourceOwners(
    ooTransInfo** ppOwners,
    ooResource* pResource,
    char** ppBootFilePath,
    ooTransId tId);
```

where:

- **ppOwners**
  Pointer to a location where a pointer to a list of transactions holding resources will be placed. `ooGetResourceOwners` allocates this list. The list is terminated by an entry whose transaction identifier (tId) field is set to `oocInValidTransId`. The list is valid until a subsequent call to `ooGetResourceOwners`.

- **pResource**
  Pointer to the location where the resource that the transaction is waiting for will be placed.

- **ppBootFilePath**
  Pointer to the location that contains a pointer to the name of the federated database file to check. If `ppBootFilePath` is zero, the federated database name is obtained from the environment variable `OO_FD_BOOT` and `ppBootFilePath` is set to the name found in `OO_FD_BOOT`.

- **tId**
  Transaction identifier. Resource information will be returned on the waiting transaction with a transaction identifier matching the value entered for `tId`.

DESCRIPTION:

(for administration) Use the `ooGetResourceOwners` function on a specific transaction to obtain the resource and list of transactions for which the specified transaction is waiting. Do not call the `ooInit` function (or any other non-recovery functions) in the application process since `ooGetResourceOwners` initializes the cache.

RETURN VALUES:

Returns a non-zero value for `ooStatus` if successful.
### ooInit

**SYNTAX**

```c
ooStatus ooInit(
    const uint32 nFiles = 12,
    const uint32 nPages = 200,
    const uint32 nMaxPages = 500,
    const ooBoolean installSigHandler = oocTrue);
```

**DESCRIPTION**

Initializes Objectivity/DB for the application process. Your application should call this function before any other Objectivity/DB services are used. You should invoke ooInit only one time in each application. Subsequent calls to this function are ignored.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`. 

### ooInitAllThreads

**SYNTAX**

```c
extern "C" ooStatus ooInitAllThreads(
    IN uint32 nFiles = 12,
    IN uint32 nPages = 200,
    IN uint32 nMaxPages = 500,
    IN ooBoolean installSigHandler = oocTrue);
```
ooInitThread

where

nFiles Number of active file descriptors reserved for Objectivity/DB. Default value is 12.
nPages Initial number of buffer pages initially allocated for the main thread’s cache. Default value is 200.
nMaxPages Maximum number of buffer pages that Objectivity/DB may allocate for the main thread’s cache. Default value is 500.
installSigHandler Indicates whether you want the system to install the operating system signal handler or not. If the value is oocFalse, then no signal handler will be installed. Default value is oocTrue.

DESCRIPTION Initializes Objectivity/DB for the application process to use multiple threads. This function is to initialize (allocate) main thread data structures and global data structures. Call this function as the first Objectivity/DB call and only one time per process.

RETURN VALUES If successful, it returns oocSuccess. Otherwise, it signals an error and returns oocError.

FUNCTION NAME ooInitThread

SYNTAX

extern "C" ooStatus ooInitThread(
    IN uint32 nPages,
    IN uint32 nMaxPages,
    OUT oothreadId *threadIDPtr);

where

nPages Initial number of buffer pages allocated for this thread’s cache (the Objectivity/C++ default value is 200)
nMaxPages Maximum number of buffer pages Objectivity/DB may allocate for this thread’s cache.
threadIDPtr Pointer to identifier for this thread.
DESCRIPTION Initializes (allocates) this thread’s data structures. ooInitThread does not automatically switch thread context. Call this function for a thread before it makes any other Objectivity/DB calls.

RETURN VALUES If successful, assigns a thread identifier to threadIDPtr and returns oocSuccess. Otherwise, it signals an error and returns oocError.

MACRO NAME ooNewConts

SYNTAX
void ooNewConts(
    className,
    uint32 number,
    const ooHandle(ooObj) &objH,
    uint32 hash,
    uint32 initPages,
    uint32 percentGrowth,
    ooBoolean open,
    ooHandle(className) *pHandle);

DESCRIPTION Creates a batch of containers of class className in the database specified by objH and returns handle. If you specify a container handle or a basic object handle, the new containers will be created under the database that contains the container or basic object referenced by the handle. The main benefit of using this macro is better performance over calling the new operator a number of times.

number is the number of containers to be created.

initPages is the initial number of pages allocated for each container (default is 4 pages with hashing, 2 pages without).

If hash = 0, then non-hashed containers are created. Neither the containers nor any basic objects they contain may be used as a scope when naming an object. You also cannot create Keyed Objects within a non-hashed container.

If hash > 0, then hashed containers are created. The value of hash is used as a clustering factor (number of sequentially indexed objects to place into a page) when you create Keyed Objects within the containers. If you do not care about the clustering factor, you should set the value of hash to 1.

percentGrow specifies that each container should grow by a percentage of the current size (default is 10%).
openP is a Boolean flag indicating whether to open or close the batch containers after creation. If openP is oocTrue, then all the containers will be left open after creation. On the other hand, all the containers will be closed if openP is oocFalse. It is better to close all the containers if none of these will be used immediately.

pHandle is a pointer to an array of container handles. This array is used to store all the handles of the newly created container.

RETURN VALUES Not applicable.

MACRO NAME ooNewKey
SYNTAX ooHandle (className) ooNewKey ( className, (initializer), const ooHandle(ooObj) & contH, ooKey keyStruct );
DESCRIPTION Creates a keyed object of class className in the container specified by handle contH. If contH is the handle of a database, the default container for the database is used. initializer is a list of arguments passed to the constructor for class className and keyStruct specifies the key the object is created with. className must be a basic object class.
RETURN VALUES Handle to newly created Keyed Object.

FUNCTION NAME ooNoLock
SYNTAX ooStatus ooNoLock ( );
DESCRIPTION Disables the Objectivity/DB locking facilities, removing concurrent access protection.
RETURN VALUES ooStatus ( oocSuccess or oocError ).
REMARKS Use ooNoLock with caution. If there is even a remote chance that another application will have access to the same data as your application, you should not disable locking. For most applications, benefits such as data integrity and concurrent access far outweigh the slight performance gain obtained by disabling locking.
FUNCTION NAME  ooRegAtStartFunc
SYNTAX         ooStatus ooRegAtStartFunc(
                  ooVoidFuncPtr atStartFunc);
DESCRIPTION     Registers a function to be called before main, and after all
                Objectivity/DB global constructors are called. The registered function
                can perform any operations necessary on the globally allocated
                Objectivity/DB oovTopTrans, oovTopFD, oovTopDB, oovTopAP
                objects, such as automatically starting a transaction and opening a
                federated database or database. Several calls may be made to
                ooRegAtStartFunc, specifying several handlers. The order in which
                they are called is the order in which they were given to
                ooRegAtStartFunc.
RETURN VALUES   Returns oocSuccess normally, or oocError if the function pointer
                could not be stored.

FUNCTION NAME  ooRegErrorHandler
SYNTAX         ooErrorHandlerPtr ooRegErrorHandler(
                  ooErrorHandlerPtr handlerName);
DESCRIPTION     Registers the error handler function handlerName with
                Objectivity/DB and returns a pointer to the previously registered error
                handler function.
RETURN VALUES   The previous error handler function.

FUNCTION NAME  ooRegMsgHandler
SYNTAX         ooMsgHandlerPtr ooRegMsgHandler(
                  ooMsgHandlerPtr msgHandler);
DESCRIPTION     Registers the message handler function msgHandler with
                Objectivity/DB and returns a pointer to the previously registered
                message handler function.
RETURN VALUES   The previous message handler function.
FUNCTION NAME: ooRegOnExitFunc

SYNTAX:

```c
ooStatus ooRegOnExitFunc(
    ooVoidFuncPtr onExitFunc);
```

DESCRIPTION:

Registers a function to be called after `main`, and before any Objectivity/DB global destructors are called. The registered function can perform any operations necessary on the globally allocated Objectivity/DB `oovTopTrans`, `oovTopFD`, `oovTopDB`, `oovTopAP` objects, such as automatically committing or aborting an active transaction at exit based on application-defined criteria. Several calls may be made to `ooRegOnExitFunc`, specifying several handlers. The order in which they are called is the reverse of that in which they were given to `ooRegOnExitFunc`.

RETURN VALUES:

Returns `oocSuccess` normally, or `oocError` if the function pointer could not be stored.

MACRO NAME: ooReplace

SYNTAX:

```c
ooHandle(ooDBObj) ooReplace(
    ooDBObj,
    (char *dbSysName,
    const uint32 defaultContInitPages = 0,
    const uint32 defaultContPercentGrow = 0,
    const char *HostName = 0,
    const char *pathName = 0),
    const ooHandle(ooFDObj) &fdH);
```

DESCRIPTION:

Creates a database with system name `dbSysName` in the federated database referenced by handle `fdH`, and returns a handle to the new database. Any existing database with the same name in the federated database is removed before the new one is created. A null handle is returned if an error occurs while creating the database.

Valid strings for `dbSysName` follow the same rules as a simple UNIX file name (for example, the slash (/) character is not allowed).

`defaultContInitPages` and `defaultContPercentGrow` are parameters used to create the default container for the database. If either is specified zero, the default value for the parameter is used. See
the discussion of the \texttt{initPages} and \texttt{percentGrow} parameters to the new \texttt{operator} for more information.

\texttt{hostName} is the name of the host system on which the database file is to be created.

\texttt{pathName} specifies the full network path of the directory where the database is to be created. If no path information is included (\texttt{pathName} = 0), the database file is created in the same directory as the federated database file.

Any existing database removed is not recovered if the transaction is aborted.

\textbf{RETURN VALUES} \hspace{1cm} Handle to newly created database

\begin{tabular}{|l|}
\hline
\textbf{FUNCTION NAME} & \texttt{ooResetError} \\
\hline
\textbf{SYNTAX} & \texttt{void ooResetError();} \\
\hline
\textbf{DESCRIPTION} & Clears the Objectivity/DB error flags: \\
& $\texttt{oovLastErrorLevel} = \texttt{oocNoError}$ \\
& $\texttt{oovLastError} = 0$ \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{FUNCTION NAME} & \texttt{ooRunStatus} \\
\hline
\textbf{SYNTAX} & \texttt{void ooRunStatus();} \\
\hline
\textbf{DESCRIPTION} & Prints out a summary of Objectivity/DB internal statistics to \texttt{stdout}. Used primarily to obtain information useful for performance tuning. \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{FUNCTION NAME} & \texttt{ooSetAMSUsage} \\
\hline
\textbf{SYNTAX} & \texttt{extern "C" void ooSetAMSUsage} ( \\
& \hspace{1cm} \texttt{ooAMSUsage amsUsage} = \texttt{oocAMSPreferred}) \\
\hline
\textbf{DESCRIPTION} & (for administration) Use this with the enumerated type \texttt{ooAMSUsage} to specify whether AMS is used if available (the default), is always used, or is never used. \\
\hline
\end{tabular}
FUNCTION NAME: `ooSetErrorFile`

SYNTAX: `ooStatus ooSetErrorFile( char *errorFileName );`

DESCRIPTION: Sets the error message output file.

FUNCTION NAME: `ooSetLargeObjectMemoryLimit`

SYNTAX: `void ooSetLargeObjectMemoryLimit( uint32 size );`

DESCRIPTION: Set the maximum amount of space (in bytes) that can be used by large objects (such as, objects greater than a page in size). This allows you to bypass the default limits set by Objectivity/DB that may be too small for your application. The limit is used to close large objects that are not currently being used so that the amount of memory used is always less than `size`. However, the limit will not be enforced if it cannot close enough objects to keep the memory below `size`. You can call this function anytime after `ooInit` and as often as needed to increase or decrease the amount of space used by large objects.

FUNCTION NAME: `ooSetLockWait`

SYNTAX: `void ooSetLockWait( int32 waitOption = oocNoWait );`

DESCRIPTION: Sets the lock waiting option for the current transaction (if any) and subsequent transactions. If you call `ooSetLockWait` within a transaction, it overrides any value you gave for the lock waiting option when you invoked the `start` member function to start the transaction. The setting remains in effect for all transactions in the current process until it is changed by another call to `ooSetLockWait`. The `waitOption` can take one of the following values:

- `oocTransNoWait`: Wait option is not specified. This is the default value.
- `oocNoWait`: Do not wait for locks.
### ooSignal

**FUNCTION NAME**

**ooSignal**

**SYNTAX**

1. `ooStatus ooSignal(
   const ooErrorLevel errorLevel,
   const ooError &errorID,
   const ooHandle(ooObj) *contextObj...);`

2. `ooStatus ooSignal(
   const ooErrorLevel errorLevel,
   const ooError &errorID,
   const ooHandle(ooObj) &contextObj...);`

**DESCRIPTION**

Raises the error signal and invokes the currently registered error handler.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError).`

---

### ooSwitchThread

**FUNCTION NAME**

**ooSwitchThread**

**SYNTAX**

```c
extern "C" ooStatus ooSwitchThread(
   IN ooThreadId targetThreadID);
```

**DESCRIPTION**

Switches context to a target thread. Switches to the appropriate thread data structures for use in subsequent Objectivity/DB calls by the thread.

**RETURN VALUES**

If successful, this function returns `oocSuccess`. Otherwise, it signals an error and returns `oocError`. 

---

### Macros and Functions B-17
FUNCTION NAME: ooTermThread
SYNTAX:
extern "C" ooStatus ooTermThread( 
    IO ooThreadId *threadIDPtr);

where
threadIDPtr Pointer to identifier for the thread being terminated.

DESCRIPTION: Terminates (frees) data structures for the thread being terminated. Call this function after a thread has completed all other Objectivity/DB calls.
If a thread terminates itself (calls ooTermThread with its own thread identifier), the context is switched back to the main thread. Otherwise, ooTermThread leaves the context unchanged.

RETURN VALUES: If successful, this function sets the given thread identifier to zero (an invalid thread identifier) and returns oocSuccess. Otherwise, it signals an error and returns oocError.

MACRO NAME: ooTypeN
SYNTAX:
ooTypeNumber ooTypeN(className);

DESCRIPTION: Returns type number of class className. This macro is used in conjunction with the ooIsKindOf persistent object member function.

RETURN VALUES: Type number of class.

FUNCTION NAME: ooUpdateIndexes
SYNTAX:
ooStatus ooUpdateIndexes( 
    ooHandle(ooObj) &pHandle);

where
pHandle Handle to the modified or new object of type ooObj, ooContObj, or any derived types of ooObj or ooContObj.
### DESCRIPTION

Explicitly updates all indexes that index the object just created or modified.

### RETURN VALUES

`ooStatus (oocSuccess or oocError)`.

### FUNCTION NAME

**ooUseIndex**

### SYNTAX

```c
extern void ooUseIndex(
    const ooBoolean useIndex = oocTrue);
```

- **useIndex**
  - Set this to `oocTrue` to use indexes during scan or `oocFalse` to disable the use of indexes.

### DESCRIPTION

Enables or disables the use of indexes. Disabling the use of indexes may be desirable when:

- A query is going to scan the entire range of the type and sorting will not be necessary. In such a case, indexes will not speed up your query.
- Data to be scanned has not been committed to indexes since the data was created or last modified.

### FUNCTION NAME

**realloc**

### SYNTAX

1. `ooVString & realloc(ooVString & varString, size_t size);`
2. `ooString( N ) & realloc(ooString( N ) & varString, size_t size);`
3. `ooVArray(className) & realloc(ooVArray(className) & varString, size_t size);`
4. `ooTVArray(className) & realloc(ooTVArray( N ) & varString, size_t size);`

### DESCRIPTION

Resizes storage for `varString`. Same behavior as the ANSI realloc function.

### RETURN VALUES

Reference to `varString`.
**FUNCTION NAME**  
`strcat`

**SYNTAX**

1. `ooVString &strcat(  
ooVString &varString,  
const char *string);`
2. `ooString(N) &strcat(  
ooString(N) &varString,  
const char *string);`

**DESCRIPTION**
Appends a copy of `string` to the end of `varString`. Resizes `varString` to accommodate additional length of `string`. Same behavior as the ANSI `strcat` function.

**RETURN VALUES**
Reference to null-terminated `varString`.

---

**FUNCTION NAME**  
`strcpy`

**SYNTAX**

1. `ooVString &strcpy(  
ooVString &varString,  
const char *string);`
2. `ooString(N) &strcpy(  
ooString(N) &varString,  
const char *string);`

**DESCRIPTION**
Copies `string` to `varString` until null character has been copied. Resizes `varString` to accommodate length of `string`. Same behavior as the ANSI `strcpy` function.

**RETURN VALUES**
Reference to null-terminated `varString`. 
FUNCTION NAME  

\textbf{strlen}

SYNTAX  

1. \texttt{uint32 \texttt{strlen}(const \texttt{ooVString} \& \texttt{varString});}
2. \texttt{uint32 \texttt{strlen}(const \texttt{ooString}(N) \& \texttt{varString});}

DESCRIPTION  

Returns length of \texttt{varString}. Same behavior as the ANSI \texttt{strlen} function.

FUNCTION NAME  

\textbf{strncat}

SYNTAX  

1. \texttt{ooVString \& \texttt{strncat}(ooVString \& \texttt{varString},
\texttt{const char} *\texttt{string},
\texttt{size_t} \texttt{size});}
2. \texttt{ooString(N) \& \texttt{strncat}(ooString(N) \& \texttt{varString},
\texttt{const char} *\texttt{string},
\texttt{size_t} \texttt{size});}

DESCRIPTION  

Appends at most \texttt{size} characters of \texttt{string} to \texttt{varString}. Resizes \texttt{varString} to accommodate additional characters from \texttt{string}. Same behavior as the ANSI \texttt{strncat} function.

RETURN VALUES  

Reference to null-terminated \texttt{varString}.

FUNCTION NAME  

\textbf{strncpy}

SYNTAX  

1. \texttt{ooVString \& \texttt{strncpy}(ooVString \& \texttt{varString},
\texttt{const char} *\texttt{string},
\texttt{size_t} \texttt{size});}
2. \texttt{ooString(N) \& \texttt{strncpy}(ooString(N) \& \texttt{varString},
\texttt{const char} *\texttt{string},
\texttt{size_t} \texttt{size});}
DESCRIPTION
Copies string to varString until either the null character has been copied or size characters have been copied. If the length of string is less than size, pads varString with null characters. If length of string is size or greater, varString will not be null-terminated. Resizes varString to size if currently less than size. Same behavior as the ANSI strncpy function.

FUNCTION NAME
strtok

SYNTAX
1. char *strtok(
   ooVString &varString,
   const char *string);
2. char *strtok(
   ooString(N) &varString,
   const char *string);

DESCRIPTION
Considers string varString to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string string. Returns a pointer to the first character of the first token, and writes a null character into varString immediately following the returned token. Same behavior as the ANSI strtok function.

RETURN VALUES
Pointer to first character of first token in varString, or zero if no token is available.
Global Variables and Constants

This appendix describes the global variables available for use by Objectivity/DB applications.

Global Variables

<table>
<thead>
<tr>
<th>GLOBAL VARIABLE</th>
<th>SYNTAX</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>oovLastError</td>
<td>ooError *oovLastError;</td>
<td>Pointer to ooError structure representing most recent error condition.</td>
</tr>
<tr>
<td>oovLastErrorLevel</td>
<td>ooErrorLevel oovLastErrorLevel;</td>
<td>Enumerated variable representing the severity of most recent error condition.</td>
</tr>
<tr>
<td>oovNError</td>
<td>uint32 oovNError;</td>
<td>contains a count of the total number of errors (not including warnings) that have occurred so far.</td>
</tr>
</tbody>
</table>
GLOBAL VARIABLE  oovTopDB
SYNTAX  ooHandle(ooDBObj)  oovTopDB;
DESCRIPTION  Handle of most recently opened database.

GLOBAL VARIABLE  oovTopFD
SYNTAX  ooHandle(ooFDObj)  oovTopFD;
DESCRIPTION  Handle of current federated database.

GLOBAL VARIABLE  oovTopTrans
SYNTAX  ooTrans  oovTopTrans;
DESCRIPTION  Transaction ID of current transaction.
This appendix describes the constructors, operators, and member functions for the Objectivity/DB persistent base classes (ooObj, ooContObj, ooDefaultContObj, ooDBObj, ooFDObj, and ooGeneObj).

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form const ooHandle(className) & are interchangeable with const ooRef(className) &.

### Class Name: ooObj (d_Persistent_Object)

**Description:** Basic object persistent class.

(ODMG only) d_Persistent_Object is the ODMG equivalent to class ooObj.

**Class Hierarchy:** ooObj

### Constructor: ooObj

**Syntax:**

```cpp
oomObj();
```

**Description:** Default constructor for the class ooObj.

Creates a new basic object.
OPERATOR ooObj::operator delete
SYNTAX void operator delete(void *);
DESCRIPTION Deletes persistent object.
RETURN VALUES Not applicable.

OPERATOR ooObj::operator new
SYNTAX void *operator new(
    size_t,
    const ooHandle(ooObj) & objH = oovTopDB);
DESCRIPTION Creates a persistent basic object using the handle objH as a clustering directive. If you do not provide a clustering directive, the object is created in the default container of the most recently opened or created database (the default database).
Operator new returns a memory pointer to the new basic object, which you can assign to a handle. It returns a null pointer if an error occurs while creating the basic object. You can verify the creation of the basic object by checking the value of the handle using the C++ operator ==0.
If you want to use an object reference to access the new basic object, you can assign the handle to an object reference.

OPERATOR ooObj::operator new
SYNTAX void *operator new(
    size_t,
    const ooRef(ooObj) & objR);
DESCRIPTION Creates a persistent basic object using the object reference objR as a clustering directive. If you do not provide a clustering directive, the object is created in the default container of the most recently opened or created database (the default database).
Operator new returns a memory pointer to the new basic object, which you can assign to a handle. It returns a null pointer if an error occurs while creating the basic object. You can verify the creation of the basic object by checking the value of the handle using the C++ operator ==0.
If you want to use an object reference to access the new basic object, you can assign the handle to an object reference.

### OPERATOR

**ooObj::operator new**

**SYNTAX**

```c
void *ooObj::operator new(
    size_t,
    const char *type);
```

**DESCRIPTION**

(ODMG only) Allocate a transient object of type `ooObj`. Note that `type` is ignored.

### OPERATOR

**ooObj::operator new**

**SYNTAX**

```c
void *ooObj::operator new(
    size_t,
    ooRefHandle(ooObj) near,
    const char *type);
```

**DESCRIPTION**

(ODMG only) If `near` is null, allocate a transient object of type `ooObj`. Otherwise, allocate a persistent object of type `ooObj` near `near`. Note that `type` is ignored.

### OPERATOR

**ooObj::operator new**

**SYNTAX**

```c
void *ooObj::operator new(
    size_t,
    d_Database *db,
    const char *type = 0);
```

**DESCRIPTION**

(ODMG only) If `db` is null, allocate a transient object of type `ooObj`. Otherwise, allocate a persistent object of type `ooObj` in `db`. Note that `type` is ignored.
<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooObj::add_derivatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus add_derivatives(</td>
</tr>
<tr>
<td></td>
<td>const ooHandle(className) &amp;objH);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Creates an association to the object referenced by handle objH, indicating it is a derivative (the successor of) of this object.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>ooStatus (oocSuccess or oocError).</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>ooObj::derivatives, ooObj::sub_derivatives, ooObj::del_derivatives</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooObj::add_derivedFrom</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus add_derivedFrom(</td>
</tr>
<tr>
<td></td>
<td>const ooHandle(className) &amp;objH);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Creates an association to the object referenced by handle objH, indicating this object is derived from it.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>ooStatus (oocSuccess or oocError).</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>ooObj::derivedFrom, ooObj::sub_derivedFrom, ooObj::del_derivedFrom</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooObj::add_mapElems</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus add_mapElems(</td>
</tr>
<tr>
<td></td>
<td>const ooHandle(ooMapElem) &amp;objH);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Creates an association to the object referenced by handle objH.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>ooStatus (oocSuccess or oocError).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooObj::add_nextVers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus add_nextVers(</td>
</tr>
<tr>
<td></td>
<td>const ooHandle(className) &amp;objH);</td>
</tr>
</tbody>
</table>

D-4 Using Objectivity/C++
DESCRIPTION
Creates an association to the object referenced by handle `objH`, indicating it is a next version of this object.

RETURN VALUES
`ooStatus (oocSuccess or oocError)`.

SEE ALSO
`ooObj::nextVers, ooObj::sub_nextVers, ooObj::del_nextVers`

MEMBER FUNCTION `ooObj::defaultToGeneObj`

SYNTAX
```cpp
ooHandle (ooGeneObj) & defaultToGeneObj(
    ooHandle (ooGeneObj) & objH,
    const ooMode openMode = oocNoOpen) const;
```

DESCRIPTION
Returns handle or object reference to the genealogy object if this object is the default version. The genealogy object is opened in the access mode specified by the optional parameter `openMode`. Returns the handle in the user-provided buffer `objH`.

RETURN VALUES
Reference to handle.

SEE ALSO
`ooObj::set_defaultToGeneObj
ooObj::del_defaultToGeneObj`

MEMBER FUNCTION `ooObj::defaultToGeneObj`

SYNTAX
```cpp
ooRef (ooGeneObj) & defaultToGeneObj(
    ooRef (ooGeneObj) & objR,
    const ooMode openMode = oocNoOpen) const;
```

DESCRIPTION
Returns handle or object reference to the genealogy object if this object is the default version. The genealogy object is opened in the access mode specified by the optional parameter `openMode`. Returns the object reference in the user-provided buffer `objR`.

RETURN VALUES
Reference to object reference.

SEE ALSO
`ooObj::set_defaultToGeneObj
ooObj::del_defaultToGeneObj`
MEMBER FUNCTION

`ooObj::defaultToGeneObj`

**SYNTAX**

```cpp
ooHandle(ooGeneObj) defaultToGeneObj
  (const ooMode openMode = oocNoOpen) const;
```

**DESCRIPTION**

Returns handle or object reference to the genealogy object if this object is the default version. The genealogy object is opened in the access mode specified by the optional parameter `openMode`. Returns the handle by value.

**RETURN VALUES**

Handle by value.

**SEE ALSO**

`ooObj::set_defaultToGeneObj`
`ooObj::del_defaultToGeneObj`

---

MEMBER FUNCTION

`ooObj::del_defaultToGeneObj`

**SYNTAX**

```cpp
ooStatus del_defaultToGeneObj();
```

**DESCRIPTION**

Removes any association that exists on link `defaultToGeneObj`, indicating this object is not the default version.

**RETURN VALUES**

`ooStatus` (`oocSuccess` or `oocError`).

**SEE ALSO**

`ooObj::defaultToGeneObj`, `ooObj::set_defaultToGeneObj`

---

MEMBER FUNCTION

`ooObj::del_derivatives`

**SYNTAX**

```cpp
ooStatus del_derivatives();
```

**DESCRIPTION**

Removes all associations that exist on link `derivatives`, indicating there are no derivatives of this object.

**RETURN VALUES**

`ooStatus` (`oocSuccess` or `oocError`).

**SEE ALSO**

`ooObj::derivatives`, `ooObj::sub_derivatives`, `ooObj::add_derivatives`
### MEMBER FUNCTION

**ooObj::del_derivedFrom**

**SYNTAX**

```cpp
ooStatus del_derivedFrom();
```

**DESCRIPTION**

Removes all associations that exist on link `derivedFrom`, indicating this object was not derived from any others.

**RETURN VALUES**

`ooStatus` (`oocSuccess` or `oocError`).

**SEE ALSO**

`ooObj::add_derivedFrom`, `ooObj::sub_derivedFrom`, `ooObj::derivedFrom`

---

### MEMBER FUNCTION

**ooObj::del_geneObj**

**SYNTAX**

```cpp
ooStatus del_geneObj();
```

**DESCRIPTION**

Removes any association that exists on link `geneObj`, indicating this object does not have a genealogy object.

**RETURN VALUES**

`ooStatus` (`oocSuccess` or `oocError`).

**SEE ALSO**

`ooObj::geneObj`, `ooObj::set_geneObj`

---

### MEMBER FUNCTION

**ooObj::del_mapElems**

**SYNTAX**

```cpp
ooStatus del_mapElems();
```

**DESCRIPTION**

Removes any association that exists on link `mapElems`.

**RETURN VALUES**

`ooStatus` (`oocSuccess` or `oocError`).

---

### MEMBER FUNCTION

**ooObj::del_nextVers**

**SYNTAX**

```cpp
ooStatus del_nextVers();
```

**DESCRIPTION**

Removes all associations that exist on link `nextVers`, indicating this object does not have a next version.

**RETURN VALUES**

`ooStatus` (`oocSuccess` or `oocError`).

**SEE ALSO**

`ooObj::nextVers`, `ooObj::add_nextVers`, `ooObj::sub_nextVers`
MEMBER FUNCTION ooObj::del_prevVers
SYNTAX ooStatus del_prevVers();
DESCRIPTION Removes any association that exists on link prevVers, indicating this object does not have a previous version.
RETURN VALUES ooStatus (oocSuccess or oocError).
SEE ALSO ooObj::prevVers, ooObj::set_prevVers

MEMBER FUNCTION ooObj::derivatives
SYNTAX 1. ooStatus derivatives(
    ooItr(className) &itr,
    const ooMode openMode = oocNoOpen) const;
2. ooStatus derivatives(ooItr(ooObj) &itr,
    const char *predicate,
    const ooAccessMode access = oocPublic,
    const ooMode openMode = oocNoOpen) const;
DESCRIPTION 1. Initializes the iterator itr with the handles of all versions of the object derived from this one.
2. Initializes the iterator itr with the handles of all versions of the object derived from this one that satisfy the specified condition.
RETURN VALUES ooStatus (oocSuccess or oocError).
SEE ALSO ooObj::del_derivatives, ooObj::sub_derivatives, ooObj::add_derivatives

MEMBER FUNCTION ooObj::derivedFrom
SYNTAX 1. ooStatus derivedFrom(
    ooItr(ooObj) &itr,
    const ooMode openMode = oocNoOpen) const;
2. ooStatus derivedFrom(
    ooItr(ooObj) &itr,
    const char *predicate,
const ooAccessMode access = oocPublic,
const ooMode openMode = oocNoOpen) const;

DESCRIPTION
1. Initializes the iterator \( \texttt{itr} \) with the handle(s) of all versions of
   the object from which this one was derived.
2. Initializes the iterator \( \texttt{itr} \) with the handle(s) of all versions of
   the object from which this one was derived that satisfy the
   specified condition.

RETURN VALUES
\texttt{ooStatus} (oocSuccess or oocError).

SEE ALSO
\texttt{ooObj::add_derivedFrom, ooObj::sub_derivedFrom, ooObj::del_derivedFrom}

MEMBER FUNCTION \texttt{ooObj::exist_defaultToGeneObj}
SYNTAX
\texttt{ooBoolean exist_defaultToGeneObj(
    const ooHandle(ooGeneObj) \& objH) const;}

DESCRIPTION
Checks for the existence of the association between the default version
and the genealogy object for the object referenced by \( \texttt{objH} \).

RETURN VALUES
\texttt{oocTrue} if the association link exists, otherwise \texttt{oocFalse}.

MEMBER FUNCTION \texttt{ooObj::exist_derivatives}
SYNTAX
\texttt{ooBoolean exist_derivatives(
    const ooHandle(ooGeneObj) \& objH) const;}

DESCRIPTION
Checks for the existence of versions derived from the version of the
object referenced by \( \texttt{objH} \).

RETURN VALUES
\texttt{oocTrue} if there are versions derived from this one, otherwise
\texttt{oocFalse}.

MEMBER FUNCTION \texttt{ooObj::exist_derivedFrom}
SYNTAX
\texttt{ooBoolean exist_derivedFrom(
    const ooHandle(ooGeneObj) \& objH) const;
<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooObj::exist_geneObj</th>
</tr>
</thead>
</table>
| SYNTAX          | ooBoolean exist_geneObj(
|                 | const ooHandle(ooGeneObj) &objH) const; |
| DESCRIPTION     | Checks for the existence of the genealogy object for the object referenced by objH. |
| RETURN VALUES   | oocTrue if the genealogy object exists, otherwise oocFalse. |

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooObj::exist_geneObj() const;</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Checks for the existence of the genealogy object for the object.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>oocTrue if the genealogy object exists, otherwise oocFalse.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooObj::exist_mapElems</th>
</tr>
</thead>
</table>
| SYNTAX          | ooBoolean exist_mapElems(
|                 | const ooHandle(ooMapElem) &objH) const; |
| DESCRIPTION     | Checks for the existence of an association on the link mapElems to the object referenced by objH. |
| RETURN VALUES   | oocTrue if the association exists, otherwise oocFalse. |

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooObj::exist_nextVers</th>
</tr>
</thead>
</table>
| SYNTAX          | ooBoolean exist_nextVers(
|                 | const ooHandle(ooObj) &objH) const; |
| DESCRIPTION     | Checks for the existence of the next version of the object referenced by objH. |
| RETURN VALUES   | oocTrue if the next version exists, otherwise oocFalse. |
MEMBER FUNCTION ooObj::exist_nextVers
SYNTAX ooBoolean exist_nextVers() const;
DESCRIPTION Checks for the existence of the next version of the object.
RETURN VALUES oocTrue if the next version exists, otherwise oocFalse.

MEMBER FUNCTION ooObj::exist_prevVers
SYNTAX ooBoolean exist_prevVers(const ooHandle(ooObj) & objH) const;
DESCRIPTION Checks for the existence of the previous version of the object referenced by objH.
RETURN VALUES oocTrue if the previous version exists, otherwise oocFalse.

MEMBER FUNCTION ooObj::exist_prevVers
SYNTAX ooBoolean exist_prevVers() const;
DESCRIPTION Checks for the existence of the previous version of the object.
RETURN VALUES oocTrue if the previous version exists, otherwise oocFalse.

MEMBER FUNCTION ooObj::geneObj
SYNTAX 1. ooHandle(ooGeneObj) & geneObj(
          ooHandle(ooGeneObj) & objH,
          const ooMode openMode = oocNoOpen) const;
2. ooRef(ooGeneObj) & geneObj(
          ooRef(ooGeneObj) & objR,
          const ooMode openMode = oocNoOpen) const;
3. ooHandle(ooGeneObj) geneObj(
          const ooMode openMode = oocNoOpen) const;
DESCRIPTION Returns handle or object reference to the genealogy object. The genealogy object is opened in the access mode specified by the optional parameter openmode.
1. Returns the handle in the user-provided buffer \textit{objH}.
2. Returns the object reference in the user-provided buffer \textit{objR}.
3. Returns the handle by value.

**RETURN VALUES**

1. Reference to handle.
2. Reference to object reference.
3. Handle by value.

**SEE ALSO**

\texttt{ooObj::set_geneObj, ooObj::del_geneObj}

---

**MEMBER FUNCTION** \texttt{ooObj::mapElems}

**SYNTAX**

\begin{verbatim}
ooStatus mapElems(
    ooItr(ooMapElem) &itr,
    const ooMode openMode = oocNoOpen) const;
\end{verbatim}

**DESCRIPTION**

Initializes the iterator \texttt{itr} to traverse the objects associated via the one-to-many or many-to-many link.

**RETURN VALUES**

\texttt{ooStatus (oocSuccess or oocError)}.

---

**MEMBER FUNCTION** \texttt{ooObj::mapElems}

**SYNTAX**

\begin{verbatim}
ooStatus mapElems(
    ooItr(ooMapElem) &itr,
    const char *predicate,
    const ooAccessMode access = oocPublic,
    const ooMode openMode = oocNoOpen) const;
\end{verbatim}

**DESCRIPTION**

Initializes the iterator \texttt{itr} to traverse the objects associated via the one-to-many or many-to-many link that satisfy the specified condition. If no associations currently exist, \texttt{itr} is returned empty (that is, the first invocation of \texttt{next} returns a null handle).

**RETURN VALUES**

\texttt{ooStatus (oocSuccess or oocError)}. 

---

**D-12 Using Objectivity/C++**
MEMBER FUNCTION ooObj::mark_modified
SYNTAX void ooObj::mark_modified();
DESCRIPTION (ODMG only) An ODMG equivalent name for the member function ooObj::ooUpdate.

MEMBER FUNCTION ooObj::nextVers
SYNTAX ooStatus nextVers(
    ooItr(className) &itr,
    const ooMode openMode = oocNoOpen) const;
DESCRIPTION Initializes the iterator itr with the object reference or handle of all versions of the object that are next versions of this one. For linear versioning this iterator will return only one object reference or handle, for branch versioning it will return one or more.
RETURN VALUES ooStatus (oocSuccess or oocError).
SEE ALSO ooObj::add_nextVers, ooObj::sub_nextVers, ooObj::del_nextVers

MEMBER FUNCTION ooObj::nextVers
SYNTAX ooStatus nextVers(
    ooItr(className) &itr,
    const char *predicate,
    const ooAccessMode access = oocPublic,
    const ooMode openMode = oocNoOpen) const;
DESCRIPTION Initializes the iterator itr with the object reference or handle of all versions of the object that are next versions of this one that satisfy the specified condition.
RETURN VALUES ooStatus (oocSuccess or oocError).
SEE ALSO ooObj::add_nextVers, ooObj::sub_nextVers, ooObj::del_nextVers
MEMBER FUNCTION ooObj::ooCopyInit
SYNTAX virtual ooStatus ooCopyInit();
DESCRIPTION As supplied, returns status of copy operation. Can be defined by user to perform constructor and object assignment operations as part of copy operation performed by copy member function.
RETURN VALUES Non-zero value for ooStatus if copy operation is successful.

MEMBER FUNCTION ooObj::ooGetTypeN
SYNTAX virtual ooTypeNumber ooGetTypeN() const;
DESCRIPTION Returns the type number of the object’s class. This member function is defined for all basic object classes by the DDL Processor.
RETURN VALUES Type number of the object’s class.
SEE ALSO ooObj::ooGetTypeName, ooObj::ooIsKindOf, ooTypeN

MEMBER FUNCTION ooObj::ooGetTypeName
SYNTAX virtual char *ooGetTypeName() const;
DESCRIPTION Returns the name of the object’s class. The string returned is also used by Objectivity/DB, and should only be used in a read-only manner. Modifying this string may result in unexpected program errors. This member function is defined for all basic object classes by the DDL Processor.
RETURN VALUES Name of the object’s class.
SEE ALSO ooObj::ooGetTypeN, ooObj::ooIsKindOf, ooTypeN
MEMBER FUNCTION  ooObj::ooIsKindOf
SYNTAX       virtual ooBoolean ooIsKindOf(
              const ooTypeNumber typeN) const;
DESCRIPTION   Returns the constant oocTrue if the object belongs to the class with the
              type number typeN, or to a class derived from that class. It returns
              oocFalse otherwise. This member function is defined for all basic
              object classes by the DDL Processor.
RETURN VALUES oocTrue if the object belongs to the class with type number typeN or
              a derived class, oocFalse otherwise.
SEE ALSO      ooObj::ooGetTypeName, ooObj::ooGetTypeN, ooTypeN

MEMBER FUNCTION  ooObj::ooNewVersInit
SYNTAX       virtual ooStatus ooNewVersInit();
DESCRIPTION   Specifies copy semantics for creating a new version of an existing
              object.
RETURN VALUES  ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooObj::ooPreMoveInit
SYNTAX       virtual ooStatus ooPreMoveInit();
DESCRIPTION   As supplied, returns status of move operation. Can be defined by user
              to perform cleanup or other operations.
RETURN VALUES  Non-zero value for ooStatus if move is successful.

MEMBER FUNCTION  ooObj::ooPostMoveInit
SYNTAX       virtual ooStatus ooPostMoveInit();
DESCRIPTION   As supplied, returns status of move operation. Can be defined by user
              to perform cleanup or other operations.
RETURN VALUES  Non-zero value for ooStatus if move is successful.
MEMBER FUNCTION ooObj::ooThis
SYNTAX 1. ooHandle(className) &ooThis( ooHandle(className) &objH) const;
2. ooRef(className) &ooThis( ooRef(className) &objR) const;
3. ooHandle(className) ooThis() const;
DESCRIPTION Returns a handle or object reference that references the object of class className. In this case className is ooObj.
1. Returns the handle in the user-provided buffer objH.
2. Returns the object reference in the user-provided buffer objR.
3. Returns the handle by value.
RETURN VALUES 1. Reference to handle.
2. Reference to object reference.
3. Handle by value.

MEMBER FUNCTION ooObj::ooUpdate
SYNTAX ooStatus ooUpdate();
DESCRIPTION Indicates the intention of updating an object when writing a member function within a persistent class.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooObj::ooValidate
SYNTAX virtual ooBoolean ooValidate();
DESCRIPTION Checks for object-level constraints that apply to objects of the given class. Returns oocTrue if the object meets all constraints for the class. Returns oocFalse if it violates any object constraints. This virtual is user-defined.
RETURN VALUES oocTrue if the object meets all object-level constraints for the class, oocFalse otherwise.
MEMBER FUNCTION ooObj::prevVers

SYNTAX
1. ooHandle(ooObj) &prevVers(
   ooHandle(ooObj) & objH,
   const ooMode openmode = oocNoOpen) const;
2. ooRef(ooObj) &prevVers(
   ooRef(ooObj) & objR,
   const ooMode openmode = oocNoOpen) const;
3. ooHandle(ooObj) prevVers(
   const ooMode openmode = oocNoOpen) const;

DESCRIPTION
Returns handle or object reference to previous version of this object. The previous version is opened in the access mode specified by the optional parameter openmode.
1. Returns the handle in the user-provided buffer objH.
2. Returns the object reference in the user-provided buffer objR.
3. Returns the handle by value.

RETURN VALUES
1. Reference to handle.
2. Reference to object reference.
3. Handle by value.

SEE ALSO
ooObj::set_prevVers, ooObj::del_prevVers

MEMBER FUNCTION ooObj::set_defaultToGeneObj

SYNTAX
ooStatus set_defaultToGeneObj(
    const ooHandle(ooGeneObj) & objH);

DESCRIPTION
Associates object to the genealogy object referenced by handle objH, indicating that this is the default object.

RETURN VALUES
ooStatus (oocSuccess or oocError).

SEE ALSO
ooObj::defaultToGeneObj, ooObj::del_defaultToGeneObj
MEMBER FUNCTION  ooObj::set_geneObj
SYNTAX      ooStatus set_geneObj(
             const ooHandle(ooGeneObj) & objH);
DESCRIPTION  Associates object to the genealogy object referenced by handle objH.
RETURN VALUES ooStatus(oocSuccess or oocError).
SEE ALSO    ooObj::geneObj, ooObj::del_geneObj

MEMBER FUNCTION  ooObj::set_prevVers
SYNTAX      ooStatus set_prevVers(
             const ooHandle(className) & objH);
DESCRIPTION  Associates object to that referenced by handle objH, indicating it is the
             previous version of this object.
RETURN VALUES ooStatus(oocSuccess or oocError).
SEE ALSO    ooObj::prevVers, ooObj::del_prevVers

MEMBER FUNCTION  ooObj::sub_derivatives
SYNTAX      ooStatus sub_derivatives(
             const ooHandle(className) & objH)
             const uint32 number = 1);
DESCRIPTION  Removes number of associations to object referenced by handle objH,
             indicating it is no longer a derivative of this object.
RETURN VALUES ooStatus(oocSuccess or oocError).
SEE ALSO    ooObj::derivatives, ooObj::del_derivatives, ooObj::add_derivatives
MEMBER FUNCTION ooObj::sub_derivedFrom
SYNTAX ooStatus sub_derivedFrom(
    const ooHandle(className) & objH,
    const uint32 number = 1);
DESCRIPTION Removes number of associations to object referenced by handle objH, indicating that this object is not derived from it.
RETURN VALUES ooStatus (oocSuccess or oocError).
SEE ALSO ooObj::derivedFrom, ooObj::del_derivedFrom, ooObj::add_derivedFrom

MEMBER FUNCTION ooObj::sub_mapElems
SYNTAX ooStatus sub_mapElems(
    const ooHandle(ooMapElems) & objH,
    const uint32 number = 1);
DESCRIPTION Removes number of associations on link mapElems to object referenced by handle objH.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooObj::sub_nextVers
SYNTAX ooStatus sub_nextVers(
    const ooHandle(className) & objH);
DESCRIPTION Removes association to object referenced by handle objH, indicating it is no longer a next version of this object.
RETURN VALUES ooStatus (oocSuccess or oocError).
SEE ALSO ooObj::add_nextVers, ooObj::nextVers, ooObj::del_nextVers
**CLASS NAME**: ooContObj  
**DESCRIPTION**: Container persistent class. Instances of this class and its subclasses may contain objects of class ooObj and its subclasses.  
**CLASS HIERARCHY**: ooObj->ooContObj

**CONSTRUCTOR**: ooContObj::ooContObj  
**SYNTAX**: ooContObj();  
**DESCRIPTION**: Constructor for class ooContObj.

**OPERATOR**: ooContObj::operator new  
**SYNTAX**: void *operator new(  
  size_t,  
  const ooHandle(ooObj) & objH = oovTopDB);  
**DESCRIPTION**: Creates a container using the handle objH as a clustering directive. If you specify a container handle or a basic object handle, the new container will be created in the database that contains the container or basic object referenced by the handle. If you do not provide a clustering directive, the container is created in the most recently opened or created database (the default database).

**OPERATOR**: ooContObj::operator new  
**SYNTAX**: void *operator new(  
  size_t,  
  const ooRef(ooObj) & objR);  
**DESCRIPTION**: Creates a container using the object reference objR as a clustering directive. If you specify a container or a basic object object reference, the new container will be created in the database that contains the container or basic object referenced by the object reference. If you do not provide a clustering directive, the container is created in the most recently opened or created database (the default database).
## ooContObj operator new

**SYNTAX**

```c
void *operator new(
    size_t,
    const char *contSysName,
    const uint32 hash,
    const uint32 initPages,
    const uint32 percentGrowth,
    const ooHandle<ooObj> &objH = ooTopDB);
```

### contSysName
Name to give the container in the system name space. If you do not wish to give the container a system name, then pass a null pointer.

### hash
Indicates if container is a hashed container. If hash = 0, then a non-hashed container is created. Neither this container or any basic objects it contains may be used as a scope when naming an object. You also cannot create keyed objects within a non-hashed container.

If hash > 0, then a hashed container is created. The value of hash is used as a clustering factor (number of sequentially indexed objects to place into a page) when you create keyed objects within this container. If you do not care about the clustering factor, you should set the value of hash to 1.

### initPages
Initial number of pages allocated for the container (default is 4 pages with hashing, 3 pages without).

### percentGrowth
Specifies that the container should grow by a percentage of the current size (default is 10%).

### objH
Handle to use as a clustering directive specifying where the new container will be created. If the clustering directive is a database handle, the container will be created in the specified database. If the clustering directive is a container handle or a basic object handle, the new container will be created within the database that contains the container or basic object referenced by the handle. The default
value is the handle of the most-recently opened or created database.

DESCRIPTION Creates a container using the handle \( \text{objH} \) as a clustering directive. If you specify a container or a basic object handle, the new container will be created in the database that contains the container or basic object referenced by the handle. If you do not provide a clustering directive, the container is created in the most recently opened or created database (the default database).

Operator \texttt{new} returns a memory pointer to the new container, which you can assign to a handle. It returns a null pointer if an error occurs while creating the container. You can verify the creation of the container by checking the value of the handle using the C++ operator \( \texttt{==}0 \).

If you want to use an object reference to access the new container, you can assign the handle to an object reference.

OPERATOR \texttt{ooContObj::operator new}

SYNTAX

\begin{verbatim}
void *operator new(
    size_t,
    const char *contSysName,
    const uint32 hash,
    const uint32 initPages,
    const uint32 percentGrowth,
    const ooRef(ooObj) & objR);
\end{verbatim}

DESCRIPTION Creates a container using the handle \( \text{objR} \) as a clustering directive. If you specify a container or a basic object object reference, the new container will be created in the database that contains the container or basic object referenced by the object reference.

Operator \texttt{new} returns a memory pointer to the new container, which you can assign to a handle. It returns a null pointer if an error occurs while creating the container. You can verify the creation of the container by checking the value of the handle using the C++ operator \( \texttt{==}0 \).

If you want to use an object reference to access the new container, you can assign the handle to an object reference.
## OPERATOR

**ooContObj::operator new**

**SYNTAX**

```c
void *ooContObj::operator new(
    size_t,
    const char *type);
```

**DESCRIPTION**

(ODMG only) Allocate a transient object of type `ooContObj`. Note that `type` is ignored.

## OPERATOR

**ooContObj::operator new**

**SYNTAX**

```c
void *ooObj::operator new(
    size_t,
    ooRefHandle(ooObj) near,
    const char *type);
```

**DESCRIPTION**

(ODMG only) If `near` is null, allocate a transient object of type `ooContObj`. Otherwise, allocate a persistent object of type `ooContObj` near `near`. Note that `type` is ignored.

## OPERATOR

**ooContObj::operator new**

**SYNTAX**

```c
void *ooContObj::operator new(
    size_t,
    d_Database *db,
    const char *type = 0);
```

**DESCRIPTION**

(ODMG only) If `db` is null, allocate a transient object of type `ooContObj`. Otherwise, allocate a persistent object of type `ooContObj` in `db`. Note that `type` is ignored.

## MEMBER FUNCTION

**ooContObj::ooIsKindOf**

**SYNTAX**

```c
virtual ooBoolean ooIsKindOf( const ooTypeNumber typeN ) const;
```

**DESCRIPTION**

Returns the constant `ooTrue` if `typeN` is identical to the type number of the object’s class, or is identical to the type number of any subclass of the class which has the type number `typeN`. It returns `ooFalse`
otherwise. This member function is defined for all container classes by
the DDL Processor.

RETURN VALUES
  oocTrue if typeN matches the object's class or a subclass, oocFalse
  otherwise.

SEE ALSO
  ooTypeN

MEMBER FUNCTION ooContObj::ooThis
SYNTAX
  ooHandle(ooContObj) & ooThis(  
    ooHandle(ooContObj) & objH) const;
DESCRIPTION
  Returns the handle that references the object of class ooContObj in a
  user-provided buffer objH.
RETURN VALUES
  Reference to handle.

MEMBER FUNCTION ooContObj::ooThis
SYNTAX
  ooRef(ooContObj) & ooThis(  
    ooRef(ooContObj) & objR) const;
DESCRIPTION
  Returns the object reference that references the object of class
  ooContObj in a user-provided buffer objR.
RETURN VALUES
  Reference to object reference.

MEMBER FUNCTION ooContObj::ooThis
SYNTAX
  ooHandle(ooContObj) ooThis() const;
DESCRIPTION
  Returns the handle by value that references the object of class
  ooContObj.
RETURN VALUES
  Handle by value.
CLASS NAME: **ooDefaultContObj**

DESCRIPTION: Default container persistent class. User defined subclasses of ooDefaultContObj are not allowed.

CLASS HIERARCHY: ooObj->ooContObj->ooDefaultContObj

CONSTRUCTOR: ooDefaultContObj::ooDefaultContObj

SYNTAX: ooDefaultContObj();

DESCRIPTION: Constructor for class ooDefaultContObj.

MEMBER FUNCTION: ooDefaultContObj::ooIsKindOf

SYNTAX: virtual ooBoolean ooIsKindOf (const ooTypeNumber typeN) const;

DESCRIPTION: Returns the constant oocTrue if typeN is identical to the type number of the object’s class, or is identical to the type number of any subclass of the class which has the type number typeN. It returns oocFalse otherwise. This member function is defined for all container classes by the DDL Processor.

RETURN VALUES: oocTrue if typeN matches the object’s class or a subclass, oocFalse otherwise.

SEE ALSO: ooTypeN

MEMBER FUNCTION: ooDefaultContObj::ooThis

SYNTAX: ooHandle(ooDefaultContObj) &ooThis( ooHandle(ooDefaultContObj) &objH) const;

DESCRIPTION: Returns handle that references the object of class ooDefaultContObj in a user-provided buffer objH.

RETURN VALUES: Reference to handle.
MEMBER FUNCTION  ooDefaultContObj::ooThis
SYNTAX          ooRef(ooDefaultContObj) &ooThis(
                  ooRef(ooDefaultContObj) &objR) const;
DESCRIPTION      Returns the object reference that references the object of class
                  ooDefaultContObj in the user-provided buffer objR.
RETURN VALUES    Reference to object reference.

MEMBER FUNCTION  ooDefaultContObj::ooThis
SYNTAX          ooHandle(ooDefaultContObj) ooThis() const;
DESCRIPTION      Returns the handle by value that references the object of class
                  ooDefaultContObj.
RETURN VALUES    Handle by value.
### ooDBObj

**CLASS NAME**  
**Description**  
Database persistent class. Instances of this class and its subclasses may contain objects of class ooContObj and its derived classes. User-defined derived classes of ooDBObj are not allowed.

**CLASS HIERARCHY**  
OOObj->OODBObj

### CONSTRUCTOR

**ooDBObj::ooDBObj**

**SYNTAX**

```cpp
ooDBObj(const char * dbSysName,
    const uint32 defContInitPages = 0,
    const uint32 defContGrowth = 0,
    const char * hostName = 0,
    const char * pathName = 0
    uint32 weight = 1);
```

**DESCRIPTION**  
Constructor for class ooDBObj. Creates a database with name specified by string `dbSysName`. `defContInitPages` and `defContGrowth` are parameters used to create the default container for the database. If either is specified zero, the default value for the parameter is used. `hostName` is the name of the host system on which the database is to be created. If you specify a host name, you must also specify `pathName`. `pathName` specifies the full network path of the directory where the database is to be created. If you specify a path name, you must also specify a host name in `hostName`. If no path information is included (`pathName = 0`), the database file is created in the same directory as the federated database file. Optionally, `pathName` can include the file name for the database file (This file naming option is not available on VMS). `weight` specifies the weight of the first database image if Objectivity/DB Data Replication Option is being used. The default weight is 1.
OPERATOR ooDBObj::operator new  
SYNTAX void *operator new(
    size_t,
    const ooHandle(ooFDObj) & fdH = oovTopFD);
DESCRIPTION Creates a database in the federated database specified by `fdH`. Operator `new` returns a memory pointer to the new database, which you can assign to a handle. It returns a null pointer if an error occurs while creating the database. You can verify the creation of the database by checking the value of the handle using the C++ `operator ==`.
If you want to use an object reference to access the new database, you can assign the handle to an object reference.

OPERATOR ooDBObj::operator new  
SYNTAX void *operator new(
    size_t,
    const ooRef(ooFDObj) & fdR);
DESCRIPTION Creates a database in the federated database specified by `fdR`. Operator `new` returns a memory pointer to the new database, which you can assign to a handle. It returns a null pointer if an error occurs while creating the database. You can verify the creation of the database by checking the value of the handle using the C++ `operator ==`.
If you want to use an object reference to access the new database, you can assign the handle to an object reference.

CLASS NAME ooFDObj  
DESCRIPTION Federated database persistent class. Instances of this class and its subclasses may contain objects of class ooDBObj. User-defined subclasses of ooFDObj are not allowed.
CLASS HIERARCHY ooObj->ooFDObj
CLASS NAME: ooGeneObj
DESCRIPTION: Version genealogy object class. Instances of this class are used to support the version genealogy facility of Objectivity/DB.
CLASS HIERARCHY: ooObj->ooGeneObj

CONSTRUCTOR: ooGeneObj::ooGeneObj
SYNTAX: ooGeneObj();
DESCRIPTION: Constructor for class ooGeneObj.

MEMBER FUNCTION: ooGeneObj::allVers
SYNTAX: ooStatus allVers(ooItr(ooObj)&itr,
const ooMode openMode = oocNoOpen) const;
DESCRIPTION: Initializes the iterator itr with the handles of all versions of the object.
RETURN VALUES: ooStatus (oocSuccess or oocError).
SEE ALSO: ooGeneObj::add_allVers, ooGeneObj::sub_allVers, ooGeneObj::del_allVers

MEMBER FUNCTION: ooGeneObj::add_allVers
SYNTAX: ooStatus add_allVers(const ooHandle(className) &objH);
DESCRIPTION: Creates an association to the object referenced by handle objH, indicating it is a version of this object genealogy.
RETURN VALUES: ooStatus (oocSuccess or oocError).
SEE ALSO: ooGeneObj::allVers, ooGeneObj::sub_allVers, ooGeneObj::del_allVers
### MEMBER FUNCTION ooGeneObj::defaultVers

**SYNTAX**

```cpp
ooHandle(ooObj) &defaultVers(
   ooHandle(ooObj) &objH,
   const ooMode openmode = oocNoOpen) const;
```

**DESCRIPTION**

Returns handle to default version. The default version is opened in the access mode specified by the optional parameter `openmode`. Returns the handle in the user-provided buffer `objH`.

**RETURN VALUES**
Reference to handle.

**SEE ALSO**
- `ooGeneObj::set_defaultVers`
- `ooGeneObj::del_defaultVers`

### MEMBER FUNCTION ooGeneObj::defaultVers

**SYNTAX**

```cpp
ooRef(ooObj) &defaultVers(
   ooRef(ooObj) &objR,
   const ooMode openmode = oocNoOpen) const;
```

**DESCRIPTION**

Returns handle to default version. The default version is opened in the access mode specified by the optional parameter `openmode`. Returns the object reference in the user-provided buffer `objR`.

**RETURN VALUES**
Reference to object reference.

**SEE ALSO**
- `ooGeneObj::set_defaultVers`
- `ooGeneObj::del_defaultVers`

### MEMBER FUNCTION ooGeneObj::defaultVers

**SYNTAX**

```cpp
ooHandle(ooObj) defaultVers(
   const ooMode openmode = oocNoOpen) const;
```

**DESCRIPTION**

Returns handle to default version. The default version is opened in the access mode specified by the optional parameter `openmode`. Returns the handle by value.

**RETURN VALUES**
Handle by value.

**SEE ALSO**
- `ooGeneObj::set_defaultVers`
- `ooGeneObj::del_defaultVers`
### MEMBER FUNCTION

**ooGeneObj::del_allVers**

**SYNTAX**

```cpp
ooStatus del_allVers();
```

**DESCRIPTION**

Removes all associations that exist on link `allVers`.

**RETURN VALUES**

`ooStatus(oocSuccess or oocError)`.

**SEE ALSO**

`ooGeneObj::add_allVers, ooGeneObj::sub_allVers, ooGeneObj::allVers`

---

### MEMBER FUNCTION

**ooGeneObj::del_defaultVers**

**SYNTAX**

```cpp
ooStatus del_defaultVers();
```

**DESCRIPTION**

Removes any association that exists on link `defaultVers`, indicating there is no default version of this object.

**RETURN VALUES**

`ooStatus(oocSuccess or oocError)`.

**SEE ALSO**

`ooGeneObj::set_defaultVers, ooGeneObj::defaultVers`

---

### MEMBER FUNCTION

**ooGeneObj::exist_allVers**

**SYNTAX**

```cpp
ooBoolean exist_allVers(
    const ooHandle(ooObj) & objH) const;
```

**DESCRIPTION**

Checks for the existence of an association on association link `allVers`.

**RETURN VALUES**

`oocTrue` if an association exists on the link, otherwise `oocFalse`.

---

### MEMBER FUNCTION

**ooGeneObj::exist_defaultVers**

**SYNTAX**

```cpp
ooBoolean exist_defaultVers(
    const ooHandle(ooObj) & objH) const;
```

**DESCRIPTION**

Checks for the existence of an association on association link `defaultVers`.

**RETURN VALUES**

`oocTrue` if an association exists on the link, otherwise `oocFalse`.

---

**Base Classes**  D-31
MEMBER FUNCTION ooGeneObj::ooIsKindOf
SYNTAX virtual ooBoolean ooIsKindOf(const ooTypeNumber typeN) const;
DESCRIPTION Returns the constant oocTrue if typeN is identical to the type number of the object’s class, or is identical to the type number of any subclass of the class which has the type number typeN. It returns oocFalse otherwise. This member function is defined for all container classes by the DDL Processor.
RETURN VALUES oocTrue if typeN matches the object’s class or a subclass, oocFalse otherwise.
SEE ALSO ooTypeN

MEMBER FUNCTION ooGeneObj::ooThis
SYNTAX ooHandle(ooGeneObj) &ooThis(ooHandle(ooGeneObj) &objH) const;
DESCRIPTION Returns handle or object reference that references the object of class ooGeneObj. Returns the handle in the user-provided buffer objH.
RETURN VALUES Reference to handle.

MEMBER FUNCTION ooGeneObj::ooThis
SYNTAX ooRef(ooGeneObj) &ooThis(ooRef(ooGeneObj) &objR) const;
DESCRIPTION Returns handle or object reference that references the object of class ooGeneObj. Returns the object reference in the user-provided buffer objR.
RETURN VALUES Reference to object reference.
MEMBER FUNCTION  ooGeneObj::ooThis
SYNTAX  ooHandle(ooGeneObj) ooThis() const;
DESCRIPTION  Returns handle or object reference that references the object of class ooGeneObj. Returns the handle by value.
RETURN VALUES  Handle by value.

MEMBER FUNCTION  ooGeneObj::set_defaultVers
SYNTAX  ooStatus set_defaultVers(  
const ooHandle(ooObj) & objH);
DESCRIPTION  Associates generic version object to that referenced by handle objH, indicating it is the default version.
RETURN VALUES  ooStatus (oocSuccess or oocError).
SEE ALSO  ooGeneObj::defaultVers, ooGeneObj::del_defaultVers

MEMBER FUNCTION  ooGeneObj::sub_allVers
SYNTAX  ooStatus sub_allVers(  
const ooHandle(className) & objH);
DESCRIPTION  Removes association to object referenced by handle objH, indicating it is no longer a version of this object genealogy.
RETURN VALUES  ooStatus (oocSuccess or oocError).
SEE ALSO  ooGeneObj::add_allVers, ooGeneObj::allVers ooGeneObj::del_allVers
Dictionary Classes

This appendix describes the constructors and member functions for the Objectivity/DB dictionary classes. Instances of these classes are persistent.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`.

### CLASS NAME

**ooMap**

**DESCRIPTION**

Defines the hash table, manipulates the table and its elements, and provides object lookup by name.

**CLASS HIERARCHY**

ooObj ooMap

### CONSTRUCTOR

**ooMap::ooMap**

**SYNTAX**

```cpp
ooMap (const uint32 nBin = oocMapInitHashBinSize,
       const uint32 maxAvgDensity = oocMapMaxAvgDensity,
       const uint32 percentGrowth = oocMapPercentGrow);
```

**DESCRIPTION**

Constructor for class ooMap.

### MEMBER FUNCTION

**ooMap::add**

**SYNTAX**

1. ```cpp
   ooStatus add(
                 const char *name,
                 const ooRef(ooObj) &objR,
                 ...);
   ```
ooHandle(ooMapElem) & elemH =
oocNullMapElemHandle);
2. ooStatus add(
    const char *name,
    const ooRef(ooObj) & objR,
    ooRef(ooMapElem) & elemR);

DESCRIPTION
1. Adds a new element to the hash table. The new element has the
   name name and the OID referenced by objR. If elemH is given, it
   is initialized to reference the new element.
2. Adds a new element to the hash table. The new element has the
   name name, and the OID referenced by objR. elemR is initialized
   to reference the new element.

RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooMap::clearParam
SYNTAX ooStatus clearParam();
DESCRIPTION Reset hash table statistical parameters to zero.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooMap::forceAdd
SYNTAX 1. ooStatus forceAdd(
    const char *name,
    const ooRef(ooObj) & objR,
    ooHandle(ooMapElem) & elemH =
    oocNullMapElemHandle);
2. ooStatus forceAdd(
    const char *name,
    const ooRef(ooObj) & objR,
    ooRef(ooMapElem) & elemR);

DESCRIPTION 1. Forces the addition of an element to the hash table without
   checking for the prior existence of an element with the same
   name. If elemH is given, it is initialized to reference the new
   element.
2. Forces the addition of an element to the hash table without checking for the prior existence of an element with the same name. \texttt{elemR} is initialized to reference the new element.

RETURN VALUES \texttt{ooStatus (oocSuccess or oocError)}.

MEMBER FUNCTION \texttt{ooMap::isMember}

SYNTAX \texttt{ooBoolean isMember(const char *name);}

DESCRIPTION Checks if any element in the hash table contains the name \texttt{name}.

RETURN VALUES \texttt{oocTrue if any element contains the name, otherwise oocFalse}.

MEMBER FUNCTION \texttt{ooMap::lookup}

SYNTAX 1. \texttt{ooRef(ooObj) lookup(const char *name);}  
2. \texttt{ooStatus lookup(const char *name, ooRef(ooObj) &objR, 
const ooMode openMode=oocRead);}  
3. \texttt{ooStatus lookup(const char *name, ooHandle(ooObj) &objH, 
const ooMode openMode=oocRead);}  
4. \texttt{ooStatus lookup(const char *name, ooRef(ooMapElem) &elemR);}  
5. \texttt{ooStatus lookup(const char *name, ooHandle(ooMapElem) &elemH);}  

DESCRIPTION 1. Looks up the string \texttt{name} in the hash table; returns OID of the object.
2. Looks up the string \texttt{name} in the hash table; returns OID of the object and opens the object in mode specified (unless \texttt{oocNoOpen} is specified).
3. Looks up the string \textit{name} in the hash table; returns the handle of the object and opens the object in mode specified (unless \texttt{oocNoOpen} is specified).
4. Looks up the string \textit{name} in the hash table; returns the object reference of the map element.
5. Looks up the string \textit{name} in the hash table; returns the handle of the map element.

RETURN VALUES \texttt{ooStatus (oocSuccess or oocError)}.

\begin{tabular}{|l|}
\hline
\textbf{MEMBER FUNCTION} & \texttt{ooMap::maxAvgDensity} \\
\hline
\textbf{SYNTAX} & \texttt{uint32 maxAvgDensity() const;} \\
\hline
\textbf{DESCRIPTION} & Returns the allowable maximum average density of the hash table. \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{MEMBER FUNCTION} & \texttt{ooMap::nBin} \\
\hline
\textbf{SYNTAX} & \texttt{uint32 nBin() const;} \\
\hline
\textbf{DESCRIPTION} & Returns the number of bins in the hash table. \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{MEMBER FUNCTION} & \texttt{ooMap::nElement} \\
\hline
\textbf{SYNTAX} & \texttt{uint32 nElement() const;} \\
\hline
\textbf{DESCRIPTION} & Returns total number of elements in the hash table. \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{MEMBER FUNCTION} & \texttt{ooMap::nameHashFunction} \\
\hline
\textbf{SYNTAX} & \texttt{static ooNameHashFuncPtr nameHashFunction();} \\
\hline
\textbf{DESCRIPTION} & Returns the name hash function used with the hash table. \\
\hline
\end{tabular}
MEMBER FUNCTION ooMap::percentGrow

SYNTAX uint32 percentGrow() const;

DESCRIPTION Returns the percent growth for the number of bins in the table.

MEMBER FUNCTION ooMap::printStat

SYNTAX void printStat(FILE * outFile = stdout) const;

RETURN VALUES Prints out runtime statistical information.

MEMBER FUNCTION ooMap::refEnable

SYNTAX ooBoolean refEnable() const;

DESCRIPTION Returns flag indicating whether referential integrity is to be maintained.

RETURN VALUES oocTrue if referential integrity is to be maintained, otherwise oocFalse.

MEMBER FUNCTION ooMap::rehash

SYNTAX ooStatus rehash(
    const uint32 binSize = oocInitMapHashBinSize);

DESCRIPTION Resize the hash table with the number of bins specified by binSize. The default value, oocInitMapHashBinSize, is 11.

RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooMap::remove

SYNTAX
1. ooStatus remove(const char *name);
2. ooStatus remove(const ooHandle(ooMapElem) &obj);
DESCRIPTION 1. Deletes the element with name `name` from the hash table.
2. Deletes the element referenced by the given handle from the hash table.

RETURN VALUES `ooStatus` (`oocSuccess` or `oocError`). `remove` will return `oocSuccess` if asked to remove a non-existing element.

MEMBER FUNCTION `ooMap::replace`

SYNTAX 1. `ooStatus replace(const char * name, const ooRef(ooObj) & objR, ooHandle(ooMapElem) & elemH = oocNullMapElemHandle);`
2. `ooStatus replace(const char * name, const ooRef(ooObj) & objR, ooRef(ooMapElem) & elemR);`

DESCRIPTION 1. Replaces the OID associated with string `name` with OID of the object referenced by `objR`. If `elemH` is given, it is initialized to reference the new element.
2. Replaces the OID associated with string `name` with OID of the object referenced by `objR`. `elemR` is initialized to reference the new element.

RETURN VALUES `ooStatus` (`oocSuccess` or `oocError`).

MEMBER FUNCTION `ooMap::set_nameHashFunction`

SYNTAX `static void set_nameHashFunction(ooNameHashFuncPtr hashFunction);`

DESCRIPTION Sets the name of the name hash function used with the hash table. The hash function is not stored persistently. Since all access to the `ooMap` must use the same hash function, this member function must be invoked in every program that uses the `ooMap`. 
MEMBER FUNCTION  ooMap::set_refEnable
SYNTAX  ooStatus set_refEnable(ooBoolean refEnable = oocTrue);
DESCRIPTION  Sets flag for enabling or disabling referential integrity. Do not disable referential integrity unless you are fully aware of the consequences.
RETURN VALUES  ooStatus (oocSuccess or oocError).

CLASS NAME  ooMapElem
DESCRIPTION  Defines each element in the hash table.
CLASS HIERARCHY  ooObj ooMapElem

MEMBER FUNCTION  ooMapElem::name
SYNTAX  const char *name();
DESCRIPTION  Obtains the name of the element.

MEMBER FUNCTION  ooMapElem::oid
SYNTAX  ooRef(ooObj) oid() const;
DESCRIPTION  Obtains the element's Object Identifier (OID).

MEMBER FUNCTION  ooMapElem::set_oid
SYNTAX  ooStatus set_oid(const ooRef(ooObj) &objR) const;
DESCRIPTION  Sets the data member OID in the element with the object referenced by objR.
**CLASS NAME**  
**ooMapItr**

**DESCRIPTION**  
Defines an iterator for the hash table so that elements can be returned one by one.

**CLASS HIERARCHY**  
`ooItr(ooMapElem) -> ooMapItr`

---

**CONSTRUCTOR**  
**ooMapItr::ooMapItr**

**SYNTAX**  
`ooMapItr();`

**DESCRIPTION**  
Constructor for class `ooMapItr`.

---

**OPERATOR**  
**ooMapItr::operator =**

**SYNTAX**  
1. `ooMapItr & operator = (ooHandle(ooMap) & mapH);`
2. `ooMapItr & operator = (ooRef(ooMap) & mapR);`

**DESCRIPTION**  
Initializes a map iterator.

1. Initializes a map iterator to reference the object referenced by `mapH`.
2. Initializes a map iterator to reference the object referenced by `mapR`.

---

**MEMBER FUNCTION**  
**ooMapItr::ooMapItr**

**SYNTAX**  
`ooMapItr (ooHandle(ooMap) & mapH);`

**DESCRIPTION**  
Initializes a map iterator for the object referenced by `mapH`.

---

**E-8 Using Objectivity/C++**
MEMBER FUNCTION

**ooMapItr::next**

SYNTAX

```cpp
void next();
```

DESCRIPTION

Sets the iterator to reference the next element of the current iteration. This member function is inherited from class `ooItr` and retrieves map elements in a non-update mode. To access the target object of a map element, use the `oid` member function, which returns a type-independent OID of the object. You must then assign this OID to a handle or to an object reference that is cast to the appropriate type. You cannot add or delete elements during the map iteration. Only read mode is allowed during the iteration. You can modify the objects the map is indexing (your objects). You should not, however, delete objects if `refEnable` is set by `set_refEnable`.

RETURN VALUES

`oocTrue` if there is a next object within the iteration, `oocFalse` if all of the objects in the iteration have been traversed or if an error occurred.
Index Classes

This appendix describes the constructors and member functions for the Objectivity/DB index classes.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`.

Persistent Index Classes

Instances of these classes are persistent.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooKeyDesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS HIERARCHY</td>
<td>ooIndexDescBase -&gt; ooKeyDesc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>ooKeyDesc::ooKeyDesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooKeyDesc(</td>
</tr>
<tr>
<td></td>
<td>const ooTypeNumber typeN,</td>
</tr>
<tr>
<td></td>
<td>const ooBoolean unique = oocFalse);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Constructor for ooKeyDesc class. Creates a key description.</td>
</tr>
</tbody>
</table>
MEMBER FUNCTION ooKeyDesc::addField
SYNTAX ooStatus addField(
    const ooHandle(ooKeyField) & fieldH);
DESCRIPTION Adds an index key field to the key description after the last field added.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooKeyDesc::createIndex
SYNTAX ooStatus createIndex(
    const ooHandle(ooSystemObj) & scopeH);
DESCRIPTION Creates an index in a container, database, or federated database from the key description.
If the call to createIndex fails, Objectivity/DB aborts the transaction, after issuing messages that state that the index creation has failed and that the transaction is being aborted. ooSystemObj is either ooContObj, ooDBObj, or ooFDObj.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooKeyDesc::dropIndex
SYNTAX ooStatus dropIndex(
    const ooHandle(ooSystemObj) & scopeH);
DESCRIPTION Drops an index from a container, database, or federated database.
ooSystemObj is either ooContObj, ooDBObj, or ooFDObj.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooKeyDesc::getTypeN
SYNTAX ooTypeNumber getTypeN() const;
DESCRIPTION Returns the Objectivity/DB type number of the indexed class.
RETURN VALUES Type number of the indexed class.
### MEMBER FUNCTION: `ooKeyDesc::getTypeName`

**SYNTAX**

```cpp
const char * getTypeName() const;
```

**DESCRIPTION**

Returns the name of the indexed class.

**RETURN VALUES**

Pointer of type `char` to the name of the indexed class.

### MEMBER FUNCTION: `ooKeyDesc::isConsistent`

**SYNTAX**

```cpp
ooBoolean ooKeyDesc::isConsistent();
```

**DESCRIPTION**

Checks the consistency of a key descriptor.

**RETURN VALUES**

Returns `oocTrue` if all the key fields that are part of the key descriptor are consistent. Returns `oocFalse` if any key field of the key descriptor is inconsistent.

### MEMBER FUNCTION: `ooKeyDesc::isUnique`

**SYNTAX**

```cpp
ooBoolean isUnique();
```

**DESCRIPTION**

Determines whether or not a particular index was created to be unique. When an index is unique, each of its indexed objects must have unique key field values.

**RETURN VALUES**

`oocTrue` if the target index is unique and `oocFalse` if it is not.

### MEMBER FUNCTION: `ooKeyDesc::nField`

**SYNTAX**

```cpp
unit32 nField() const;
```

**DESCRIPTION**

Gets the number of fields in the index.

**RETURN VALUES**

Integer giving the number of fields in the index.
MEMBER FUNCTION  
**ooKeyDesc::removeIndexes**

SYNTAX  
```
ooStatus removeIndexes();
```

DESCRIPTION  
Removes all indexes in the federated database created from the key scan description.

RETURN VALUES  
```
ooStatus (oocSuccess or oocError).
```

CLASS NAME  
**ooKeyField**

CLASS HIERARCHY  
```
ooObj ooKeyField
```

CONSTRUCTOR  
**ooKeyField::ooKeyField**

SYNTAX  
```
ooKeyField(
    const ooTypeNumber typeN,
    const char *memberName,
    ooBoolean fixed = oocFalse,
    uint32 maxstrlen = 24);
```

where  
- **typeN**  
  Type number of the class containing the member for the key field. This class must match the class specified when creating the key description.

- **memberName**  
  Name of the member in the class

- **fixed**  
  Specifies whether the strings in the index are to be of fixed or variable length. The value can be either oocTrue or oocFalse; the default is oocFalse.

- **maxstrlen**  
  Specifies what the maximum length of a string stored in the index can be. This length is always reserved for the string. If fixed is set to oocTrue, strings are limited to a length of maxstrlen; if fixed is set to oocFalse, strings of any length can be indexed, but the
first maxstrlen bytes are for first attempts at resolving comparisons of indexed strings.

DESCRIPTION Constructor for ooKeyField class. Creates a key field.

MEMBER FUNCTION ooKeyField::getName
SYNTAX virtual const char *getName() const;
DESCRIPTION Gets the name of the index key field.
RETURN VALUES Pointer of type char to the name of the index key field.

MEMBER FUNCTION ooKeyField::getTypeN
SYNTAX ooTypeNumber getTypeN() const;
DESCRIPTION Gets the type number of the class of the member to which the key field refers.

MEMBER FUNCTION ooKeyField::isConsistent
SYNTAX ooBoolean ooKeyField::isConsistent();
DESCRIPTION Checks the consistency of a key field.
RETURN VALUES Returns oocTrue if the class and the type of the data member are consistent with the one used to initialize the index field. If either the class or data member has been deleted, or the type of the data member has been changed, this member function returns oocFalse.

MEMBER FUNCTION ooKeyField::isNamed
SYNTAX virtual ooBoolean isNamed(const char *const name) const;
DESCRIPTION Checks whether the name of the field matches name.
RETURN VALUES ooStatus (oocSuccess or oocError).
Non-Persistent Index Classes

This section lists the constructors and member functions for the Objectivity/DB non-persistent index classes.

**CLASS NAME**  
*ooLookupFieldBase*

**CLASS HIERARCHY**  
*ooLookupFieldBase*

**CONSTRUCTOR**  
*ooLookupFieldBase::ooLookupFieldBase*

**SYNTAX**  
```cpp
ooLookupFieldBase(
    const ooKeyField &field,
    const void *valuePtr,
    ooRelatOp relatOp = oocEQ);
```

**DESCRIPTION**  
Constructor for *ooLookupFieldBase* class.

**MEMBER FUNCTION**  
*ooLookupFieldBase::isNamed*

**SYNTAX**  
```cpp
virtual ooBoolean isNamed(
    const char *name) const;
```

**DESCRIPTION**  
Find out whether an input name matches the lookup field name.

**CLASS NAME**  
*ooEqualLookupField*

**CLASS HIERARCHY**  
*ooLookupFieldBase->ooEqualLookupField*

**CONSTRUCTOR**  
*ooEqualLookupField::ooEqualLookupField*

**SYNTAX**  
```cpp
1. ooEqualLookupField(
    const ooKeyField &field,
    const void *valuePtr);
```
2. ooEqualLookupField(
    const ooTypeNumber typeN,
    const char *memberName,
    const void *valuePtr);

DESCRIPTION Constructor for ooEqualLookupField class.

CLASS NAME ooGreaterThanLookupField
CLASS HIERARCHY ooLookupFieldBase->ooGreaterThanLookupField

CONSTRUCTOR ooGreaterThanLookupField::
    ooGreaterThanLookupField

SYNTAX 1. ooGreaterThanLookupField(
    const ooKeyField &field,
    const void *valuePtr);
2. ooGreaterThanLookupField(
    const ooTypeNumber typeN,
    const char *memberName,
    const void *valuePtr);

DESCRIPTION Constructor for ooGreaterThanLookupField class.

CLASS NAME ooLessThanLookupField
CLASS HIERARCHY ooLookupFieldBase->ooLessThanLookupField

CONSTRUCTOR ooLessThanLookupField::
    ooLessThanLookupField

SYNTAX 1. ooLessThanLookupField(
    const ooKeyField &field,
    const void *valuePtr);
2. ooLessThanLookupField(
    const ooTypeNumber typeN,
ooGreaterThanEqualLookupField

    const char *memberName,
    const void *valuePtr);

DESCRIPTION Constructor for ooGreaterThanEqualLookupField class.

CLASS NAME ooGreaterThanEqualLookupField
CLASS HIERARCHY ooLookupFieldBase->ooGreaterThanEqualLookupField

CONSTRUCTOR ooGreaterThanEqualLookupField::
    ooGreaterThanEqualLookupField
SYNTAX 1. ooGreaterThanEqualLookupField(
    const ooKeyField &field,
    const void *valuePtr);
2. ooGreaterThanEqualLookupField(
    const ooTypeNumber typeN,
    const char *memberName,
    const void *valuePtr);

DESCRIPTION Constructor for ooGreaterThanEqualLookupField class.

CLASS NAME ooLessThanEqualLookupField
CLASS HIERARCHY ooLookupFieldBase->ooLessThanEqualLookupField

CONSTRUCTOR ooLessThanEqualLookupField::
    ooLessThanEqualLookupField
SYNTAX 1. ooLessThanEqualLookupField(
    const ooKeyField &field,
    const void *valuePtr);
2. ooLessThanEqualLookupField(
    const ooTypeNumber typeN,
const char *memberName,
const void *valuePtr);

DESCRIPTION Constructor for ooLessThanEqualLookupField class.

CLASS NAME ooLookupKey

CLASS HIERARCHY ooLookupKey

CONSTRUCTOR ooLookupKey::ooLookupKey

SYNTAX 1. ooLookupKey(
    const ooTypeNumber typeN,
    const uint32 number);

2. ooLookupKey(
    const ooTypeNumber typeN,
    const char *memberName,
    const void *valuePtr);

DESCRIPTION Constructor for ooLookupKey class. Creates a lookup key.

MEMBER FUNCTION ooLookupKey::addField

SYNTAX uint32 addField(
    const ooLookupFieldBase &field);

DESCRIPTION Adds a lookup field to the end of the current series of fields in a lookup key.

MEMBER FUNCTION ooLookupKey::anyIndex

SYNTAX ooBoolean anyIndex(
    const ooHandle(ooSystemObj) &scopeH) const;

DESCRIPTION $scopeH$ can be a container, database, or federated database handle. An error will be issued if any other handle is given. Finds out whether
there is any index in the given scope to which the lookup key can compare. ooSystemObj is either ooContObj, ooDBObj, or ooFDObj.

RETURN VALUES

- oocTrue if there is a usable index
- oocFalse if there is no usable index

MEMBER FUNCTION ooLookupKey::nField

SYNTAX

```
uint32 nField() const;
```

DESCRIPTION

Gets the number of fields currently defined for a lookup key.

RETURN VALUES

The number of fields currently defined for the lookup key.
**String Classes**

This appendix describes the constructors, operators, and member functions for the Objectivity/DB string classes:

- Class `ooVString` uses a VArray of characters to implement a C++ string. Instances of this class are persistent.
- Class `ooString(N)` contains a VArray of characters and a fixed character array of length \(N\). This class is non-persistent, but an instance of the class may be persistent when it is contained within a persistent object.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooVString, d_String</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Class for variable-length strings based on VArrays of characters. (ODMG only) d_String is the ODMG equivalent to class ooVString.</td>
</tr>
<tr>
<td>CLASS HIERARCHY</td>
<td>ooVArray(ooChar) -&gt; ooVString</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>ooVString::ooVString</th>
</tr>
</thead>
</table>
| SYNTAX          | 1. ooVString();  
                   2. ooVString(const char *p);  
                   3. ooVString(const ooVString &s); |
| DESCRIPTION     | Constructors for the class ooVString.  
                   1. Generic constructor, no string is created.  
                   2. A string is created containing a copy of the characters pointed to by \(p\). If \(p\) is null, no string is created.  
                   3. A string is created containing a copy of the characters in \(s\). If the length of \(s\) is 0, no string is created. |
## ooVString::operator =

**SYNTAX**

```cpp
ooVString & operator = (const char * p);
```

**DESCRIPTION**

Assign a C++/C string to an ooVString. The characters in the string are replaced by the characters pointed to by `p`. If `p` is null, the effect is to delete the string.

## ooVString::operator []

**SYNTAX**

```cpp
char & operator [] (const uint32 index);
```

**DESCRIPTION**

Access operator. Returns the value of the indexed character. Objectivity/DB automatically checks whether the index is within the allowable range of the string.

## ooVString::operator +=

**SYNTAX**

```cpp
ooVString & operator += (const char * p);
```

**DESCRIPTION**

Concatenate two strings and return the resulting string in ooVString. The characters pointed to by `p` are added at the end of the string.

## ::operator ==

**SYNTAX**

```cpp
int ::operator == (const ooVString & left,
                   const ooVString & right);
```

**DESCRIPTION**

(ODMG only) Equality operator to compare ooVStrings.
OPERATOR :: operator ==
SYNTAX int ::operator ==(
    const ooVString &left,
    const char *right);
DESCRIPTION (ODMG only) Equality operator to compare ooVString and char *.

OPERATOR :: operator ==
SYNTAX int ::operator ==(
    const char *left,
    const ooVString &right);
DESCRIPTION (ODMG only) Equality operator to compare char * and ooVString.

OPERATOR :: operator !=
SYNTAX int ::operator !=(
    const ooVString &left,
    const ooVString &right);
DESCRIPTION (ODMG only) Inequality operator to compare ooVStrings.

OPERATOR :: operator !=
SYNTAX int ::operator !=(
    const ooVString &left,
    const char *right);
DESCRIPTION (ODMG only) Inequality operator to compare ooVString and char *.
OPERATOR  :: operator !=
SYNTAX    int :: operator !=(
          const char * left,
          const ooVString & right);
DESCRIPTION (ODMG only) Inequality operator to compare char * and ooVString.

OPERATOR  :: operator <
SYNTAX    int :: operator <(
          const ooVString & left,
          const ooVString & right);
DESCRIPTION (ODMG only) Less than operator to compare ooVStrings.

OPERATOR  :: operator <
SYNTAX    int :: operator <(
          const ooVString & left,
          const char * right);
DESCRIPTION (ODMG only) Less than operator to compare ooVString and char *.

OPERATOR  :: operator <
SYNTAX    int :: operator <(
          const char * left,
          const ooVString & right);
DESCRIPTION (ODMG only) Less than operator to compare char * and ooVString.
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int ::operator &gt; (const ooVString &amp; left, const ooVString &amp; right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Greater than operator to compare ooVStrings.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int ::operator &gt; (const ooVString &amp; left, const char * right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Greater than operator to compare ooVString and char *.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int ::operator &gt; (const char * left, const ooVString &amp; right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Greater than operator to compare char * and ooVString.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &lt;=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int ::operator &lt;= (const ooVString &amp; left, const ooVString &amp; right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Less than or equal to operator to compare ooVStrings.</td>
</tr>
<tr>
<td>OPERATOR</td>
<td>::operator &lt;=</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>int ::operator &lt;=(</td>
</tr>
<tr>
<td></td>
<td>const ooVString &amp;left,</td>
</tr>
<tr>
<td></td>
<td>const char *right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Less than or equal to operator to compare ooVString and char *.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &lt;=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int ::operator &lt;=(</td>
</tr>
<tr>
<td></td>
<td>const char *left,</td>
</tr>
<tr>
<td></td>
<td>const ooVString &amp;right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Less than or equal to operator to compare char * and ooVString.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &gt;=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int ::operator &gt;=(</td>
</tr>
<tr>
<td></td>
<td>const ooVString &amp;left,</td>
</tr>
<tr>
<td></td>
<td>const ooVString &amp;right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Greater than or equal to operator to compare ooVString.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &gt;=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int ::operator &gt;=(</td>
</tr>
<tr>
<td></td>
<td>const ooVString &amp;left,</td>
</tr>
<tr>
<td></td>
<td>const char *right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Greater than or equal to operator to compare ooVString and char *.</td>
</tr>
</tbody>
</table>
OPERATOR ::operator >=
SYNTAX int ::operator >=(
    const char * left,
    const ooVString & right);
DESCRIPTION (ODMG only) Greater than or equal to operator to compare char * and ooVString.

OPERATOR ooVString::operator char *
SYNTAX operator const char*() const;
DESCRIPTION Conversion operator to get the safe pointer to the string. Convert the ooVString to an object of type const char*.

MEMBER FUNCTION ooVString::head
SYNTAX char *head() const;
DESCRIPTION Access member function. Returns the pointer to the first character of the string.

MEMBER FUNCTION ooVString::length
SYNTAX uint32 length() const;
DESCRIPTION Returns the length of the string.

MEMBER FUNCTION ooVString::resize
SYNTAX ooStatus resize(const uint32 newLength);
DESCRIPTION Resize the string to a new length of newLength. The actual number of bytes allocated is newLength + 1 since an extra byte is reserved for the terminating null character, 0.
CLASS NAME\n\n**ooString**(\(N\))

DESCRIPTION
A parameterized string class that contains a VArray of characters and a fixed character array of length \(N\).

CONSTRUCTOR
\n\n**ooString**(\(N\))::\n
SYNTAX
1.  \texttt{ooString}(\(N\)) \();\
2.  \texttt{ooString}(\(N\)) (const char *\(p\));
3.  \texttt{ooString}(\(N\)) (const \texttt{ooString}(\(N\)) &\(s\));
4.  \texttt{ooString}(\(N\)) (const \texttt{ooVString} &\(s\));

DESCRIPTION
1. Constructor for the class \texttt{ooString}(\(N\)).
2. Default constructor. No string is constructed.
3. A string containing a copy of the characters pointed to by \(p\) is created.
4. A string containing a copy of characters in \(s\) is created.
5. A string containing a copy of characters in \(s\) is created.

OPERATOR
\n\n**ooString**(\(N\))::\texttt{operator }=\;

SYNTAX
\texttt{ooString}(\(N\)) &\texttt{operator }=(\texttt{const char }*\texttt{s});

DESCRIPTION
Characters in the string are replaced with the characters pointed to by \(s\).

OPERATOR
\n\n**ooString**(\(N\))::\texttt{operator }+=\;

SYNTAX
\texttt{ooString}(\(N\)) &\texttt{operator }+=\texttt{(const char }*\texttt{s2});

DESCRIPTION
Characters pointed to by \(s2\) are added at the end of the string.
OPERATOR  
\texttt{ooString(N) :: operator []}  
SYNTAX  
\begin{verbatim}
char &operator [](const uint32 index) const;
\end{verbatim}  
DESCRIPTION  
Returns the value of the indexed character. Objectivity/DB automatically checks whether the index is within the allowable range of the string.

OPERATOR  
\texttt{ooString(N) :: operator ==}  
SYNTAX  
1. \texttt{ooBoolean operator ==(const char *p) const;}
2. \texttt{ooBoolean operator ==(const ooString(N) &s) const;}
3. \texttt{ooBoolean operator ==(const ooVString &s) const;}
DESCRIPTION  
Compares two strings for equality.

OPERATOR  
\texttt{ooString(N) :: operator !=}  
SYNTAX  
1. \texttt{ooBoolean operator !=(const char *p) const;}
2. \texttt{ooBoolean operator !=(const ooString(N) &s) const;}
3. \texttt{ooBoolean operator !=(const ooVString &s) const;}
DESCRIPTION  
Compares two strings for inequality.

OPERATOR  
\texttt{ooString(N) :: operator char *}  
SYNTAX  
\texttt{operator const char*() const;}
DESCRIPTION  
Converts the string \texttt{ooString(N)} to an object of type \texttt{const char\*}.

OPERATOR  
\texttt{ooString(N) :: operator ooVString}  
SYNTAX  
\texttt{operator const ooVString() const;}
DESCRIPTION  
Converts the string \texttt{ooString(N)} to an object of type \texttt{ooVString}.  

\textit{String Classes}  
G-9
### MEMBER FUNCTION: `ooString(N)::head`

**SYNTAX**
```cpp
cchar *head() const;
```

**DESCRIPTION**
Returns a pointer to the first character of the string.

### MEMBER FUNCTION: `ooString(N)::length`

**SYNTAX**
```cpp
uint32 length() const;
```

**DESCRIPTION**
Returns the length of the string.

### MEMBER FUNCTION: `ooString(N)::resize`

**SYNTAX**
```cpp
ooStatus resize(const uint32 newLength);
```

**DESCRIPTION**
Resize the string to a new length of `newLength`. The actual number of bytes allocated is `newLength + 1` since an extra byte is reserved for the terminating null character, 0.
## Transaction Class

This appendix describes the constructor and member functions for the Objectivity/DB transaction class. Instances of this class are not persistent-capable.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooTrans, d_Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Transaction object class. (ODMG only) d_Transaction is the ODMG equivalent to class ooTrans.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>ooTrans::ooTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooTrans();</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Constructor for class ooTrans.</td>
</tr>
</tbody>
</table>
MEMBER FUNCTION ooTrans: abort
SYNTAX ooStatus abort(
    ooHandleMode mode = oocHandleToNull);
DESCRIPTION Terminates the currently active transaction and aborts (does not apply) changes to the federated database.
mode determines what happens to active handles of the transaction after it is aborted. The default, oocHandleToNull, converts all active handles to null. Alternatively, oocHandleToOID converts the handles to object identifiers (OIDs).
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooTrans::begin
SYNTAX void ooTrans::begin();
DESCRIPTION (ODMG only) An ODMG equivalent name for member function ooTrans::start to starts a new transaction. This member function calls ooTrans::start with no arguments, and uses its default values of oocNoMROW, oocTransNoWait, and oocInsensitive.

MEMBER FUNCTION ooTrans::checkpoint
SYNTAX void ooTrans::checkpoint();
DESCRIPTION (ODMG only) An ODMG equivalent name for member function ooTrans::commitAndHold.

MEMBER FUNCTION ooTrans::commit
SYNTAX ooStatus commit();
DESCRIPTION Terminates the currently active transaction and commits all changes made to the database. If the federated database is open, this member function will close it.
RETURN VALUES ooStatus (oocSuccess or oocError).
MEMBER FUNCTION  ooTrans::commitAndHold
SYNTAX  
    ooStatus commitAndHold(
        ooDowngradeMode mode = oocNoDowngrade);
where  
    mode  Lock downgrade mode. The default, oocNoDowngrade preserves all locks held by the transaction. If set to oocDowngradeAll, all locks are downgraded to read locks (MROW read if the transaction is an MROW transaction, and normal read locks otherwise).
DESCRIPTION  Terminates the currently active transaction, commits all changes made to the database, and implicitly starts a new transaction. All locks acquired during the transaction are preserved, or all locks are downgraded to read locks if mode is set to oocDowngradeAll.
RETURN VALUES  ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooTrans::isActive
SYNTAX  
    ooBoolean isActive();
DESCRIPTION  Checks whether or not a transaction is active.
RETURN VALUES  oocTrue if a transaction is active, otherwise oocFalse.

MEMBER FUNCTION  ooTrans::start
SYNTAX  
    ooStatus start(
        ooMode openMode = oocNoMROW,
        const int32 waitOption = oocTransNoWait,
        ooIndexMode indexMode = oocInsensitive);
where  
    openMode  Specifies whether this is an MROW (Multiple Readers, One Writer) transaction. The default, oocNoMROW, specifies a non-MROW transaction. Alternatively, oocMROW activates MROW for this transaction.
waitOption
Specifies whether or not to wait for locks held by other transactions. The default, oocTransNoWait, indicates that the wait option is not specified and to use the current lock waiting setting (set by the ooSetLockMode function). oocNoWait specifies to not wait for locks. oocWait specifies to wait forever for locks. An integer value, n, (where 1 <= n <= 14400) specifies to wait for n seconds. If n = 0, then it is treated as oocNoWait. If n is less than 0 or greater than 14400, then it is treated as oocWait.

indexMode
Specifies the index mode. There are three index modes, oocInsensitive, oocSensitive, and oocExplicitUpdate. The default, oocInsensitive, specifies that automatic updating of indexes occurs at commit time. oocSensitive specifies that updates to indexed fields are immediately and automatically reflected in the corresponding indexes. Immediate updates ensure that you do not have to wait until commit for changes to be reflected in the index. Scans of an index will be consistent. The updates, if any, of the index will be done when scan is called and before the index entries are returned, or at commit time if no scans are done.

oocExplicitUpdate gives you explicit control over changes to indexed objects during a transaction. In certain update-intensive applications, where database-wide and federated database-wide indexes are used, but where the updated members are not indexed, using oocExplicitUpdate will improve performance. You must call the ooUpdateIndexes function when using this mode.

A warning will be issued if ooUpdateIndexes is called when the index update mode is not oocExplicitUpdate.
DESCRIPTION

Starts a new transaction.

RETURN VALUES

ooStatus (oocSuccess or oocError).

MEMBER FUNCTION

ooTrans::upgrade

SYNTAX

ooStatus ooTrans::upgrade();

DESCRIPTION

Indicates that this transaction is an upgrade application for schema evolution. You must call this member function before calling ooFDObj::open and ooTrans::start. Objectivity/DB will issue an error if transaction is not an update transaction. Objectivity/DB will issue an error if the federated database does not include any pending changes that require an upgrade application.

RETURN VALUES

ooStatus (oocSuccess or oocError).
ooTrans, d_Transaction

H-6 Using Objectivity/C++
# VArray Classes

This appendix describes the constructors and member functions for the Objectivity/DB VArray classes. Instances of these classes are non-persistent. Elements of VArrays can be of most types, not only class types.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooVArray(type), d_Varray&lt;type&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Variable-size array (VArray) class. This class is non-persistent, but an instance of the class may be persistent when it is contained within a persistent object. This is a parameterized class, where type specifies the class of each element within the VArray. Member functions are defined on the class to allow dynamic resizing of the VArray. Elements of the VArray are guaranteed contiguous within virtual memory. (ODMG only) d_Varray&lt;type&gt; is the ODMG equivalent to the Objectivity/DB class ooVArray(type). Note that ooVArray(type) does not inherit from d_Collection&lt;type&gt;, which is not implemented by Objectivity/DB.</td>
</tr>
</tbody>
</table>

| CLASS HIERARCHY | ooVArray(type) |

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>ooVArray(type) : :ooVArray(type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>1. ooVArray(type) ();</td>
</tr>
<tr>
<td></td>
<td>2. ooVArray(type) (uint32 initSize);</td>
</tr>
<tr>
<td></td>
<td>3. ooVArray(type) (</td>
</tr>
<tr>
<td></td>
<td>ooVArray(type) &amp;arrayName);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Constructor for class ooVArray(type)</td>
</tr>
</tbody>
</table>
1. Default constructor, VArray remains uninitialized, the size of the VArray is 0.
2. Constructor allocates the VArray with \textit{initSize} number of elements.
3. Constructor to copy a VArray.

\begin{tabular}{|l|l|}
\hline
\textbf{OPERATOR} & \texttt{ooVArray\texttt{(type)}::operator =} \\
\textbf{SYNTAX} & \texttt{ooVArray\texttt{(type)} \&operator = (} \\
& \texttt{\texttt{ooVArray\texttt{(type)} \&arrayName});} \\
\textbf{DESCRIPTION} & Copies the VArray element-by-element. Automatically adjusts the VArray size. \\
\textbf{RETURN VALUES} & Reference to the VArray. \\
\hline
\end{tabular}

\begin{tabular}{|l|l|}
\hline
\textbf{OPERATOR} & \texttt{ooVArray\texttt{(type)}::operator []} \\
\textbf{SYNTAX} & \texttt{type \&operator\texttt{[]} (} \\
& \texttt{uint32 index);} \\
\textbf{DESCRIPTION} & Returns reference to the \texttt{index}th element of the VArray. This operation also performs array boundary checking to ensure integrity. \\
\textbf{RETURN VALUES} & Reference to \texttt{index}th element of the VArray. \\
\hline
\end{tabular}

\begin{tabular}{|l|l|}
\hline
\textbf{MEMBER FUNCTION} & \texttt{ooVArray\texttt{(type)}::cardinality} \\
\textbf{SYNTAX} & \texttt{uint32 ooVArray\texttt{(type)}::cardinality() const;} \\
\textbf{DESCRIPTION} & (ODMG only) An ODMG equivalent name for the member function \texttt{ooVArray\texttt{(type)}::size}. \\
\hline
\end{tabular}

\begin{tabular}{|l|l|}
\hline
\textbf{MEMBER FUNCTION} & \texttt{ooVArray\texttt{(type)}::create_iterator} \\
\textbf{SYNTAX} & \texttt{d\_Iterator\texttt{<type> ooVArray\texttt{(type)}::create\_iterator() const;} \\
\textbf{DESCRIPTION} & (ODMG only) Return an iterator for the \texttt{ooVArray\texttt{(type)}}. \\
\hline
\end{tabular}
MEMBER FUNCTION  ooVArray(type) :: elem
SYNTAX  
```c
    type &elem(
        uint32 index);
```
DESCRIPTION  Returns reference to the \texttt{index}'th element of the VArray but performs no array bounds checking.
RETURN VALUES  Reference to the \texttt{index}'th element of the VArray.

MEMBER FUNCTION  ooVArray(type) :: extend
SYNTAX  ooStatus extend(
```
        type &newValue);
```
DESCRIPTION  Allocates a new element at the end of the VArray, and sets it to the value \texttt{newValue}.
RETURN VALUES  ooStatus (\texttt{oocSuccess} or \texttt{oocError}).

MEMBER FUNCTION  ooVArray(type) :: head
SYNTAX  
```
    type arrayName *head();
```
DESCRIPTION  Returns the pointer to the first element of the VArray if the VArray exists. Otherwise it returns a null pointer value.
RETURN VALUES  Pointer to the first element of the VArray.

MEMBER FUNCTION  ooVArray(type) :: insert_element
SYNTAX  
```
    void ooVArray(type) :: insert_element(
        const type &element);
```
DESCRIPTION  (ODMG only) An ODMG equivalent name for the member function \texttt{ooVArray(type) :: extend}. Adds \texttt{element} to the end of the \texttt{ooVArray(type)}.
 MEMBER FUNCTION  ooVArray(type)::is_empty  
SYNTAX  int ooVArray(type)::is_empty() const;  
DESCRIPTION  (ODMG only) Return true if the ooVArray(type) is of size 0.

 MEMBER FUNCTION  ooVArray(type)::remove_all  
SYNTAX  void ooVArray(type)::remove_all();  
DESCRIPTION  (ODMG only) Change the ooVArray(type) size to 0.

 MEMBER FUNCTION  ooVArray(type)::replace_element_at  
SYNTAX  void ooVArray(type)::replace_element_at(  
            const type &value,  
            const Iterator<type> &iterator);  
DESCRIPTION  (ODMG only) Replace the ooVArray(type) element indicated by iterator with value.

 MEMBER FUNCTION  ooVArray(type)::replace_element_at  
SYNTAX  void ooVArray(type)::replace_element_at(  
            const type &value,  
            uint32 index);  
DESCRIPTION  (ODMG only) Replace the indexth ooVArray(type) element with value.
### MEMBER FUNCTION

`ooVArray(type)::resize`

**SYNTAX**

```cpp
ooStatus resize(uint32 newSize);
```

**DESCRIPTION**

The VArray is resized to contain the number of elements specified by `newSize`.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`.

---

### MEMBER FUNCTION

`ooVArray(type)::retrieve_element_at`

**SYNTAX**

```cpp
const type &ooVArray(type)::retrieve_element_at(uint32 index) const;
```

**DESCRIPTION**

(ODMG only) Return the `index`th `ooVArray(type)` element.

---

### MEMBER FUNCTION

`ooVArray(type)::set`

**SYNTAX**

```cpp
ooStatus set(uint32 index, type &newValue);
```

**DESCRIPTION**

Sets the `index`th element of the VArray to the value `newValue`.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`.

---

### MEMBER FUNCTION

`ooVArray(type)::size`

**SYNTAX**

```cpp
uint32 size();
```

**DESCRIPTION**

Returns number of elements in the VArray.

**RETURN VALUES**

Number of elements in the VArray.

---

### MEMBER FUNCTION

`ooVArray(type)::update`

**SYNTAX**

```cpp
ooStatus update();
```

**DESCRIPTION**

Ensures that any modifications will be saved, allowing the use of the `elem` member function or `[]` operator on the left side of an assignment.
ooTVArray(type)

rather than using the set member function. If the persistent object containing the VArray is open in read mode, this member function will attempt to upgrade its access to update mode.

RETURN VALUES

ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooArray(type)::upper_bound

SYNTAX

uint32 ooArray(type)::upper_bound() const;

DESCRIPTION (ODMG only) An ODMG equivalent name for the member function ooArray(type)::size.

CLASS NAME ooTVArray (type)

DESCRIPTION Variable-size array (VArray) class. This is a parameterized class, where type specifies the class of each element within the VArray. Member functions are defined on the class to allow dynamic resizing of the VArray. The member functions defined in ooTVarray are similar to those defined in ooArray. The semantics of resize, extend, and VArray copy are different.

CONSTRUCTOR ooTVArray (type)::ooTVArray (type)

SYNTAX

1. ooTVArray (type) ();
2. ooTVArray (type) (uint32 initSize);
3. ooTVArray (type) (ooTVArray (type) &varray);

DESCRIPTION Constructor for class ooTVArray(type).

1. Default constructor, temporary VArray remains uninitialized, the size of the temporary VArray is 0.
2. Constructor allocates the temporary VArray with initSize number of elements.
3. Constructor to copy a temporary VArray.
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>ooTVArray(type)::operator =</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooTVArray (type) &amp;operator=(</td>
</tr>
<tr>
<td></td>
<td>ooTVArray(type) &amp;arrayName);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Copies the temporary VArray element-by-element. Automatically adjusts the temporary VArray size.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>Reference to the temporary VArray.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>ooTVArray(type)::operator []</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>type &amp;operator[](uint32 index);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Returns reference to the \textit{index}th element of the temporary VArray. This operation also performs array boundary checking to ensure integrity.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>Reference to the \textit{index}th element of the temporary VArray.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooTVArray(type)::elem</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>type &amp;elem(uint32 index);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Returns reference to the \textit{index}th element of the temporary VArray but performs no array bounds checking.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>Reference to the \textit{index}th element of the temporary VArray.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooTVArray(type)::extend</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus extend(type &amp;newElem);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Allocates a new element at the end of the temporary VArray, and sets it to the value \textit{newElem}.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>ooStatus (oocSuccess or oocError).</td>
</tr>
</tbody>
</table>
ooTVArray(type)

MEMBER FUNCTION  

**ooTVArray(type)::head**

**SYNTAX**  

`type *head() const;`

**DESCRIPTION**  
Returns the pointer to the first element of the VArray if the VArray exists. Otherwise it returns a null pointer value.

**RETURN VALUES**  
Pointer to the first element of the VArray.

MEMBER FUNCTION  

**ooTVArray(type)::resize**

**SYNTAX**  

`ooStatus resize(uint32 newSize);`

**DESCRIPTION**  
The temporary VArray is resized to contain the number of elements specified by `newSize`.

**RETURN VALUES**  
`ooStatus (oocSuccess or oocError)`.

MEMBER FUNCTION  

**ooTVArray(type)::set**

**SYNTAX**  

`ooStatus set(uint32 index, type &newValue);`

**DESCRIPTION**  
Sets the `index`th element of the temporary VArray to the value `newValue`.

**RETURN VALUES**  
`ooStatus (oocSuccess or oocError)`.

MEMBER FUNCTION  

**ooTVArray(type)::size**

**SYNTAX**  

`uint32 size();`

**DESCRIPTION**  
Returns number of elements in the temporary VArray.

**RETURN VALUES**  
Number of elements in the temporary VArray.
MEMBER FUNCTION  ooTVArray(type) :: update

SYNTAX  

ooStatus update();

DESCRIPTION  oocSuccess is returned because a temporary VArray can always be updated.

RETURN VALUES  ooStatus (oocSuccess or oocError).
Iterator Classes

This appendix describes the constructors and member functions for the Objectivity/DB iterator classes. Iterators allow you to traverse collections of objects. Instances of these classes are non-persistent.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(className) &` are interchangeable with `const ooRef(className) &`.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooItr(ooObj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Iterator class for Basic Objects.</td>
</tr>
<tr>
<td>CLASS HIERARCHY</td>
<td>ooHandle(ooObj) -&gt; ooItr(ooObj)</td>
</tr>
</tbody>
</table>

**CONSTRUCTOR**

```
ooItr(ooObj)::ooItr
```

**SYNTAX**

```
();
```

**DESCRIPTION**

Default constructor for class `ooItr(ooObj)`.

**MEMBER FUNCTION**

```
next
```

**SYNTAX**

```
ooBoolean next();
```

**DESCRIPTION**

Sets iterator to reference the next object of the current iteration.

**RETURN VALUES**

- `oocTrue` if there is a next object within the iteration, `oocFalse` if all of the objects in the iteration have been traversed or if an error occurred.
MEMBER FUNCTION ooItr(ooObj)::scan
SYNTAX ooStatus scan(
        const ooHandle(className) & objH,
        const ooMode openmode = oocNoOpen);
DESCRIPTION Initializes an iterator to traverse all basic objects in the container, database, or federated database referenced by \texttt{objH}.
RETURN VALUES ooStatus (oocSuccess or oocError).
SEE ALSO ooItr(ooObj)::next

MEMBER FUNCTION ooItr(ooObj)::scan
SYNTAX ooStatus scan(
        const ooHandle(ooSystemObj) & scopeH,
        const ooLookupKey & lookupKey,
        const ooMode openmode = oocNoOpen);
DESCRIPTION Initializes the iterator with only those persistent objects that are both in the scanned scope and satisfy the condition specified by \texttt{lookupkey}.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooItr(ooObj)::typeN
SYNTAX ooTypeNumber typeN();
DESCRIPTION Gets class type number of currently referenced object.
**CLASS NAME**

**ooItr(ooContObj)**

**DESCRIPTION**

Iterator class for containers.

**CLASS HIERARCHY**

ooHandle(ooObj) -> ooItr(ooContObj)

---

**CONSTRUCTOR**

**ooItr(ooContObj)::ooItr**

**SYNTAX**

```cpp
ooItr(ooContObj)();
```

**DESCRIPTION**

Default constructor for class ooItr(ooContObj).

---

**MEMBER FUNCTION**

**ooItr(ooContObj)::next**

**SYNTAX**

```cpp
ooBoolean next();
```

**DESCRIPTION**

Sets the iterator to reference the next container in the current iteration.

**RETURN VALUES**

Returns oocTrue if there is a next container within the iteration, and oocFalse if all of the containers in the iteration have been traversed or if an error occurred.

---

**MEMBER FUNCTION**

**ooItr(ooContObj)::scan**

**SYNTAX**

```cpp
ooStatus scan(
    const ooHandle(className) &objH,
    const ooMode openmode = oocNoOpen);
```

**DESCRIPTION**

Initializes an iterator to traverse all containers in the database or federated database referenced by objH.

**RETURN VALUES**

ooStatus (oocSuccess or oocError).
MEMBER FUNCTION  ooItr(ooContObj)::typeN
SYNTAX  ooTypeNumber typeN();
DESCRIPTION  Gets class type number of currently referenced object.
### CLASS NAME
**ooItr(ooDBObj)**

### DESCRIPTION
Iterator class for databases.

### CLASS HIERARCHY
ooHandle(ooObj) -> ooItr(ooDBObj)

### CONSTRUCTOR
**ooItr(ooDBObj)::ooItr**

**SYNTAX**

```c
ooItr(ooDBObj)();
```

**DESCRIPTION**
Constructor for class ooItr(ooDBObj).

### MEMBER FUNCTION
**ooItr(ooDBObj)::next**

**SYNTAX**

```c
ooBoolean next();
```

**DESCRIPTION**
Sets iterator to reference the next object of the current iteration.

**RETURN VALUES**
oocTrue if there is a next object within the iteration, oocFalse if all of the objects in the iteration have been traversed or if an error occurred.

### MEMBER FUNCTION
**ooItr(ooDBObj)::scan**

**SYNTAX**

```c
ooStatus scan(
    const ooHandle(ooFDObj) & fdH,
    const ooMode openmode = oocNoOpen);
```

**DESCRIPTION**
Initializes an iterator to traverse all databases contained in the federated database referenced by *fdH*.

**RETURN VALUES**
ooStatus (oocSuccess or oocError).
**CLASS NAME**  
d_Iterator

**DESCRIPTION**  
(ODMG only) Class d_Iterator supports iteration over objects of type d_Varray. The construction of a d_Iterator for a d_Varray will pin the d_Varray.

**CONSTRUCTOR**  
d_Iterator::d_Iterator  
**SYNTAX**  
d_Iterator::d_Iterator (  
    const d_VArray<type> &);  
**DESCRIPTION**  
(ODMG only) Copy constructor to construct an iterator from a d_VArray.

**CONSTRUCTOR**  
d_Iterator::d_Iterator  
**SYNTAX**  
d_Iterator::d_Iterator (  
    const d_Iterator<type> &);  
**DESCRIPTION**  
(ODMG only) Copy constructor for class d_Iterator.

**OPERATOR**  
d_Iterator::operator =  
**SYNTAX**  
d_Iterator<type> &d_Iterator::operator = (  
    const d_Iterator<type> &);  
**DESCRIPTION**  
(ODMG only) Assignment operator to assign a const iterator to a non-const iterator.

**OPERATOR**  
d_Iterator::operator ++  
**SYNTAX**  
d_Iterator<type> &d_Iterator::operator ++();  
**DESCRIPTION**  
(ODMG only) Prefix increment operator that move the iterator forward to the next element. Equivalent to advance.
**d_Iterator**

---

**OPERATOR**

d_Iterator::operator ++

**SYNTAX**

d_Iterator<type> d_Iterator::operator ++(int);

**DESCRIPTION**

(ODMG only) Postfix increment operator to move the iterator forward to the next element. Equivalent to `advance`.

---

**MEMBER FUNCTION**

d_Iterator::advance

**SYNTAX**

void d_Iterator::advance();

**DESCRIPTION**

(ODMG only) Move the iterator forward to the next element. If the iteration has not begun, it sets the iterator to the first element in the iteration.

---

**MEMBER FUNCTION**

d_Iterator::get_element

**SYNTAX**

const type &d_Iterator::get_element() const;

**DESCRIPTION**

(ODMG only) Returns a constant reference to the current element. This function returns the first element at the start of the iteration. Elsewhere in the iteration, it returns the current element. Unlike `d_Iterator::next`, this function returns a constant reference to an element, instead of doing an object assignment. Because of this, `d_Iterator::get_element` may be preferable where the element type has deep copy semantics and the application cannot afford the performance overhead of copying the object.

**RETURN VALUES**

Signals an error if there are no elements remaining in the iteration.

---

**MEMBER FUNCTION**

d_Iterator::next

**SYNTAX**

int d_Iterator::next (type &objReference);

**DESCRIPTION**

(ODMG only) Checks for end of iteration and advances the iterator if not at the end. This function returns the first element at the start of the iteration. Elsewhere in the iteration, it returns the current element.
This function performs an object assignment, whereas `get_element` returns a constant reference to the element.

**RETURN VALUES** Signals an error if there are no elements remaining in the iteration.

---

**MEMBER FUNCTION**  
**d_Iterator::not_done**

**SYNTAX**  
```cpp
int d_Iterator::not_done() const;
```

**DESCRIPTION**  
(ODMG only) Checks if iteration is complete or not.

**RETURN VALUES**  
Returns true if there are more elements in the iteration and false if the iteration is complete.

---

**MEMBER FUNCTION**  
**d_Iterator::reset**

**SYNTAX**  
```cpp
void d_Iterator::reset();
```

**DESCRIPTION**  
(ODMG only) Resets the iterator to the start of the iteration.
**CLASS NAME**

**ooQuery**

**DESCRIPTION**

Class ooQuery supports iteration over sequences of objects that are not directly supported by the Objectivity/DB predicate query feature. For example, you can use ooQuery to iterate over a VArray of object references, such as ooVArray(ooRef(userClass)).

---

**MEMBER FUNCTION**

**ooQuery::setup**

**SYNTAX**

```
ooStatus ooQuery::setup(
    char *predicate,
    ooTypeNumber typeN,
    oqrUserOperatorSet * = 0);
```

**DESCRIPTION**

Sets the predicate string, type number, and user operator set for the query.

**RETURN VALUES**

ooStatus (oocSuccess or oocError).

---

**MEMBER FUNCTION**

**ooQuery::evaluate**

**SYNTAX**

```
ooBoolean ooQuery::evaluate(
    ooHandle(ooObj) &objH);
```

**DESCRIPTION**

Evaluates the query for the object referenced by objH.

**RETURN VALUES**

oocTrue if object is a match for predicate defined by ooQuery::setup, and oocFalse otherwise.
Object Reference and Handle Classes

This appendix describes the constructors, operators, and member functions for the object reference (ooRef) and handle (ooHandle) classes for the Objectivity/DB persistent classes ooObj, ooContObj, ooDBObj, and ooFDObj.

Where the syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form const ooHandle(className) & are interchangeable with const ooRef(className) &.

ooRef and ooHandle Inheritance Hierarchy

When you run the DDL Processor on a schema file, DDL generates ooRef and ooHandle classes that follow the same inheritance hierarchy as the corresponding user-defined persistent class.

Type-specific member functions for the ooRef and ooHandle classes on user-defined persistent classes override those found in their base classes. For example, given class className that inherits from ooObj, the function

```
className *ooHandle(className)::operator ->;
```

in class ooHandle(className) overrides the corresponding function

```
ooObj *ooHandle(ooObj)::operator ->;
```

in class ooHandle(ooObj).
Relationship Between ooRef and ooHandle Classes

The ooRef and ooHandle classes provide the same functionality and can be used virtually interchangeably. ooHandle classes maintain additional state information that allows them to more efficiently access multiple fields of the same object than ooRef. Several member functions only available for ooHandle classes rely on the additional state information; these member functions include operator ooObj *, and operator *.

ooRef and ooHandle class member functions provide prototypes that accept either ooRef or ooHandle objects explicitly or by relying on implicit type conversions between ooRef and ooHandle objects.

ooRef classes may be embedded in persistent classes, but ooHandle classes may not.

Basic Objects (ooObj)

This section describes the object reference and handle classes for basic objects. Where the syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooRef (ooObj), d_Ref&lt;ooObj&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Object reference class for basic objects. All other object reference classes are derived from this one. d_Ref&lt;ooObj&gt; is the ODMG equivalent name for ooRefHandle(ooObj).</td>
</tr>
<tr>
<td>CLASS HIERARCHY</td>
<td>ooRef(ooObj)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooHandle (ooObj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Handle class for basic objects. All other object handle classes are derived from this one. ooHandle provides the same member functions as ooRef, as well as operator *, and operator ooObj *.</td>
</tr>
<tr>
<td>CLASS HIERARCHY</td>
<td>ooHandle(ooObj)</td>
</tr>
<tr>
<td>CONSTRUCTOR</td>
<td>ooRefHandle(ooObj)::ooRefHandle(ooObj)</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>ooRefHandle(ooObj)::ooRefHandle(ooObj)();</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Default constructor for class ooRef(ooObj) and ooHandle(ooObj).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>ooRefHandle(ooObj)::ooRefHandle(ooObj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooRefHandle(ooObj)::ooRefHandle(ooObj) { const ooRef(ooObj) &amp;objR};</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Constructor for class ooRef(ooObj) and ooHandle(ooObj).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>ooRefHandle(ooObj)::ooRefHandle(ooObj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooRefHandle(ooObj)::ooRefHandle(ooObj) { const ooHandle(ooObj) &amp;objH};</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Constructor for class ooRef(ooObj) and ooHandle(ooObj).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>ooRefHandle(ooObj)::ooRefHandle(ooObj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooRefHandle(ooObj)::ooRefHandle(ooObj) { const ooObj *objP};</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Constructor for class ooRef(ooObj) and ooHandle(ooObj).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>ooRefHandle(ooObj)::ooRefHandle(ooObj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooHandle(ooObj)::ooHandle(ooObj) { const ooRef(ooFDObj) &amp;objR};</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Constructor for class ooRef(ooObj) and ooHandle(ooObj).</td>
</tr>
</tbody>
</table>
### CONSTRUCTOR

**ooRefHandle (ooObj) :: ooRefHandle (ooObj)**

**SYNTAX**

\[
\text{ooRefHandle (ooObj) :: ooRefHandle (ooObj)} (\text{const d_Ref_Any &from});
\]

**DESCRIPTION**

(ODMG only) Copy constructor for class ooRef (ooObj) and ooHandle (ooObj). Sets this ooRefHandle (ooObj) to from.

**RETURN VALUES**

Signals an error if from is non-null and references an object that is not of type ooObj, signal an error.

### OPERATOR

**ooRefHandle (ooObj) :: operator ->**

**SYNTAX**

\[
\text{const ooObj *ooRefHandle (ooObj) :: operator -> (); const;}
\]

**DESCRIPTION**

Returns pointer to the basic object in virtual memory. Also opens the basic object for read access if it is not open.

**RETURN VALUES**

Pointer to basic object in virtual memory.

### OPERATOR

**ooHandle (ooObj) :: operator ***

**SYNTAX**

\[
\text{const ooRef (ooObj) ooHandle (ooObj) :: operator *(); const;}
\]

**DESCRIPTION**

Returns reference to the basic object in virtual memory. Also opens the basic object for read access if it is not open. The basic object is pinned in memory only for the lifetime of the handle. This operation is not available for ooRef (ooObj).

**RETURN VALUES**

Reference to basic object in virtual memory.

### OPERATOR

**ooRefHandle (ooObj) :: operator =**

**SYNTAX**

1. \[
\text{ooRefHandle (ooObj)}
\]

\[
\&\text{ooRefHandle (ooObj) :: operator = (}
\text{const ooRef (ooObj) &objR);}
\]

2. \[
\text{ooRefHandle (ooObj)}
\]

\[
\&\text{ooRefHandle (ooObj) :: operator = (}
\text{const ooShortRef (ooObj) &shortObjR);}
\]

---

**K-4 Using Objectivity/C++**
3. \texttt{ooRefHandle(ooObj)} & \texttt{ooRefHandle(ooObj)::operator = (const ooHandle(ooObj) & objH)};

4. \texttt{ooRefHandle(ooObj)} & \texttt{ooRefHandle(ooObj)::operator = (const ooObj * objP)};

5. \texttt{ooRefHandle(ooObj)} & \texttt{ooRefHandle(ooObj)::operator = (const ooRef(ooFDObj) & objR)};

\textbf{DESCRIPTION} \hspace{1cm} Sets object reference or handle to reference specified basic object.

The database and container identified by the \texttt{ooRef} or \texttt{ooHandle} are used as the scope of \texttt{shortObjR}. The value of \texttt{shortObjR} specifies the page and slot of the referenced object in the container.

\textbf{RETURN VALUES} \hspace{1cm} Reference to object reference or handle.

\textbf{OPERATOR} \hspace{1cm} \texttt{ooRefHandle(ooObj)::operator ==}

\textbf{SYNTAX} 1. \texttt{ooBoolean ooRefHandle(ooObj)::operator == (const ooRef(ooObj) & objR) const;}

2. \texttt{ooBoolean ooRefHandle(ooObj)::operator == (const ooShortRef(ooObj) & shortObjR) const;}

3. \texttt{ooBoolean ooRefHandle(ooObj)::operator == (const ooHandle(ooObj) & objH) const;}

4. \texttt{ooBoolean ooRefHandle(ooObj)::operator == (const ooObj * objP) const;}

\textbf{DESCRIPTION} \hspace{1cm} Tests for equality between the object reference or handle and the specified basic object or value.

\textbf{RETURN VALUES} \hspace{1cm} \texttt{oocTrue} if the object reference or handle references the specified basic object, \texttt{oocFalse} otherwise.

\textbf{OPERATOR} \hspace{1cm} \texttt{ooRefHandle(ooObj)::operator !=}

\textbf{SYNTAX} 1. \texttt{ooBoolean ooRefHandle(ooObj)::operator != (const ooRef(ooObj) & objR) const;}

2. \texttt{ooBoolean ooRefHandle(ooObj)::operator != (const ooShortRef(ooObj) & shortObjR) const;}
3. `ooBoolean ooRefHandle(ooObj)::operator !=(const ooHandle(ooObj) &objH) const;`  
   DESCRIPTION Tests for inequality between the object reference or handle and the specified basic object or value.  
   RETURN VALUES `oocTrue` if the object reference or handle does not reference the specified basic object, `oocFalse` otherwise.

4. `ooBoolean ooRefHandle(ooObj)::operator !=(const ooObj* objP) const;`  

**OPERATOR**

<table>
<thead>
<tr>
<th>ooRefHandle(ooObj)::operator d_Ref_Any</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SYNTAX</strong></td>
</tr>
<tr>
<td><strong>DESCRIPTION</strong></td>
</tr>
</tbody>
</table>

**MEMBER FUNCTION**

<table>
<thead>
<tr>
<th>ooRef(ooObj)::operator int</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SYNTAX</strong></td>
</tr>
<tr>
<td><strong>DESCRIPTION</strong></td>
</tr>
<tr>
<td><strong>RETURN VALUES</strong></td>
</tr>
</tbody>
</table>

**MEMBER FUNCTION**

<table>
<thead>
<tr>
<th>ooHandle(ooObj)::operator const ooObj*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SYNTAX</strong></td>
</tr>
<tr>
<td><strong>DESCRIPTION</strong></td>
</tr>
<tr>
<td><strong>RETURN VALUES</strong></td>
</tr>
</tbody>
</table>
### MEMBER FUNCTION

**ooHandle(ooObj)::operator ooRef(ooObj)**

**SYNTAX**

```cpp
operator ooHandle(ooObj)::operator ooRef(ooObj)()
const;
```

**DESCRIPTION**

Returns basic object identifier of the handle. Note: this operation is implicitly defined for `ooRef(ooObj)`.

**RETURN VALUES**

Object identifier of the handle.

---

### MEMBER FUNCTION

**ooRefHandle(ooObj)::checkin**

**SYNTAX**

```cpp
ooStatus ooRefHandle(ooObj)::checkin() const;
```

**DESCRIPTION**

Checks in the basic object.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`.

---

### MEMBER FUNCTION

**ooRefHandle(ooObj)::checkout**

**SYNTAX**

```cpp
ooStatus ooRefHandle(ooObj)::checkout(const ooMode mode) const;
```

**DESCRIPTION**

Checks out the basic object in the indicated mode.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`.

---

### MEMBER FUNCTION

**ooRefHandle(ooObj)::close**

**SYNTAX**

```cpp
ooStatus ooRefHandle(ooObj)::close() const;
```

**DESCRIPTION**

Closes the basic object referenced by the object reference or handle.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`.
**MEMBER FUNCTION**  
**ooRefHandle (ooObj) :: containedIn**

**SYNTAX**
1. `ooHandle(ooContObj)`
   ```cpp
   ooRefHandle(ooObj)::containedIn() const;
   ```
2. `ooHandle(ooContObj)`
   ```cpp
   &ooRefHandle(ooObj)::containedIn( 
   ooHandle(ooContObj) & contH) const;
   ```
3. `ooRef(ooContObj)`
   ```cpp
   &ooRefHandle(ooObj)::containedIn( 
   ooRef(ooContObj) & contH) const;
   ```

**DESCRIPTION**  
Returns object reference or handle to container in which the basic object is contained.

**RETURN VALUES**  
Object reference or handle of container that contains the referenced basic object.

**MEMBER FUNCTION**  
**ooRefHandle (ooObj) :: copy**

**SYNTAX**
1. `ooRef(ooObj) &ooRefHandle(ooObj)::copy(`
   ```cpp
   const ooHandle(ooObj) & nearH, 
   ooRef(ooObj) & newObjR) const;
   ```
2. `ooHandle(ooObj) &ooRefHandle(ooObj)::copy(`
   ```cpp
   const ooHandle(ooObj) & nearH, 
   ooHandle(ooObj) & newObjH) const;
   ```
3. `ooHandle(ooObj) ooRefHandle(ooObj)::copy(`
   ```cpp
   const ooHandle(ooObj) & nearH) const;
   ```

**DESCRIPTION**  
Creates a copy of a basic object, returning the handle or object reference of the new basic object.

**RETURN VALUES**  
`ooStatus (oocSuccess or oocError)`.

**MEMBER FUNCTION**  
**ooRefHandle (ooObj) :: delete_object**

**SYNTAX**
```cpp
void ooRefHandle(ooObj)::delete_object();
```

**DESCRIPTION**  
(ODMG only) Delete the object referenced by this `ooRefHandle(ooObj)`.
### MEMBER FUNCTION

**ooRefHandle(ooObj)::getDefaultVers**

**SYNTAX**

1. `ooStatus ooRefHandle(ooObj)::getDefaultVers( ooRef(ooObj) & defR) const;`
2. `ooStatus ooRefHandle(ooObj)::getDefaultVers( ooHandle(ooObj) & defH) const;`

**DESCRIPTION**

Returns object reference or handle to default version of the basic object.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`.

---

### MEMBER FUNCTION

**ooRefHandle(ooObj)::getNameObj**

**SYNTAX**

`ooStatus ooRefHandle(ooObj)::getNameObj( ooItr(ooObj) & objI) const;`

**DESCRIPTION**

Initializes iterator `objI` to find all named basic objects in a given scope.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`.

---

### MEMBER FUNCTION

**ooRefHandle(ooObj)::getNameScope**

**SYNTAX**

`ooStatus ooRefHandle(ooObj)::getNameScope( ooItr(ooObj) & objI) const;`

**DESCRIPTION**

Initializes iterator `objI` to traverse basic objects that serve as a name scope.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`.
### MEMBER FUNCTION

**ooRefHandle (ooObj) :: getNextVers**

**SYNTAX**

```c
ooStatus ooRefHandle(ooObj)::getNextVers(
    ooHandle(ooObj) & nextVersItrH,
    const ooMode openMode = oocNoOpen) const;
```

**DESCRIPTION**

Initializes `nextVersItrH` iterator to traverse all next versions of the basic object. Objects of the iteration are automatically opened in mode `openMode`. If no next versions exist, an empty iterator is returned (the first `next` member function invocation will return `oocFalse`).

**RETURN VALUES**

`ooStatus` (`oocSuccess` or `oocError`).

### MEMBER FUNCTION

**ooRefHandle (ooObj) :: getObjName**

**SYNTAX**

1. `char *ooRefHandle(ooObj)::getObjName(
   const ooHandle(ooObj) & scopeH) const;`
2. `char *ooRefHandle(ooObj)::getObjName() const;`

**DESCRIPTION**

Returns the name of the basic object in the given scope.

**RETURN VALUES**

Name of the basic object in the scope specified by `scopeH`. Default scope basic object is `oovTopDB`. The string is statically allocated by the member function and overwritten with each invocation. For this reason it is recommended that the user copy the string to locally allocated storage if it is to be used later.

### MEMBER FUNCTION

**ooRefHandle (ooObj) :: getPrevVers**

**SYNTAX**

1. `ooStatus ooRefHandle(ooObj)::getPrevVers(
   ooRef(ooObj) & prevR) const;`
2. `ooStatus ooRefHandle(ooObj)::getPrevVers(
   ooHandle(ooObj) & prevH) const;`

**DESCRIPTION**

Returns object reference or handle to previous version of the basic object.

**RETURN VALUES**

`ooStatus` (`oocSuccess` or `oocError`).

---

K-10 Using Objectivity/C++
MEMBER FUNCTION  
\texttt{ooRefHandle(ooObj)::getVersStatus}

SYNTAX  
\texttt{ooVersMode ooRefHandle(ooObj)::getVersStatus() const;}

DESCRIPTION  
Returns current versioning mode of referenced basic object.

RETURN VALUES  
Current versioning mode is either oocNoVers, oocLinearVers, or oocBranchVers.

MEMBER FUNCTION  
\texttt{ooRefHandle(ooObj)::isNull}

SYNTAX  
\texttt{int ooRefHandle(ooObj)::isNull();}

DESCRIPTION  
(ODMG only) An ODMG equivalent name for the member function \texttt{ooRefHandle(ooObj)::isNull}.

MEMBER FUNCTION  
\texttt{ooRefHandle(ooObj)::isValid}

SYNTAX  
\texttt{ooBoolean ooRefHandle(ooObj)::isValid() const;}

DESCRIPTION  
Checks the validity of an object reference or handle, but does not check the type. A value of oocTrue returned by isValid indicates only that the object reference or handle is valid; that is, that it points to a basic object in the federated database; it does not indicate that the type of that basic object corresponds to the type of the object reference or handle.

RETURN VALUES  
oocTrue if the object reference or handle is valid.

MEMBER FUNCTION  
\texttt{ooRefHandle(ooObj)::lock}

SYNTAX  
\texttt{ooStatus ooRefHandle(ooObj)::lock( const ooLockMode lockMode) const;}

DESCRIPTION  
Explicitly locks the basic object and propagates the lock to any basic objects associated through links that have lock propagation enabled.

RETURN VALUES  
ooStatus (oocSuccess or oocError).
MEMBER FUNCTION  ooRefHandle(ooObj) :: lockNoProp

SYNTAX  ooStatus ooRefHandle(ooObj)::lockNoProp(
    const ooLockMode lockMode) const;

DESCRIPTION  Explicitly locks the basic object without propagation semantics.

RETURN VALUES  ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooRefHandle(ooObj) :: lookupObj

SYNTAX  1.  ooStatus ooRefHandle(ooObj)::lookupObj (const ooHandle(ooObj) & scopeH,
    const ooKey &keyStruct,
    const ooMode openMode = oocRead);  
2.  ooStatus ooRefHandle(ooObj)::lookupObj (const ooKey &keyStruct,
    const ooMode openMode = oocRead);
3.  ooStatus ooRefHandle(ooObj)::lookupObj (const ooHandle(ooObj) & scopeH,
    const char *name,
    const ooMode openMode = oocRead);
4.  ooStatus ooRefHandle(ooObj)::lookupObj (const char *name,
    const ooMode openMode = oocRead);

DESCRIPTION  Sets object reference or handle to reference the basic object matching
the key structure keyStruct or named name in the scope of the handle
scopeH, and opens the basic object in mode openMode (default is
oocRead). Default scope object is oovTopDB. This member function is
not type safe.

RETURN VALUES  ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooRefHandle(ooObj) :: move

SYNTAX  ooStatus ooRefHandle(ooObj)::move (const ooHandle(ooObj) & targetContH);
DESCRIPTION  Moves a basic object to a different container.

RETURN VALUES  ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooRefHandle (ooObj)::nameObj

SYNTAX  1. ooStatus ooRefHandle (ooObj)::nameObj(  
       const ooHandle (ooObj) & scopeH,  
       const char * name ) const;

2. ooStatus ooRefHandle (ooObj)::nameObj(  
       const char * name ) const;

DESCRIPTION  Assigns the referenced basic object the scope name name in the scope specified by handle scopeH. Default scope is oovTopDB.

RETURN VALUES  ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooRefHandle (ooObj)::open

SYNTAX  ooStatus ooRefHandle (ooObj)::open(  
       const ooMode openMode = oocRead ) const;

DESCRIPTION  Opens the referenced basic object in the access mode specified by openMode.

Valid values for openMode are the constants oocRead and oocUpdate. The default open mode is oocRead.

Any number of processes may open the same basic object for read access as long as no process has it open for update access. No process may open a basic object for update access if any other process has it open for read access. No other process may open a basic object if any other process has it open for update access unless MROW is in effect.

The access restrictions imposed by a process's access to a basic object applies until the transaction in which the basic object was opened is terminated (either committed or aborted).

RETURN VALUES  ooStatus (oocSuccess or oocError).
MEMBER FUNCTION  ooRefHandle(ooObj) :: openMode
SYNTAX          ooMode ooRefHandle(ooObj)::openMode () const;
DESCRIPTION     Returns the mode in which the basic object currently referenced has
                been opened.
RETURN VALUES   If basic object is not open through this object reference or handle, returns oocNoOpen.
                If basic object is open through this object reference or handle: if it has
                been opened for update using any handle, returns oocUpdate; otherwise, returns oocRead.

MEMBER FUNCTION  ooRefHandle(ooObj) :: print
SYNTAX          void ooRefHandle(ooObj)::print(
                FILE *fp = stdout) const;
DESCRIPTION      Prints information about the object, including its object identifier
                (OID), class, and open mode.

MEMBER FUNCTION  ooRefHandle(ooObj) :: ptr
SYNTAX          ooObj * ooRefHandle(ooObj)::ptr();
DESCRIPTION      (ODMG only) Return a pointer to the object referenced by this
                ooRefHandle(ooObj). This member function pins the object in
                memory.

MEMBER FUNCTION  ooRefHandle(ooObj) :: set_container
SYNTAX          1. ooStatus ooRef(ooObj)::set_container(
                const ooRef(ooObj) &objR);
                2. ooStatus ooHandle(ooObj)::set_container(
                const ooHandle(ooObj) &objH);
DESCRIPTION      Sets the context container before assigning a short OID to an object
                reference or handle. The object reference or handle is not valid after
invoking set_container because only partial information about a basic object is stored in the object reference or handle.

RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooRefHandle(ooObj)::setDefaultVers
SYNTAX ooStatus ooRefHandle(ooObj)::setDefaultVers() const;
DESCRIPTION Sets default version to the current basic object.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooRefHandle(ooObj)::setVersStatus
SYNTAX ooStatus ooRefHandle(ooObj)::setVersStatus(const ooVersMode versMode) const;
DESCRIPTION Sets the versioning mode for the referenced basic object to versMode.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooRefHandle(ooObj)::sprint
SYNTAX char void *ooRefHandle(ooObj)::sprint(char *buf = 0) const;
DESCRIPTION Returns a string representation of the object’s identifier (OID). If a pointer is passed in, sprint stores the string in the buffer; otherwise the string is stored in a statically allocated buffer and should be copied before the next call to sprint.
RETURN VALUES Pointer to the buffer containing the string.

MEMBER FUNCTION ooRefHandle(ooObj)::typeN
SYNTAX ooTypeNumber ooRefHandle(ooObj)::typeN() const;
DESCRIPTION Returns the type number of the referenced basic object.
RETURN VALUES

Type number of the referenced basic object. You must not modify the string in any manner. Doing so may result in unexpected program errors.

MEMBER FUNCTION ooRefHandle(ooObj)::typeName

SYNTAX

char *ooRefHandle(ooObj)::typeName() const;

DESCRIPTION

Return string containing class name of the basic object currently referenced. The string must be treated as read only.

RETURN VALUES

Class name of basic object currently referenced.

MEMBER FUNCTION ooRefHandle(ooObj)::unnameObj

SYNTAX

1. ooStatus ooRefHandle(ooObj)::unnameObj(const ooHandle(ooObj) & scopeH, const char *name = 0) const;
2. ooStatus ooRefHandle(ooObj)::unnameObj(const char *name = 0) const;

DESCRIPTION

Removes the basic object name name in the scope specified by scopeH. Default scope object is oovTopDB.

RETURN VALUES

ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooRefHandle(ooObj)::update

SYNTAX

ooStatus ooRefHandle(ooObj)::update() const;

DESCRIPTION

Opens the referenced basic object in update access mode.

RETURN VALUES

ooStatus (oocSuccess or oocError).
Containers (ooContObj)

This section describes the object reference and handle classes for containers. Where the syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle.

---

**CLASS NAME**

<table>
<thead>
<tr>
<th>ooRef (ooContObj)</th>
</tr>
</thead>
</table>

**DESCRIPTION**

Object reference class for containers.

**CLASS HIERARCHY**

ooRef(ooObj)->ooRef(ooContObj)

**INHERITED MEMBERS**

- ooRef(ooObj)::operator==
- ooRef(ooObj)::operator!=
- ooRef(ooObj)::operator int
- ooRef(ooObj)::checkin
- ooRef(ooObj)::checkout
- ooRef(ooObj)::getNameScope
- ooRef(ooObj)::getObjName
- ooRef(ooObj)::isValid
- ooRef(ooObj)::lock
- ooRef(ooObj)::nameObj
- ooRef(ooObj)::typeN
- ooRef(ooObj)::typeName
- ooRef(ooObj)::unnameObj
- ooRef(ooObj)::update

---

**CLASS NAME**

<table>
<thead>
<tr>
<th>ooHandle (ooContObj)</th>
</tr>
</thead>
</table>

**DESCRIPTION**

Handle class for containers.

**CLASS HIERARCHY**

ooHandle(ooObj)->ooHandle(ooContObj)

**INHERITED MEMBERS**

- ooHandle(ooObj)::operator==
- ooHandle(ooObj)::operator!=
- ooHandle(ooObj)::operator int
- ooHandle(ooObj)::checkin
ooHandle(ooContObj)

ooHandle(ooObj)::checkout
ooHandle(ooObj)::getNameScope
ooHandle(ooObj)::getName
ooHandle(ooObj)::isValid
ooHandle(ooObj)::lock
ooHandle(ooObj)::nameObj
ooHandle(ooObj)::typeN
ooHandle(ooObj)::typeName
ooHandle(ooObj)::unnameObj
ooHandle(ooObj)::update

### CONSTRUCTOR

**ooRefHandle(ooContObj)::**

**ooRefHandle(ooContObj)**

**SYNTAX**

`ooRefHandle(ooContObj)::ooRefHandle(ooContObj)();`

**DESCRIPTION**

Default constructor for class `ooRef(ooContObj)` and `ooHandle(ooContObj)`.

### CONSTRUCTOR

**ooRefHandle(ooContObj)::**

**ooRefHandle(ooContObj)**

**SYNTAX**

`ooRefHandle(ooContObj)::ooRefHandle(ooContObj) (const ooRef(ooContObj) &objR);`

**DESCRIPTION**

Constructor for class `ooRef(ooContObj)` and `ooHandle(ooContObj)`.

### CONSTRUCTOR

**ooRefHandle(ooContObj)::**

**ooRefHandle(ooContObj)**

**SYNTAX**

`ooRefHandle(ooContObj)::ooRefHandle(ooContObj) (const ooHandle(ooContObj) &objH);`

**DESCRIPTION**

Constructor for class `ooRef(ooContObj)` and `ooHandle(ooContObj)`. 
CONSTRUCTOR  ooRefHandle(ooContObj)::  ooRefHandle(ooContObj)  
SYNTAX  ooRefHandle(ooContObj)::ooRefHandle(ooContObj) (  
   const ooContObj *objP);  
DESCRIPTION  Constructor for class ooRef(ooContObj) and  
   ooHandle(ooContObj).

CONSTRUCTOR  ooRefHandle(ooContObj)::  ooRefHandle(ooContObj)  
SYNTAX  ooRefHandle(ooContObj)::ooRefHandle(ooContObj) (  
   const d_Ref_Any &from);  
DESCRIPTION  (ODMG only) Set this ooRefHandle(ooContObj) to from.  
RETURN VALUES  Signals an error if from is non-null and references an object that is not  
   of type ooContObj, signal an error.

OPERATOR  ooRefHandle(ooContObj)::operator ->  
SYNTAX  const ooContObj *ooRefHandle(ooContObj)::operator ->()  
   const;  
DESCRIPTION  Returns pointer to the container in virtual memory. Also opens the  
   container for read access if it is not open.  
RETURN VALUES  Pointer to container in virtual memory.

OPERATOR  ooHandle(ooContObj)::operator *  
SYNTAX  const ooRef(ooContObj)  
   ooHandle(ooContObj)::operator *() const;  
DESCRIPTION  Returns reference to the container in virtual memory. Also opens the  
   container for read access if it is not open. The container is pinned in  
   memory only for the lifetime of the handle.
K-20   Using Objectivity/C++

This operation is not available for ooRef(ooContObj).

RETURN VALUES Reference to container in virtual memory.

OPERATOR ooRefHandle(ooContObj)::operator =
SYNTAX 1. ooRefHandle(ooContObj)
            &ooRefHandle(ooContObj)::operator =(
                const ooRef(ooContObj) objR);
2. ooRefHandle(ooContObj)
            &ooRefHandle(ooContObj)::operator =(
                const ooHandle(ooContObj) & objH);
3. ooRefHandle(ooContObj)
            &ooRefHandle(ooContObj)::operator =(
                const ooContObj * objP);
DESCRIPTION Sets object reference or handle to reference specified container.
RETURN VALUES Reference to object reference or handle.

OPERATOR ooHandle(ooContObj)::operator const ooContObj*
SYNTAX operator ooHandle(ooContObj)::operator const ooContObj*() const;
DESCRIPTION Returns pointer to the container in virtual memory. Object is pinned in memory only for the lifetime of the handle.
Note: this operation is not available for ooRef because only handles can temporarily pin containers in memory.
RETURN VALUES Pointer to container in virtual memory.

OPERATOR ooRefHandle(ooContObj)::operator d_Ref_Any
SYNTAX ooRefHandle(ooContObj)::operator d_Ref_Any() const;
DESCRIPTION (ODMG only) Return this ooRefHandle(ooContObj) as a d_Ref_Any.
MEMBER FUNCTION  
`ooRefHandle (ooContObj) :: close`

SYNTAX  
`ooStatus ooRefHandle (ooContObj) :: close () const;`

DESCRIPTION  
Explicitly closes the container.

RETURN VALUES  
`ooStatus (oocSuccess or oocError).`

----

MEMBER FUNCTION  
`ooRefHandle (ooContObj) :: containedIn`

SYNTAX  
1.  
   `ooHandle (ooDBObj) ooRefHandle (ooContObj) :: containedIn () const;`

2.  
   `ooHandle (ooDBObj) & ooRefHandle (ooContObj) :: containedIn ( ooHandle (ooDBObj) & dbH ) const;`

3.  
   `ooRef (ooDBObj) & ooRefHandle (ooContObj) :: containedIn ( ooRef (ooDBObj) & dbH ) const;`

DESCRIPTION  
Returns object reference or handle to database in which the container is contained.

RETURN VALUES  
Object reference or handle of database that contains the referenced container.

----

MEMBER FUNCTION  
`ooRefHandle (ooContObj) :: contains`

SYNTAX  
`ooStatus ooRefHandle (ooContObj) :: contains ( ooItr (ooObj) & objI, const ooMode openMode = oocNoOpen ) const;`

DESCRIPTION  
Initializes the iterator `objI` to traverse all referenced basic objects in the container. Objects of the iteration are automatically opened in the mode specified by `openMode`; the default is `oocNoOpen` (do not open the container automatically).

RETURN VALUES  
`ooStatus (oocSuccess or oocError).`
MEMBER FUNCTION  ooRefHandle (ooContObj) :: convertObjects
SYNTAX       ooStatus  ooRefHandle (ooContObj) :: convertObjects()
DESCRIPTION Convert all objects affected by schema evolution operations within a container to their current class representations. This will also bring all indexes up to date. Calling this member function more than once on a specific container has no effect.
RETURN VALUES oocSuccess if successful, or oocError if the federated database is opened for read-only access.

MEMBER FUNCTION  ooRefHandle (ooContObj) :: exist
SYNTAX       ooBoolean  ooRefHandle (ooContObj) :: exist(
                          const ooHandle(ooDBObj) & dbH,
                          const char * contSysName,
                          const ooMode openMode  = oocNoOpen);
DESCRIPTION Test for the existence of a container. Returns the constant oocTrue if the container with system name contSysName exists in the database referenced by handle dbH, otherwise returns oocFalse. If the container exists and openMode is oocRead or oocUpdate, the container is opened and its object reference or handle is initialized to the container. If openMode is oocRead or oocUpdate, and exist returns oocFalse, the container either does not exist or the container exists but is not accessible, and the object reference or handle is set to null. If openMode is oocNoOpen and the container exists, the container is not opened, but the object reference or handle is initialized to the container. If openMode is oocNoOpen and the container does not exist, the container is not opened, and the object reference or handle is set to null.
RETURN VALUES oocTrue if the specified container exists.
<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooRefHandle (ooContObj)::hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int32 ooRefHandle (ooContObj)::hash () const;</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Returns the hash value used as a clustering factor for an indexed container. The hash value is the number of sequentially indexed objects to place into a page when creating named or keyed objects within the container. The hash value is set when the container is created. See ooContObj::operator new for more information.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>Returns 0 if the container is not hashed; otherwise, returns a non-zero value. Returns -1 if the handle is null or the container can not be opened.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooRefHandle (ooContObj)::isUpdated</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooBoolean ooRefHandle (ooContObj)::isUpdated () const;</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Checks whether the container has already been updated and committed by another process. The checking is performed in the middle of a transaction.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>Returns oocTrue only if the container has been updated by another process since being opened by the current process for an MROW read operation. It returns oocFalse in all other cases.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooRefHandle (ooContObj)::lockNoProp</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus ooRefHandle (ooContObj)::lockNoProp ( const ooLockMode lockMode ) const;</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Explicitly locks the container without propagation semantics.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>ooStatus (oocSuccess or oocError).</td>
</tr>
</tbody>
</table>
MEMBER FUNCTION  ooRefHandle(ooContObj)::lookupObj

SYNTAX  1. ooStatus ooRefHandle(ooContObj)::lookupObj(
  const ooHandle(ooObj) & scopeH,
  const char *name,
  const ooMode openMode = oocRead) const;
  2. ooStatus ooRefHandle(ooContObj)::lookupObj(
  const char *name,
  const ooMode openMode = oocRead) const;

DESCRIPTION  Sets object reference or handle to reference the container with the name name in the scope of the object specified by the handle scopeH, and opens the container in the mode specified by openMode. Default scope object is oovTopDB.

RETURN VALUES  ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooRefHandle(ooContObj)::name

SYNTAX  char *ooRefHandle(ooContObj)::name() const;

DESCRIPTION  Returns the system name of the referenced container. The string is statically allocated by the member function and is overwritten with each invocation (you may wish to copy the string to locally allocated storage rather than storing the pointer).

RETURN VALUES  System name of the container.

MEMBER FUNCTION  ooRefHandle(ooContObj)::nPage

SYNTAX  uint32 ooRefHandle(ooContObj)::nPage() const;

DESCRIPTION  Returns current number of pages in the container.

RETURN VALUES  Returns 0 if the handle is null or the container can not be opened.
MEMBER FUNCTION  ooRefHandle (ooContObj)::open

SYNTAX  
1. ooStatus ooRefHandle (ooContObj)::open(
    const ooMode openMode = oocRead) const;
2. ooStatus ooRefHandle (ooContObj)::open(
    const char * contSysName,
    const ooMode openMode = oocRead);
3. ooStatus ooRefHandle (ooContObj)::open(
    const ooHandle (ooDBObj) & dbH,
    const char * contSysName,
    const ooMode openMode = oocRead);

DESCRIPTION  Opens the referenced container in the access mode specified by openMode, either oocRead and oocUpdate. The default open mode is oocRead.

1. Opens the container referenced by ooRefHandle (ooContObj).
2. Opens the container specified by contSysName.
3. Opens the container specified by contSysName and the database in which the container resides, dbH.

Any number of processes may open the same container for read access as long as no process has it open for update access. No process may open a container for update access if any other process has it open for read access. No other process may open a container if any other process has it open for update access unless MROW is in effect. The access restrictions imposed by a process’s access to a container applies until the transaction in which the container was opened is terminated (either committed or aborted).

RETURN VALUES  ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooRefHandle (ooContObj)::openMode

SYNTAX  ooMode ooRefHandle (ooContObj)::openMode () const;

DESCRIPTION  Returns the mode in which the currently referenced container has been opened.

RETURN VALUES  If container is not open through this object reference or handle, returns oocNoOpen.
If container is open through this object reference or handle: if it has been opened for update using any handle, returns oocUpdate; otherwise, returns oocRead.

**MEMBER FUNCTION**

`ooRefHandle (ooContObj) :: percentGrow`

**SYNTAX**

```c++
uint32 ooRefHandle (ooContObj) :: percentGrow () const;
```

**DESCRIPTION**

Returns the growth factor for the container. The container grows by this percent of its current size. The percent growth is set when the container is created. See `ooContObj::operator new` for more information.

**RETURN VALUES**

Returns 0 if the handle is null or the container can not be opened.

**MEMBER FUNCTION**

`ooRefHandle (ooObj) :: ptr`

**SYNTAX**

```c++
ooContObj * ooRefHandle (ooContObj) :: ptr ();
```

**DESCRIPTION**

(ODMG only) Return a pointer to the object referenced by this `ooRefHandle (ooContObj)`. This member function pins the object in memory.

**MEMBER FUNCTION**

`ooRefHandle (ooContObj) :: refreshOpen`

**SYNTAX**

```c++
ooStatus ooRefHandle (ooContObj) :: refreshOpen (
    const ooMode openMode,
    ooBoolean *pIsUpdated) const;
```

**DESCRIPTION**

Refresh an MROW container by getting the most recent version updated and committed by another process. `openMode` is the mode in which to reopen the container and `pIsUpdated` is an `ooBoolean` pointer whose value corresponds to the result of the member function `ooHandle(ooContObj) :: isUpdated`.

**RETURN VALUES**

`ooStatus (oocSuccess or oocError)`.
Databases (ooDBObj)

This section describes the object reference and handle classes for databases. Where the syntax and parameter information is identical for the object reference and handle classes, ooRefHandle is used to represent either ooRef or ooHandle.

---

**CLASS NAME**  ooRef (ooDBObj)

**DESCRIPTION**  Object reference class for databases.

**CLASS HIERARCHY**  ooRef(ooObj) -> ooRef(ooDBObj)

**INHERITED MEMBERS**

- ooHandle(ooObj)::operator ==
- ooHandle(ooObj)::operator !=

---

**CLASS NAME**  ooHandle (ooDBObj)

**DESCRIPTION**  Handle class for databases.

**CLASS HIERARCHY**  ooHandle(ooObj) -> ooHandle(ooDBObj)

---

**CONSTRUCTOR**  ooRefHandle (ooDBObj)  ::  ooRefHandle (ooDBObj)

**SYNTAX**

1. ooRefHandle(ooDBObj)  ::  ooRefHandle (ooDBObj) ;
2. ooRefHandle(ooDBObj)  ::  ooRefHandle (ooDBObj) ( const ooRef(ooDBObj) & objR );
3. ooRefHandle(ooDBObj)  ::  ooRefHandle (ooDBObj) ( const ooHandle(ooDBObj) & objH );
4. ooRefHandle(ooDBObj)  ::  ooRefHandle (ooDBObj) ( const ooDBObj * objP );

**DESCRIPTION**  Constructor for class ooRef(ooDBObj) and ooHandle(ooDBObj).
OPERATOR

*ooRefHandle (ooDBObj)::*operator =

SYNTAX

1. ooRefHandle (ooDBObj)  
   &ooRefHandle (ooDBObj)::operator = (  
      const ooRef (ooDBObj) objR);
2. ooRefHandle (ooDBObj)  
   &ooRefHandle (ooDBObj)::operator = (  
      const ooHandle (ooDBObj) &objH);
3. ooRefHandle (ooDBObj)  
   &ooRefHandle (ooDBObj)::operator = (  
      const ooDBObj *objP);

DESCRIPTION
Sets object reference or handle to reference specified database.

RETURN VALUES
Reference to object reference or handle.

MEMBER FUNCTION

*ooRefHandle (ooDBObj)::*change

SYNTAX

ooStatus ooRefHandle (ooDBObj)::change (  
   const char* newDBSysName = 0,  
   const char* newHostName = 0,  
   const char* newLocalPathName = 0,  
   FILE* fp = stdout) const;

where

newDBSysName New system name of the database. This feature is currently not implemented. Always pass the value of zero for this parameter.
newHostName New name of the host on which the database file resides
newLocalPathName New local path name of the database file. The database file name should be the last component of the path name.
fp Pointer to a transcript file to which the information for the database is echoed before and after invoking this member function. The default is stdout.
DESCRIPTION (for administration) Change database attributes. The newHostName and newLocalPathName determine the network address of the database file.

Use this member function to rename or physically move a database file to a new location. Objectivity/DB tries to get an exclusive update lock on the corresponding database when the member function is invoked. If it cannot obtain the lock, the member function returns an error status.

If Objectivity/DB obtains the exclusive update lock, this member function logically renames or relocates the database file within the federated database. However, you must then physically move or rename the file using the operating system command to move a file.

RETURN VALUES Returns a non-zero value for ooStatus if successful. If there are multiple images of the database, an error is signalled.

EXAMPLE This code sets the database file host name to myHost and the local path name of the database file to /mnt/john/design/adder.ecad.DB.

```c++
 ooHandle(ooDBObj) dbH;
 ...
 // dbH is a valid handle to a database
 // Set the new host name of the database to be myHost and
 // new local path name to be
 // /mnt/john/design/adder.ecad.DB
 dbH.change(0, "myHost",
            "/mnt/john/design/adder.ecad.DB");
```

MEMBER FUNCTION ooRefHandle(ooDBObj)::close

SYNTAX ooStatus ooRefHandle(ooDBObj)::close() const;

DESCRIPTION Closes the referenced database.

RETURN VALUES ooStatus (oocSuccess or oocError).
MEMBER FUNCTION  ooRefHandle (ooDBObj)::containedIn

SYNTAX
1. ooHandle(ooFDObj)
   ooRefHandle(ooDBObj)::containedIn() const;
2. ooHandle(ooFDObj)
   &ooRefHandle(ooDBObj)::containedIn(
      ooHandle(ooFDObj) &fdH) const;
3. ooRef(ooFDObj)
   &ooRefHandle(ooDBObj)::containedIn(
      ooRef(ooFDObj) &fdH) const;

DESCRIPTION Returns object reference or handle to federated database in which the database is contained.

RETURN VALUES Object reference or handle of the federated database that contains the referenced database.

MEMBER FUNCTION  ooRefHandle (ooDBObj)::contains

SYNTAX  ooStatus ooRefHandle(ooDBObj)::contains(
            ooItr(ooContObj) &contI,
            const ooMode openMode = oocNoOpen) const;

DESCRIPTION Initializes the iterator contI to traverse all containers in the referenced database. Objects of the iteration are automatically opened in the mode specified by openMode. The default is oocNoOpen (do not open the container automatically).

RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooRefHandle (ooDBObj)::convertObjects

SYNTAX  ooStatus ooRefHandle(ooDBObj)::convertObjects()

DESCRIPTION Convert all objects affected by schema evolution operations within a database to their current class representations.

RETURN VALUES oocSuccess if successful, or oocError if the federated database is opened for read-only access.
MEMBER FUNCTION  ooRefHandle (ooDBObj) :: exist

SYNTAX  

ooBoolean ooRefHandle (ooDBObj) :: exist (  
   const ooHandle (ooFDObj) & fdH ,  
   const char * dbSysName ,  
   const ooMode openMode = oocNoOpen );

DESCRIPTION  Tests for the existence of a database. Returns the constant oocTrue if  
the database with system name dbSysName exists in the federated  
database referenced by handle fdH, otherwise returns oocFalse. If the  
database exists and openMode is oocRead or oocUpdate, the database  
is opened and its object reference or handle is initialized to the  
database. If openMode is oocRead or oocUpdate, and exist returns  
oocFalse, the database either does not exist or the database exists but  
is not accessible, and the object reference or handle is set to null. If  
openMode is oocNoOpen and the database exists, the database is not  
opened, but the object reference or handle is initialized to the database.  
If openMode is oocNoOpen and the database does not exist, the  
database is not opened, and the object reference or handle is set to null.

RETURN VALUES  The constant oocTrue if a database with system name dbSysName  
exists in the federated database referenced by handle fdH; the constant  
oocFalse otherwise.

MEMBER FUNCTION  ooRefHandle (ooDBObj) :: fileName

SYNTAX  

char* ooRefHandle (ooDBObj) :: fileName () const ;

DESCRIPTION  (for administration) Returns string containing full file name of the  
database file, including path information. The string is statically  
allocated by the member function and is overwritten with each  
invocation. You should copy the string if you wish to use it later in your  
application.

RETURN VALUES  Full file name of the database file, including path information. If there  
are multiple images of the database, an error is signalled.

EXAMPLE  In this example, the handle dbH is set to reference the database with  
system name testDb located in the federated database with system  
name testFd. Information about the corresponding database file is  
printed out.
ooHandle(ooDBObj) dbH;
ooHandle(ooFDObj) fdH;
...

// dbH is set to reference database with system name testDb
fdH.open("testFd", oocRead);
dbH.open(fdH, "testDb", oocRead);

// Get and print out info on database file
printf("hostname: %s\n", dbH.hostName());
printf("pathname: %s\n", dbH.pathName());
printf("filename: %s\n", dbH.fileName());

SEE ALSO
“ooRefHandle(ooDBObj)::hostName” and “ooRefHandle(ooDBObj)::pathName”

MEMBER FUNCTION ooRefHandle(ooDBObj)::getDefaultContObj

SYNTAX 1. ooHandle(ooContObj)
    ooRefHandle(ooDBObj)::getDefaultContObj(
    const ooMode openMode = oocNoOpen) const;

2. ooHandle(ooContObj)
    &ooRefHandle(ooDBObj)::getDefaultContObj(
    ooHandle(ooContObj) & contH,
    const ooMode openMode = oocNoOpen) const;

3. ooRef(ooContObj)
    &ooRefHandle(ooDBObj)::getDefaultContObj(
    ooRef(ooContObj) & contR,
    const ooMode openMode = oocNoOpen) const;

DESCRIPTION Returns object reference or handle of the default container for the database.

RETURN VALUES Object reference or handle of the default container for the database.

MEMBER FUNCTION ooRefHandle(ooDBObj)::hostName

SYNTAX char *ooRefHandle(ooDBObj)::hostName() const;

DESCRIPTION (for administration) Returns string containing name of the network host on which the database file is located. The string is statically
allocated by the member function and is overwritten with each invocation. You should copy the string if you wish to use it later in your application.

RETURN VALUES
Name of network host where database file is located. If there are multiple images of the database, an error is signalled.

SEE ALSO
“ooRefHandle(ooDBObj)::fileName” and “ooRefHandle(ooDBObj)::pathName”

MEMBER FUNCTION ooRefHandle(ooDBObj)::isValid
SYNTAX ooBoolean ooRefHandle(ooDBObj)::isValid() const;
DESCRIPTION Checks the validity of an object reference or handle.
RETURN VALUES oocTrue if the object reference or handle is valid.

MEMBER FUNCTION ooRefHandle(ooDBObj)::lock
SYNTAX ooStatus ooRefHandle(ooDBObj)::lock( const ooLockMode lockMode ) const;
DESCRIPTION Explicitly locks the database and propagates the lock to any objects associated through links that have lock propagation enabled.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooRefHandle(ooDBObj)::name
SYNTAX char *ooRefHandle(ooDBObj)::name() const;
DESCRIPTION Returns the system name of the referenced database. The string is statically allocated by the member function and is overwritten with each invocation (you may wish to copy the string to locally allocated storage rather than storing the pointer).
RETURN VALUES Name of the database.
MEMBER FUNCTION  

\texttt{ooRefHandle(ooDBObj)::numContObjs}

SYNTAX  

\begin{verbatim}
unsigned long ooRefHandle(ooDBObj)::numContObjs()
const;
\end{verbatim}

DESCRIPTION  

Returns the number of containers in a database.

---

MEMBER FUNCTION  

\texttt{ooRefHandle(ooDBObj)::open}

SYNTAX  

1. \begin{verbatim}
ooStatus ooRefHandle(ooDBObj)::open(
    const ooMode openMode=oocRead) const;
\end{verbatim}

2. \begin{verbatim}
ooStatus ooRefHandle(ooDBObj)::open(
    const ooHandle(ooFDObj) & fdH,
    const char *dbSysName,
    const ooMode openMode = oocRead);
\end{verbatim}

DESCRIPTION  

Opens the referenced database in the access mode specified by \texttt{openMode}, either \texttt{oocRead} and \texttt{oocUpdate}. The default open mode is \texttt{oocRead}.

1. Opens the database referenced by \texttt{ooRefHandle(ooDBObj)}.

2. Opens the database specified by \texttt{dbSysName} and the federated database in which the database resides, \texttt{fdH}.

RETURN VALUES  

\texttt{ooStatus (oocSuccess or oocError)}.

---

MEMBER FUNCTION  

\texttt{ooRefHandle(ooDBObj)::openMode}

SYNTAX  

\begin{verbatim}
ooMode ooRefHandle(ooDBObj)::openMode() const;
\end{verbatim}

DESCRIPTION  

Returns the mode in which the currently referenced database has been opened.

RETURN VALUES  

If the database is not open through this object reference or handle, returns \texttt{oocNoOpen}.

If the database is open through this object reference or handle: if it has been opened for update using any handle, returns \texttt{oocUpdate}; otherwise, returns \texttt{oocRead}.
MEMBER FUNCTION  ooRefHandle (ooDBObj)::pathName

SYNTAX  char* ooRefHandle (ooDBObj)::pathName() const;

DESCRIPTION  (for administration) Returns string containing full path of the directory where the database file is located. The string is statically allocated by the member function and is overwritten with each invocation. You should copy the string if you wish to use it later in your application.

RETURN VALUES  Path of directory where database file is located. If there are multiple images of the database, an error is signalled.

SEE ALSO  "ooRefHandle (ooDBObj)::hostName" and "ooRefHandle (ooDBObj)::fileName"

MEMBER FUNCTION  ooRefHandle (ooDBObj)::tidy

SYNTAX  ooStatus ooRefHandle (ooDBObj)::tidy(
    FILE* fp = stdout,
    const char* hostName = 0,
    const char* pathName = 0) const;

where

fp  Pointer to the file in which to store the audit report generated by running the member function

hostName  Name of the host machine on which to create the scratch file

pathName  Specifies the path for the scratch file.

DESCRIPTION  (for administration) Use the tidy member function to invoke the ootidy tool from within an application to tidy a database.

When Objectivity/DB is tidying a database, it needs a temporary scratch file to hold the intermediate data. By default, this scratch file is created in the directory that contains the database file. The hostName and pathName arguments allow you to control where the scratch file is created. If either hostName or pathName is zero or empty (""), the scratch file is created in the default directory.
When using this member function, you must follow these steps:
1. Start a transaction.
2. Open the database for update.
3. Invoke the `ooRefHandle(ooDBObj)::tidy` member function.
4. Immediately commit the transaction.

You cannot manipulate any database, container, or basic object within the transaction before invoking the `ooRefHandle(ooDBObj)::tidy` member function. This member function performs physical storage compaction and relocation. After calling this function, some of the cached system data is in an inconsistent state, so it is important that you immediately commit the transaction so that the system data can be reinitialized to a known state.

You cannot abort a transaction after calling this member function.

**RETURN VALUES**

Returns a non-zero value for `ooStatus` if successful.

**EXAMPLE**

This code starts a transaction, opens a federated database for update, opens a database for update, calls the `ooRefHandle(ooDBObj)::tidy` member function, and then commits the transaction:

```cpp
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooHandle(ooDBObj) dbH;
...
trans.start();
fdH.open("Documentation", oocUpdate);
dbH.open(fdH, "Introduction", oocUpdate);
dbH.tidy();
trans.commit();
```

---

**MEMBER FUNCTION**

`ooRefHandle(ooDBObj)::typeN`

**SYNTAX**

`ooTypeNumber ooRefHandle(ooDBObj)::typeN() const;`

**DESCRIPTION**

Returns the type number of `ooDBObj`.

**RETURN VALUES**

Type number of the database.
MEMBER FUNCTION ooRefHandle(ooDBObj)::typeName
SYNTAX char *ooRefHandle(ooDBObj)::typeName() const;
DESCRIPTION Returns the string "ooDBObj". The string must be treated as read only.
RETURN VALUES Class name of currently referenced database.

MEMBER FUNCTION ooRefHandle(ooDBObj)::update
SYNTAX ooStatus ooRefHandle(ooDBObj)::update() const;
DESCRIPTION Opens the referenced database in update access mode.
RETURN VALUES ooStatus (oocSuccess or oocError).
This section describes the object reference and handle classes for federated databases. Where the syntax and parameter information is identical for the object reference and handle classes, `ooRefHandle` is used to represent either `ooRef` or `ooHandle`.

**CLASS NAME**  
**ooRef (ooFDObj)**

**DESCRIPTION** Object reference class for federated databases.

**CLASS HIERARCHY**  
`ooRef (ooObj) -> ooRef (ooFDObj)`

**CLASS NAME**  
**ooHandle (ooFDObj)**

**DESCRIPTION** Handle class for federated databases.

**CLASS HIERARCHY**  
`ooHandle (ooObj) -> ooHandle (ooFDObj)`

**INHERITED MEMBERS**  
`ooHandle (ooObj)::operator ==
`  
`ooHandle (ooObj)::operator !`.

**CONSTRUCTOR**  
`ooRefHandle (ooFDObj)::ooRefHandle (ooFDObj)`

**SYNTAX**
1. `ooRefHandle (ooFDObj)::ooRefHandle (ooFDObj)();`
2. `ooRefHandle (ooFDObj)::ooRefHandle (ooFDObj) (const ooRef (ooFDObj) &objR);`
3. `ooRefHandle (ooFDObj)::ooRefHandle (ooFDObj) (const ooHandle (ooFDObj) &objH);`
4. `ooRefHandle (ooFDObj)::ooRefHandle (ooFDObj) (const ooFDObj *objP);`

**DESCRIPTION** Constructor for class `ooRef (ooFDObj)` and `ooHandle (ooFDObj)`. 
**OPERATOR**

\[
\text{ooRefHandle}(\text{ooFDObj})::\text{operator} =
\]

**SYNTAX**

1. \[
\text{ooRefHandle}(\text{ooFDObj})
&\text{ooRefHandle}(\text{ooFDObj})::\text{operator} = (\text{const} \text{ooRef}(\text{ooFDObj}) \text{ objR});
\]

2. \[
\text{ooRefHandle}(\text{ooFDObj})
&\text{ooRefHandle}(\text{ooFDObj})::\text{operator} = (\text{const} \text{ooHandle}(\text{ooFDObj}) & \text{objH});
\]

3. \[
\text{ooRefHandle}(\text{ooFDObj})
&\text{ooRefHandle}(\text{ooFDObj})::\text{operator} = (\text{const} \text{ooFDObj} * \text{objP});
\]

**DESCRIPTION**

Sets object reference or handle to reference specified federated database.

**RETURN VALUES**

Reference to object reference or handle.

**MEMBER FUNCTION**

\[
\text{ooRefHandle}(\text{ooFDObj})::\text{change}
\]

**SYNTAX**

\[
\text{ooStatus} \text{ ooRefHandle}(\text{ooFDObj})::\text{change}(\text{const char* bootFilePath} = 0,\text{ const char* lockServer} = 0,\text{ const uint32 fdNumber} = 0,\text{ FILE* fp} = \text{stdout}) \text{ const};
\]

where

- **bootFilePath**
  - Path to the boot file of the federated database.
  - The default value is zero.

- **lockServer**
  - Name of the new host running the lock server process for this federated database. The default value is zero.

- **fdNumber**
  - New identifier of the federated database

- **fp**
  - Pointer to the transcript file to which the system name of the federated database, the lock server host name, and the identifier of the federated database are echoed before and after the change member function is invoked. The default value is stdout.
DESCRIPTION

(for administration) Change federated database attributes. If any of the first three arguments has a value other than zero, then the catalog information corresponding to that argument is set to the specified value. Catalog information corresponding to arguments with zero values is not modified. After calling change, the old boot file remains. You must delete this file manually.

When using the `ooRefHandle(ooFDObj)::change` member function to modify the attributes of a federated database, you must strictly follow these steps:

1. Shut down the lock server.
2. Run in single-user mode (call `ooNoLock` in your program).
3. Start a transaction.
4. Open the federated database for update.
5. Invoke the `ooRefHandle(ooFDObj)::change` member function.
6. Commit the transaction.
7. Exit from the process.
8. Restart the lock server.

The federated database attributes are in a new and different state after the `ooRefHandle(ooFDObj)::change` member function has been run. For this reason, it is mandatory to exit from the process immediately after changing attributes.

RETURN VALUES

Returns a non-zero value for `ooStatus` if successful.

EXAMPLES

Assume the following handle declaration:

```cpp
ooHandle(ooFDObj) fdH;
...
// fdH is a valid handle to a federated database
```

The following examples show how to use the `ooHandle(ooFDObj)::change` member function to update the federated database attributes.
◆ This code changes the federated database identifier to 101. The lock server host name is not changed.
◆ fdH.change(0, 0, 101);
◆ This code changes the lock server host name to myHost:
◆ fdH.change(0, "myHost";

The following code uses the ooHandle(ooFDObj)::change member function to change the host name that is running the lock server process to moon.

```
ooTrans trans;
ooHandle(ooFDObj) fdH;
...
ooInit();
ooNoLock();
trans.start();
fdH.open("Documentation", oocUpdate);
fdH.change(0, "moon"); // change the lock server
trans.commit();
exit(0);
```

---

**MEMBER FUNCTION**  
**ooRefHandle(ooFDObj)::close**

**SYNTAX**  
```
ooStatus ooRefHandle(ooFDObj)::close() const;
```

**DESCRIPTION**  
Closes the referenced federated database.

**RETURN VALUES**  
ooStatus (oocSuccess or oocError).

---

**MEMBER FUNCTION**  
**ooRefHandle(ooFDObj)::contains**

**SYNTAX**  
```
ooStatus ooRefHandle(ooFDObj)::contains( 
    ooItr(ooDBObj) &dbI, 
    const oMode openMode = oocNoOpen) const;
```

**DESCRIPTION**  
Initializes the iterator dbI to traverse all databases in the referenced federated database. Objects of the iteration are automatically opened in the mode specified by openMode. The default is oocNoOpen (do not open the federated database automatically).

**RETURN VALUES**  
ooStatus (oocSuccess or oocError).
MEMBER FUNCTION  
**ooRefHandle(ooFDObj)::convertObjects**

SYNTAX  
`ooStatus ooRefHandle(ooFDObj)::convertObjects()`

DESCRIPTION  
Convert all objects affected by schema evolution operations within a federated database to their current class representations. Schema information used for object conversion will be purged as a side-effect of this operation.

RETURN VALUES  
oocSuccess if successful, or oocError if the federated database is opened for read-only access.

MEMBER FUNCTION  
**ooRefHandle(ooFDObj)::decodeSchema**

SYNTAX  
`ooStatus ooRefHandle(ooFDObj)::decodeSchema(const char* password) const;`

where  
**password**  
Password required to decode the schema

DESCRIPTION  
(for administration) Decode the contents of the schema of a federated database. You must use this member function by itself. You must use this member function exclusively; that is, you cannot invoke any other member function on a transaction at the same time.

To invoke `ooRefHandle(ooFDObj)::decodeSchema`, you must do the following:
1. Start a transaction.
2. Open the federated database.
3. Invoke `decodeSchema` using the password created when the schema was encoded.
4. Immediately commit the transaction.
5. Exit the process.

RETURN VALUES  
Returns a non-zero value for `ooStatus` if successful.

SEE ALSO  
“ooRefHandle(ooFDObj)::encodeSchema”
MEMBER FUNCTION  

**ooRefHandle(ooFDObj)::dumpCatalog**

**SYNTAX**

```c
ooStatus ooRefHandle(ooFDObj)::dumpCatalog(
    FILE* outputFile = stdout,
    const ooFileNameFormat format = oocHostLocal,
    const ooBoolean printLabels = oocTrue) const;
```

**where**

- **outputFile**
  - Pointer to the output file that stores the federated database attribute information normally printed in the standard output by the `oodumpcatalog` tool. The default value is `stdout`.

- **format**
  - Flag indicating the format of the printed file name. The default value is `oocHostLocal`, which formats the file name as `host:localPath` (for example, `object:/mnt/ed/design/up.FDB`). The other allowable value is `oocNative`, which formats the file name as a full path name or just a local name (for example, `/net/object/usr/mnt/ed/dsgn/up.FDB`).

- **printLabels**
  - Specifies whether or not to label the file in the output. The default is `oocTrue`, which specifies that each file name in the output is labeled.

**DESCRIPTION**

(for administration) Print out federated database attributes (catalog information) from within an application. This performs the same function as `oodumpcatalog`.

**RETURN VALUES**

Returns a non-zero value for `ooStatus` if successful.

**EXAMPLE**

```c
ooHandle(ooFDObj) fdH;
FILE* fp;

fp = fopen("catalog.inf", "w");

// Print the attributes on the standard output
fdH.dumpCatalog();

// Print file names in host-local format and with labels
```
// in the file catalog.inf.
fdH.dumpCatalog(fp, oocHostLocal, oocTrue);

// Print the attributes in the standard output
// with file names in native format.
fdH.dumpCatalog(stdout, oocNative, oocFalse);

**SEE ALSO**
“Using oodumpcatalog to List all Files in a Federated Database” on page 3-5 in *Objectivity/DB Administration*.

---

**MEMBER FUNCTION**  
`ooRefHandle(ooFDObj)::encodeSchema`

**SYNTAX**  

```c
ooStatus ooRefHandle(ooFDObj)::encodeSchema(
    const char* password) const;
```

**DESCRIPTION**

(for administration) Encodes the contents of the schema of a federated database. Use this member function alone. You must use this member function exclusively; that is, you cannot invoke any other member function on a transaction at the same time.

To invoke `ooRefHandle(ooFDObj)::encodeSchema`,

1. Start a transaction.
2. Open the federated database.
3. Invoke `encodeSchema` using a password. The password secures and protects the schema so that it cannot be viewed through Objectivity/DB tools.
4. Immediately commit the transaction.
5. Exit the process.

**WARNING:** Be sure to record your password in a safe place. If you cannot remember the password when you wish to decode the schema, Objectivity cannot help you.

**RETURN VALUES**

Returns a non-zero value for `ooStatus` if successful.

**SEE ALSO**

“`ooRefHandle(ooFDObj)::decodeSchema`”
MEMBER FUNCTION  
\texttt{ooRefHandle(ooFDObj)::exist}

SYNTAX  
\texttt{ooBoolean \ ooRefHandle(ooFDObj)::exist(}
\texttt{const char \*bootFilePath,}
\texttt{const ooMode \ openMode \ = \ oocNoOpen);}

DESCRIPTION  
Tests for the existence of the federated database specified by boot file 
\textit{bootFilePath}. If the federated database exists and \texttt{openMode} is 
\texttt{oocRead} or \texttt{oocUpdate}, the federated database is opened and its object 
reference or handle is initialized. If \texttt{openMode} is not \texttt{oocNoOpen}, and 
x\texttt{exist} returns \texttt{oocFalse}, the object either does not exist or the object 
exists but is not accessible.

DESCRIPTION  
Tests for the existence of a database. Returns the constant \texttt{oocTrue} if 
the federated database specified by boot file \textit{bootFilePath} exists, 
otherwise returns \texttt{oocFalse}. If the federated database exists and 
\texttt{openMode} is \texttt{oocRead} or \texttt{oocUpdate}, the federated database is opened 
and its object reference or handle is initialized to the federated 
database. If \texttt{openMode} is \texttt{oocRead} or \texttt{oocUpdate}, and \texttt{exist} returns 
\texttt{oocFalse}, the federated database either does not exist or the object 
exists but is not accessible, and the object reference or handle is set to 
null. If \texttt{openMode} is \texttt{oocNoOpen} and the federated database exists, the 
federated database is not opened, but the federated database reference 
or handle is initialized to the federated database. If \texttt{openMode} is 
\texttt{oocNoOpen} and the federated database does not exist, the federated 
database is not opened, and the object reference or handle is set to null.

RETURN VALUES  
\texttt{oocTrue} if the federated database exists.

MEMBER FUNCTION  
\texttt{ooRefHandle(ooFDObj)::isValid}

SYNTAX  
\texttt{ooBoolean \ ooRefHandle(ooFDObj)::isValid() const;}

DESCRIPTION  
Checks the validity of an object reference or handle.

RETURN VALUES  
\texttt{oocTrue} if the object reference or handle is valid.
### MEMBER FUNCTION ooRefHandle(ooFDObj)::lock

**SYNTAX**

```cpp
ooStatus ooRefHandle(ooFDObj)::lock(
    const ooLockMode lockMode) const;
```

**DESCRIPTION**

Explicitly locks the federated database and propagates the lock to any objects associated through links that have lock propagation enabled.

**RETURN VALUES**

`ooStatus` (oocSuccess or oocError).

---

### MEMBER FUNCTION ooRefHandle(ooFDObj)::lockServerName

**SYNTAX**

```cpp
char* ooRefHandle(ooFDObj)::lockServerName() const;
```

**DESCRIPTION**

(for administration) Retrieves the name of the host running the lock server process for the federated database.

**RETURN VALUES**

Returns a pointer to a string of `char`.

---

### MEMBER FUNCTION ooRefHandle(ooFDObj)::name

**SYNTAX**

```cpp
char *ooRefHandle(ooFDObj)::name() const;
```

**DESCRIPTION**

Returns the system name of the referenced federated database. The string is statically allocated by the member function and is overwritten with each invocation (you may wish to copy the string to locally allocated storage rather than storing the pointer).

**RETURN VALUES**

Name of the federated database.

---

### MEMBER FUNCTION ooRefHandle(ooFDObj)::number

**SYNTAX**

```cpp
uint32 ooRefHandle(ooFDObj)::number() const;
```

**DESCRIPTION**

(for administration) Retrieves the identifier of the federated database.
MEMBER FUNCTION  ooRefHandle (ooFDObj) :: open

SYNTAX  

    ooStatus ooRefHandle (ooFDObj) :: open(
       const char* bootFilePath,
       const ooMode openMode = oocRead,
       ooBoolean recover = oocFalse);

where

- **bootFilePath**  Path to the boot file of the federated database. The default value is zero. You can specify this path with or without a host name. If you specify it as a host path, use the format host::path.
- **openMode**  Specifies the open mode for the federated database. Valid values for openMode are the constants oocRead and oocUpdate. The default open mode is oocRead.
- **recover**  Specifies whether automatic recovery on the local host should be performed on application startup.

DESCRIPTION  Initializes object reference or handle to reference the federated database specified by bootFilePath. The federated database is opened in the mode specified by openMode.

Any number of processes may open the same federated database for either read or update access as long as no other process has obtained an explicit lock on it using the lock member function. If a process has obtained an explicit read lock, any other process may open the federated database for read access. If a process has obtained an explicit update lock, no other process may open it.

If you intend to alter the contents (any objects it contains) of a federated database, you must open it for update access, or any changes made will be lost.

RETURN VALUES  ooStatus (oocSuccess or oocError).
MEMBER FUNCTION: `ooRefHandle (ooFDObj)::openMode`

SYNTAX: `ooMode ooRefHandle (ooFDObj)::openMode () const;`

DESCRIPTION: Returns the mode in which the currently referenced federated database has been opened.

RETURN VALUES:
- If the federated database is not open through this object reference or handle, returns `oocNoOpen`.
- If the federated database is open through this object reference or handle: if it has been opened for update using any handle, returns `oocUpdate`; otherwise, returns `oocRead`.

MEMBER FUNCTION: `ooRefHandle (ooFDObj)::pageSize`

SYNTAX: `uint32 ooRefHandle (ooFDObj)::pageSize () const;`

DESCRIPTION: (for administration) Obtains the page size of the federated database.

MEMBER FUNCTION: `ooRefHandle (ooFDObj)::setConversion`

SYNTAX:
```
ooConversionFunction ooRefHandle (ooFDObj)::setConversion(
    const char * className,
    ooConversionFunction convFunction);
```

where:
- `className`: Class of objects to be converted by the conversion function
- `convFunction`: Name of the conversion function

DESCRIPTION: Register a conversion function for the objects classes changed by schema evolution. You can have no more than one conversion function for each changed persistent-capable class. Registering a second conversion function for a class replaces the previously registered function and returns a pointer to it.

RETURN VALUES: Returns a pointer to the previously registered function and null otherwise.
MEMBER FUNCTION  ooRefHandle(ooFDObj)::tidy

SYNTAX  

ooStatus ooRefHandle(ooFDObj)::tidy(
    FILE *fp = stdout,
    const char* hostName = 0,
    const char* pathName = 0,
    const char* localDirName = 0) const;

where

- fp  Pointer to the file in which to store the audit report generated by running the member function
- hostName  Specifies the name of the host machine on which to create the scratch file.
- pathName  Specifies the path for the scratch file.
- localDirName  Specifies the name of the directory in which to create the scratch file.

DESCRIPTION  Tidies all of the databases in a federated database.

When Objectivity/DB is tidying a federated database, it needs a temporary scratch file to hold the intermediate data. By default, this scratch file is created in the directory that contains the federated database file. The hostName and localDirName arguments allow you to control where the scratch file is created. If either hostName or localDirName is zero or empty ("") or empty, the scratch file is created in the default directory.

When using this member function, you must follow these steps:

1. Start a transaction.
2. Open the federated database for update.
3. Invoke the ooRefHandle(ooFDObj)::tidy member function and immediately commit the transaction.

You cannot manipulate any database, container, or basic object within the transaction before invoking the ooRefHandle(ooFDObj)::tidy member function. This member function performs physical storage compaction and relocation. After calling this function, some of the cached system data is in an inconsistent state. It is important that you immediately commit the transaction so that the system data can be reinitialized to a known state.
You cannot abort a transaction after calling the 
`ooRefHandle(ooFDObj)::tidy` member function.

**RETURN VALUES**
Returns a non-zero value for `ooStatus` if successful.

**EXAMPLE**
The following code starts a transaction, opens a federated database for update, invokes the `tidy` member function, and then commits the transaction:

```cpp
ooTrans trans;
ooHandle(ooFDObj) fdH;
ooStatus status;
...
trans.start();
status = fdH.open("Documentation", oocUpdate);
if (status == oocSuccess) {
    status = fdH.tidy();
    if (status == oocSuccess) {
        trans.commit();
    }
}
```

---

**MEMBER FUNCTION**
`ooRefHandle(ooFDObj)::typeN`

**SYNTAX**
`ooTypeNumber ooRefHandle(ooFDObj)::typeN() const;`

**DESCRIPTION**
Return the type number of `ooFDObj`.

**RETURN VALUES**
Type number of the federated database.

---

**MEMBER FUNCTION**
`ooRefHandle(ooFDObj)::typeName`

**SYNTAX**
`char *ooRefHandle(ooFDObj)::typeName() const;`

**DESCRIPTION**
Returns the string `ooFDObj`. The string must be treated as read only.

**RETURN VALUES**
Class name of federated database currently referenced.
MEMBER FUNCTION  ooRefHandle(ooFDObj)::update
SYNTAX         ooStatus ooRefHandle(ooFDObj)::update() const;
DESCRIPTION  Opens the referenced federated database in update access mode.
RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION  ooRefHandle(ooFDObj)::upgradeObjects
SYNTAX         ooStatus ooRefHandle(ooFDObj)::upgradeObjects();
DESCRIPTION  Convert objects using stored schema evolution information and user-defined functions (if any).
After object conversion is complete, the evolved classes are marked so that applications created using the evolved schema can access the federated database. Also, unneeded stored schema evolution information is deleted upon successful completion.
RETURN VALUES  oocSuccess if successful, or oocError if no classes have been changed by schema evolution.
**CLASS NAME**

**d_Database**

**DESCRIPTION**

(ODMG only) Class *d_Database* is equivalent to class *ooRefHandle (ooFDObj)*. Note that an ODMG database is equivalent to an Objectivity/DB federated database.

When you use a *d_Database* object as a clustering directive when creating a persistent object, the new persistent object is placed in the default container of an Objectivity/DB (not ODMG) database. The database used is chosen in the following order:

1. The most recently opened database, if one has been opened.
2. The database named by the environment variable *OO_DB_NAME*, if that variable has been set. Otherwise, the database named *default_odmg_db*.

**TYPE NAME**

*d_Database::access_status*

**SYNTAX**

```cpp
enum d_Database::access_status {
    not_open,
    read_write,
    read_only,
    exclusive;
};
```

**DESCRIPTION**

(ODMG only) An ODMG equivalent to the Objectivity/DB enumerated type *ooMode*.

These values map as follows:

<table>
<thead>
<tr>
<th>ODMG Value</th>
<th>Objectivity/DB value</th>
</tr>
</thead>
<tbody>
<tr>
<td>not_open</td>
<td>oocNoOpen</td>
</tr>
<tr>
<td>read_write</td>
<td>oocUpdate</td>
</tr>
<tr>
<td>read_only</td>
<td>oocRead</td>
</tr>
<tr>
<td>exclusive</td>
<td>oocXUpdate</td>
</tr>
</tbody>
</table>

**MEMBER FUNCTION**

*d_Database::close*

**SYNTAX**

```cpp
void d_Database::close();
```

**DESCRIPTION**

(ODMG only) Close an ODMG database (Objectivity/DB federated database).
MEMBER FUNCTION  
\texttt{d\_Database::get\_object\_name}

SYNTAX  
\texttt{const char *d\_Database::get\_object\_name (const d\_Ref\_Any &) const;}

DESCRIPTION  
(ODMG only) Get the name of an object.

MEMBER FUNCTION  
\texttt{d\_Database::lookup\_object}

SYNTAX  
\texttt{d\_Ref\_Any d\_Database::lookup\_object (const char * name) const;}

DESCRIPTION  
(ODMG only) Lookup an object by name.

MEMBER FUNCTION  
\texttt{d\_Database::open}

SYNTAX  
\texttt{void d\_Database::open (const char * databaseName , access\_status status = read\_write);}  

DESCRIPTION  
(ODMG only) Open an ODMG database (Objectivity/DB federated database). \texttt{databaseName} is the federated database boot file path.

MEMBER FUNCTION  
\texttt{d\_Database::rename\_object}

SYNTAX  
\texttt{void d\_Database::rename\_object (const char *oldName , const char *newName);}  

DESCRIPTION  
(ODMG only) Rename an object.

MEMBER FUNCTION  
\texttt{d\_Database::set\_object\_name}

SYNTAX  
\texttt{void d\_Database::set\_object\_name (const d\_Ref\_Any &theObject , const char *theName);}  

DESCRIPTION  
(ODMG only) Set the name of an object.
**d_Database**

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>d_Database::<code>transient_memory</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td><code>static const d_Database *const d_Database::transient_memory;</code></td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Creates objects of transient lifetime if used as database argument to operator <code>new</code>.</td>
</tr>
</tbody>
</table>
CLASS NAME     d_Ref_Any
DESCRIPTION    (ODMG only) Class d_Ref_Any provides a reference to any type. This class contains operators and member functions for d_Ref_Any objects.

CONSTRUCTOR    d_Ref_Any::d_Ref_Any
SYNTAX         d_Ref_Any::d_Ref_Any();
DESCRIPTION    (ODMG only) Default constructor for d_Ref_Any class.

CONSTRUCTOR    d_Ref_Any::d_Ref_Any
SYNTAX         d_Ref_Any::d_Ref_Any ( const d_Ref_Any &);
DESCRIPTION    (ODMG only) Copy constructor for d_Ref_Any class.

CONSTRUCTOR    d_Ref_Any::d_Ref_Any
SYNTAX         d_Ref_Any::d_Ref_Any ( d_Persistent_Object *);
DESCRIPTION    (ODMG only) Constructor for d_Ref_Any from an object pointer.

OPERATOR       d_Ref_Any::operator =
SYNTAX         d_Ref_Any &d_Ref_Any::operator = ( const d_Ref_Any &);
DESCRIPTION    (ODMG only) Default assignment operator.
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>d_Ref_Any::operator =</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>d_Ref_Any &amp;d_Ref_Any::operator = (</td>
</tr>
<tr>
<td></td>
<td>d_Persistent_Object *);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Assignment operator to assign an object pointer to a d_Ref_Any.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator ==</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator == (</td>
</tr>
<tr>
<td></td>
<td>const d_Ref_Any &amp;,</td>
</tr>
<tr>
<td></td>
<td>const d_Ref_Any &amp;);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Equality operator to compare d_Ref_Any.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator ==</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator == (</td>
</tr>
<tr>
<td></td>
<td>const d_Ref_Any &amp;,</td>
</tr>
<tr>
<td></td>
<td>const d_Persistent_Object *);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Equality operator to compare a d_Ref_Any and an object pointer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator ==</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int d_Ref_Any::operator == (</td>
</tr>
<tr>
<td></td>
<td>const d_Persistent_Object *,</td>
</tr>
<tr>
<td></td>
<td>const d_Ref_Any &amp;);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Equality operator to compare an object pointer and a d_Ref_Any.</td>
</tr>
<tr>
<td>OPERATOR</td>
<td>::operator ==</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator == (</td>
</tr>
<tr>
<td></td>
<td>const ooRefHandle(ooObj) &amp; left,</td>
</tr>
<tr>
<td></td>
<td>const d_Ref_Any &amp; right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Equality operator to compare an Objectivity/DB reference or handle and a d_Ref_Any.</td>
</tr>
</tbody>
</table>

| OPERATOR   | ::operator ==               |
| SYNTAX     | friend int ::operator == ( |
|            | const d_Ref_Any & left, |
|            | const ooRefHandle(ooObj) & right); |
| DESCRIPTION| (ODMG only) Equality operator to compare a d_Ref_Any and an Objectivity/DB reference or handle. |

| OPERATOR   | ::operator ==               |
| SYNTAX     | friend int ::operator == ( |
|            | const ooHandle(ooObj) & left, |
|            | const d_Ref_Any & right); |
| DESCRIPTION| (ODMG only) Equality operator to compare an Objectivity/DB reference or handle and a d_Ref_Any. |

<p>| OPERATOR   | ::operator !=               |
| SYNTAX     | friend int ::operator != ( |
|            | const d_Ref_Any &amp; |
|            | const d_Ref_Any &amp;); |
| DESCRIPTION| (ODMG only) Inequality operator to compare d_Ref_Any s. |</p>
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator !=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator != (</td>
</tr>
<tr>
<td></td>
<td>const d_Ref_Any &amp;,</td>
</tr>
<tr>
<td></td>
<td>const d_Persistent_Object *);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Inequality operator to compare a d_Ref_Any and object pointer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator !=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator != (</td>
</tr>
<tr>
<td></td>
<td>const d_Persistent_Object *,</td>
</tr>
<tr>
<td></td>
<td>const d_Ref_Any &amp;);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Inequality operator to compare an object pointer and a d_Ref_Any.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator !=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator != (</td>
</tr>
<tr>
<td></td>
<td>const ooRefHandle(ooObj) &amp;left,</td>
</tr>
<tr>
<td></td>
<td>const d_Ref_Any &amp;right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Inequality operator to compare an Objectivity/DB reference or handle and a d_Ref_Any.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator !=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator != (</td>
</tr>
<tr>
<td></td>
<td>const d_Ref_Any &amp;left,</td>
</tr>
<tr>
<td></td>
<td>const ooRefHandle(ooObj) &amp;right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Inequality operator to compare a d_Ref_Any and an Objectivity/DB reference or handle.</td>
</tr>
</tbody>
</table>
MEMBER FUNCTION  


d_Ref_Any::clear

SYNTAX  
void d_Ref_Any::clear();

DESCRIPTION  
(ODMG only) Sets this d_Ref_Any to null.

MEMBER FUNCTION  

d_Ref_Any::delete_object

SYNTAX  
void d_Ref_Any::delete_object();

DESCRIPTION  
(ODMG only) Deletes the object referenced by d_Ref_Any.

MEMBER FUNCTION  

d_Ref_Any::is_null

SYNTAX  
int d_Ref_Any::is_null() const;

DESCRIPTION  
Checks whether or not this d_Ref_Any is null.

RETURN VALUES  
(ODMG only) Returns true if this d_Ref_Any is null and false if it is not null.
d_Ref_Any
Operators on User-Defined Classes

This appendix lists operators for user-defined classes. Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle( userClass ) &` are interchangeable with `const ooRef( userClass ) &`.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>userClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>User-defined class.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>userClass::operator new</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td><code>void *userClass::operator new(</code> \n<code>    size_t,</code> \n<code>    const char *);</code></td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Allocate a transient object of type <code>userClass</code>. Note that <code>type</code> is ignored.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>userClass::operator new</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td><code>void *userClass::operator new(</code> \n<code>    size_t,</code> \n<code>    ooRefHandle(ooObj) near,</code> \n<code>    const char *type);</code></td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) If <code>near</code> is null, allocate a transient object of type <code>userClass</code>. Otherwise, allocate a persistent object of type <code>userClass</code> near <code>near</code>. Note that <code>type</code> is ignored.</td>
</tr>
</tbody>
</table>
OPERATOR  userClass::operator new
SYNTAX    void *userClass::operator new(
            size_t,
            d_Database *db,
            const char *type = 0);
DESCRIPTION (ODMG only) If db is null, allocate a transient object of type
userClass. Otherwise, allocate a persistent object of type userClass
in db. Note that type is ignored.
User-Defined Operators

This appendix provides the syntax of the ooOperatorSet class used for user-defined relational operators.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form const ooHandle(userClass) & are interchangeable with const ooRef(userClass) &.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooOperatorSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Class for user-defined relational operators.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>ooOperatorSet::ooOperatorSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooOperatorSet::ooOperatorSet();</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Default constructor to create a set of user-defined relational operators.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooOperatorSet::registerOperator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus ooOperatorSet::registerOperator( const char *name, ooQueryOperatorPtr funcPtr);</td>
</tr>
<tr>
<td>where</td>
<td>name is the operator name. This name should not begin or end with the following symbols or symbol combinations: )(&amp; &amp;</td>
</tr>
</tbody>
</table>
If the name is the same as an existing operator, the new operator will override the standard behavior.

*funcPtr* is a pointer to the function that defines the operator’s behavior.

**DESCRIPTION**
Used to register a user-defined relational operator.

**RETURN VALUES**
`ooStatus` (`oocSuccess` or `oocError`).

---

**MEMBER FUNCTION**
`ooOperatorSet::clear`

**SYNTAX**
`ooOperatorSet::clear()`

**DESCRIPTION**
Used to clear all user-defined relational operators.
User-Defined Functions

This appendix provides the required syntax for user-defined relational operator and object conversion functions.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(userClass) &` are interchangeable with `const ooRef(userClass) &`.

<table>
<thead>
<tr>
<th>FUNCTION NAME</th>
<th>userRelOperFunction</th>
</tr>
</thead>
</table>
| SYNTAX                | `ooBoolean userRelOperFunction(
|                       |   const void * lPtr,
|                       |   const void * rPtr,
|                       |   ooDataType lAType,
|                       |   ooDataType rAType);`     |
| where                 |                              |
| `lPtr`                | Pointer to the left operand of the operator |
| `rPtr`                | Pointer to the right operand of the operator |
| `lAType`, `rAType`     | Data types for the left and right operands, respectively. Every operator function is expected to handle arguments of whatever type the query yields. |
| DESCRIPTION           | Signature for user-defined relational operator functions. |
| RETURN VALUES         | `oocTrue` if the expression satisfied the condition of the operator, or `oocFalse` if the expression did not satisfy the condition. |
FUNCTION NAME: **userConversionFunction**

SYNTAX:
```c
typedef void userConversionFunction(
    const ooConvertInObject & existObj,
    ooConvertInOutObj & convObj);

where
existObj    Existing object before schema evolution
convObj    Converted object after schema evolution
```

DESCRIPTION:
Signature for user-defined object conversion functions. Use `ooRefHandle(ooFDObj)::setConversion` to register this function to be invoked automatically on each converted object of a changed class. You can have no more than one conversion function for each changed persistent-capable class, and the function must not access any other persistent object.
User-Defined Iterator Classes

This appendix describes the constructor and member functions for the iterator class that is declared for each user-defined class. Iterators allow you to traverse collections of objects.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form `const ooHandle(userClass) &` are interchangeable with `const ooRef(userClass) &`.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooItr(userClass)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Iterator class that corresponds to a user-defined persistent class.</td>
</tr>
<tr>
<td>CLASS HIERARCHY</td>
<td>ooHandle(userClass) -&gt; ooItr(userClass)</td>
</tr>
</tbody>
</table>
| INHERITED MEMBERS | ooHandle(userClass)::operator=  
                           ooHandle(userClass)::operator->  
                           ooHandle(userClass)::operator*  
                           ooHandle(userClass)::checkin  
                           ooHandle(userClass)::checkout  
                           ooHandle(userClass)::containedIn  
                           ooHandle(userClass)::getDefaultVers  
                           ooHandle(userClass)::getNameScope  
                           ooHandle(userClass)::getNextVers  
                           ooHandle(userClass)::getPrevVers  
                           ooHandle(userClass)::getObjName  
                           ooHandle(userClass)::getVersStatus  
                           ooHandle(userClass)::lock  
                           ooHandle(userClass)::lockProp  
                           ooHandle(userClass)::lookupObj |
ooItr(userClass)

**ooHandle(userClass)::nameObj**
- **ooHandle(userClass)::open**
- **ooHandle(userClass)::openMode**
- **ooHandle(userClass)::setDefaultVers**
- **ooHandle(userClass)::setVersStatus**
- **ooHandle(userClass)::typeName**
- **ooHandle(userClass)::unnameObj**
- **ooHandle(userClass)::update**

**DESTRUCTOR**
- **ooItr(userClass)::ooItr(userClass)**

**SYNTAX**
- **ooItr(userClass)();**

**DESCRIPTION**
- Constructor for class ooItr(userClass).

**MEMBER FUNCTION**
- **ooItr(userClass)::end**

**SYNTAX**
- **ooStatus end();**

**DESCRIPTION**
- Explicitly closes an iterator when the iteration does not complete normally.

**MEMBER FUNCTION**
- **ooItr(userClass)::next**

**SYNTAX**
- **ooBoolean next();**

**DESCRIPTION**
- Sets iterator to reference the next object of the current iteration.

**RETURN VALUES**
- **oocTrue** if there is a next object within the iteration, **oocFalse** if all of the objects of the iterations have been traversed or if an error occurred.
MEMBER FUNCTION  ooItr(userClass)::scan

SYNTAX 1.  ooStatus scan(const ooHandle(userClass) & objH,
const ooMode openMode = oocNoOpen);

2.  ooStatus scan(const ooHandle(ooSystemObj) & scopeH,
const ooLookupKey & lookupKey,
const ooMode openMode = oocNoOpen);

3.  ooStatus ooItr(userClass)::scan(
  ooHandle(scopeClass) & scopeH,
  const char * predicate);

4.  ooStatus ooItr(userClass)::scan(
  ooHandle(scopeClass) & scopeH,
  const ooMode openMode,
  const ooAccessMode access,
  const char * predicate);

DESCRIPTION 1.  Initializes iterator to traverse all objects of class userClass and
its user-defined derived classes within the object referenced by objH.

2.  Initializes iterator to traverse only those objects of class userClass
and its user-defined derived classes within the object referenced by scopeH that satisfy the condition specified by lookupKey.

3.  Initializes an iterator to traverse the objects of class userClass
and its user-defined derived classes within the object referenced by scopeH that satisfy the condition specified by predicate.

4.  Initializes an iterator to traverse the objects of class userClass
and its user-defined derived classes within the object referenced by scopeH that satisfy the condition specified by predicate. Also specifies the mode in which to open the traversed objects and specifies the accessibility of the class members in the predicate.

scopeObject  Handle of the object to scan.
scopeClass  Class of the object to scan. scopeClass may be either ooContObj, ooDBObj, ooFDObj, or a subclass of ooContObj.
openMode  Optional parameter specifying the mode in which each object is opened when the next member function is used. Its value may be any
of the constants oocRead, oocUpdate, or oocNoOpen. If openMode is not specified, the default is oocNoOpen.

**access**

Specifies the accessibility of the class members in the predicate. Its value must be one of the constants oocPublic and oocAll. oocPublic is the default value.

If you do not specify access, or if you use a value of oocPublic, you can access only public class members in the predicate, thus preserving encapsulation.

If you use oocAll, you can access any class member in the predicate. Using oocAll decreases the encapsulation of your system. To preserve maximum encapsulation, use it only within member functions of the class you are querying.

**predicate**

Condition that each retrieved object must satisfy. This string must be a valid expression in the predicate language. The iterator returns only the objects that match the predicate.

**RETURN VALUES**

ooStatus (oocSuccess or oocError).
User-Defined Object Reference and Handle Classes

This appendix describes the constructors, operators, and member functions for the object reference (\texttt{ooRef}) and handle (\texttt{ooHandle}) classes for user-defined classes. A description of the short object reference (\texttt{ooShortRef}) class begins on page P-6.

Where the syntax and parameter information is identical for the object reference and handle classes, \texttt{ooRefHandle} is used to represent either \texttt{ooRef} or \texttt{ooHandle}.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form \texttt{const ooHandle(userClass) &} are interchangeable with \texttt{const ooRef(userClass) &}.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>\texttt{ooRef(userClass)}, (\texttt{d_Ref&lt;userClass&gt;})</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Object reference class for user-defined classes. \texttt{d_Ref&lt;userClass&gt;} is the ODMG equivalent to \texttt{ooRefHandle(userClass)}.</td>
</tr>
<tr>
<td>CLASS HIERARCHY</td>
<td>\texttt{ooRef(userClass)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>\texttt{ooHandle(userClass)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>Handle class for user-defined classes. \texttt{ooHandle} provides the same member functions as \texttt{ooRef}, as well as \texttt{operator *}, and \texttt{operator userClass *}.</td>
</tr>
<tr>
<td>CLASS HIERARCHY</td>
<td>\texttt{ooHandle(userClass)}</td>
</tr>
</tbody>
</table>
**CONSTRUCTOR**

```
ooRefHandle(userClass) ::
 ooRefHandle(userClass)
```

**SYNTAX**

```
ooRefHandle(userClass) :: ooRefHandle(userClass) ();
```

**DESCRIPTION**

Default constructor for class `ooRef(userClass)` and `ooHandle(userClass)`.

---

**CONSTRUCTOR**

```
ooRefHandle(userClass) ::
 ooRefHandle(userClass)
```

**SYNTAX**

```
ooRefHandle(userClass) :: ooRefHandle(userClass) (const ooHandle(userClass) & objH);
```

**DESCRIPTION**

Constructor for class `ooRef(userClass)` and `ooHandle(userClass)`.

---

**CONSTRUCTOR**

```
ooRefHandle(userClass) ::
 ooRefHandle(userClass)
```

**SYNTAX**

```
ooRefHandle(userClass) :: ooRefHandle(userClass) (const ooRef(userClass) & objR);
```

**DESCRIPTION**

Constructor for class `ooRef(userClass)` and `ooHandle(userClass)`.

---

**CONSTRUCTOR**

```
ooRefHandle(userClass) ::
 ooRefHandle(userClass)
```

**SYNTAX**

```
ooRefHandle(userClass) :: ooRefHandle(userClass) (const userClass * objP);
```

**DESCRIPTION**

Constructor for class `ooRef(userClass)` and `ooHandle(userClass)`.
CONSTRUCTOR

`ooRefHandle(userClass):: ooRefHandle(userClass)`

SYNTAX

`ooRefHandle(userClass):: ooRefHandle(userClass) ( const d_Ref_Any & from );`

DESCRIPTION (ODMG only) Copy constructor for class `ooRef(userClass)` and `ooHandle(userClass)`. Sets this `ooRefHandle(userClass)` to `from`.

RETURN VALUES Signals an error if `from` is non-null and references an object that is not of type `userClass`.

OPERATOR

`ooRefHandle(userClass):: operator ->`

SYNTAX

`userClass * ooRefHandle(userClass):: operator -> () const;`

DESCRIPTION Returns pointer to the object in virtual memory. Also opens the object for read access if it is not open.

RETURN VALUES Pointer to object in virtual memory.

OPERATOR

`ooHandle(userClass):: operator *`

SYNTAX

`const userClass & ooHandle(userClass):: operator * () const;`  

Returns reference to the object in virtual memory. Also opens the object for read access if it is not open.  

This operation is not available for ooRef(ooObj).

DESCRIPTION Reference to object in virtual memory.

OPERATOR

`ooRefHandle(userClass):: operator =`

SYNTAX

1. `ooRefHandle(userClass) & ooRefHandle(userClass):: operator = ( const ooHandle(userClass) & objH);`
2. `ooRefHandle(userClass)`
   
   `&ooRefHandle(userClass)::operator = (const ooRef(userClass) &objR);`

3. `ooRefHandle(userClass)`
   
   `&ooRefHandle(userClass)::operator = (const ooShortRef(userClass) &shortR);`

4. `ooRefHandle(userClass)`
   
   `&ooRefHandle(userClass)::operator = (const userClass *objP);`

**DESCRIPTION**
Sets object reference or handle to reference specified object.
The database and container identified by the `ooRef` or `ooHandle` are used as the scope of `shortObjR`.

**RETURN VALUES**
Reference to object reference or handle.

---

**OPERATOR**
`ooRefHandle(userClass)::operator d_Ref_Any`

**SYNTAX**
`ooRefHandle(userClass)::operator d_Ref_Any() const;

**DESCRIPTION**
(ODMG only) Return this `ooRefHandle(userClass)` as a `d_Ref_Any`.

---

**MEMBER FUNCTION**
`ooHandle(userClass)::operator userClass*`

**SYNTAX**
`operator ooHandle(userClass)::operator userClass*() const;

**DESCRIPTION**
Returns pointer to the object in virtual memory. Object is pinned in memory only for the lifetime of the handle.

Note: this operation is not available for `ooRef` because only handles can temporarily pin objects in memory. Use `operator int` on object references to determine whether an object reference is null or non-null.

**RETURN VALUES**
Pointer to object in virtual memory.
MEMBER FUNCTION ooRefHandle(userClass)::copy

SYNTAX
1. ooRef(userClass) &ooRefHandle(userClass)::copy (const ooHandle(ooObj) & nearH, ooRef(userClass) & newObjR) const;
2. ooHandle(userClass) &ooRefHandle(userClass)::copy (const ooHandle(ooObj) & nearH, ooHandle(userClass) & newObjH) const;
3. ooHandle(userClass) ooRefHandle(userClass)::copy (const ooHandle(ooObj) & nearH) const;

DESCRIPTION Creates a copy of an object, returning the handle, object reference, or value of the new object.

RETURN VALUES ooStatus (oocSuccess or oocError).

MEMBER FUNCTION ooRefHandle(userClass)::lookupObj

SYNTAX
1. ooStatus ooRefHandle(userClass)::lookupObj (const ooHandle(ooObj) & scopeHandle, ooKey & key, const ooMode openMode = oocRead);
2. ooStatus ooRefHandle(userClass)::lookupObj (const ooHandle(ooObj) & scopeH, const ooKey & key, const ooMode openMode = oocRead);
3. ooStatus ooRefHandle(userClass)::lookupObj (const ooKey & key, const ooMode openMode = oocRead);
4. ooStatus ooRefHandle(userClass)::lookupObj (const ooHandle(ooObj) & scopeH, const char *name, const ooMode openMode = oocRead);
5. ooStatus ooRefHandle(userClass)::lookupObj (const char *name, const ooMode openMode = oocRead);
### ooShortRef(userClass)

**DESCRIPTION**
Sets object reference or handle to reference the object matching the key structure `key` in the scope of the handle `scopeH`, and opens the object in mode `openMode` (default is `oocRead`).

**RETURN VALUES**
`ooStatus (oocSuccess or oocError)`.

**MEMBER FUNCTION**
`ooRefHandle (userClass) :: ptr`

**SYNTAX**
`ooObj * ooRefHandle (userClass) :: ptr ();`

**DESCRIPTION**
(ODMG only) Return a pointer to the object referenced by this `ooRefHandle (userClass)`. This member function pins the object in memory.

---

**CLASS NAME**
**ooShortRef (userClass)**

**DESCRIPTION**
Short object reference class. All other short object references are derived from this one.

**CLASS HIERARCHY**
`ooShortRef (userClass)`

---

**CONSTRUCTOR**
`ooShortRef (userClass) :: ooShortRef (userClass)`

**SYNTAX**
1. `ooShortRef (userClass) :: ooShortRef (userClass) ();`
2. `ooShortRef (userClass) :: ooShortRef (userClass) (const ooShortRef (userClass) & shortObjR);`
3. `ooShortRef (userClass) :: ooShortRef (userClass) (const ooRef (userClass) & objR);`
4. `ooShortRef (userClass) :: ooShortRef (userClass) (const ooHandle (userClass) & objH);`
5. `ooShortRef (userClass) :: ooShortRef (userClass) (const userClass * objP);`

**DESCRIPTION**
Constructor for class `ooShortRef (userClass)`.
OPERATOR ooShortRef(userClass)::operator =

SYNTAX shortRef(userClass)::ooShortRef(userClass)
   &operator=()
1. ooShortRef(userClass)
   &ooShortRef(userClass)::operator =(
      const ooShortRef(userClass) &shortObjR);
2. ooShortRef(userClass)
   &ooShortRef(userClass)::operator =(
      const ooRef(userClass) &objR);
3. ooShortRef(userClass)
   &ooShortRef(userClass)::operator =(
      const ooHandle(userClass) &objH);
4. ooShortRef(userClass)
   &ooShortRef(userClass)::operator =(
      const userClass *objP);

DESCRIPTION
1. Sets object reference to reference same object as shortObjR.
2. Sets short object reference to reference same object as objR.
3. Sets object reference to reference same object as objH.
4. Sets object reference to OID referred to by objP.
Association Member Functions for User Classes

This appendix lists the member functions that DDL generates automatically for the association links declared in your user-defined classes.

Parameters are shown as they are declared in the Objectivity/DB header files. Since Objectivity/DB provides for automatic conversions between object references and handles, parameters of the form \texttt{const ooHandle(userClass) \&} are interchangeable with \texttt{const ooRef(userClass) \&}.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>userClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>User-defined class.</td>
</tr>
</tbody>
</table>

**MEMBER FUNCTION** \( \texttt{userClass::add\_nameOfLink} \)

**SYNTAX**

\[
\text{ooStatus add\_nameOfLink(}
\text{ \texttt{const ooHandle(userClass) \&objH);}}
\]

**DESCRIPTION**

Adds an association to the object referenced by \texttt{objH} to the one-to-many or many-to-many association link \texttt{nameOfLink}.

**RETURN VALUES**

\texttt{ooStatus (oocSuccess or oocError)}.

**MEMBER FUNCTION** \( \texttt{userClass::del\_nameOfLink} \)

**SYNTAX**

\[
\text{ooStatus del\_nameOfLink();}
\]

**DESCRIPTION**

Deletes the associations on association link \texttt{nameOfLink}.

**RETURN VALUES**

\texttt{ooStatus (oocSuccess or oocError)}.
MEMBER FUNCTION  

userClass::exist_nameOfLink

SYNTAX  

ooBoolean exist_nameOfLink(  
    const ooHandle(userClass) objH) const;

DESCRIPTION  

Checks for the existence of an association between two objects on association link nameOfLink.

RETURN VALUES  

oocTrue if an association exists on the link, otherwise oocFalse.

MEMBER FUNCTION  

userClass::nameOfLink

SYNTAX  

ooHandle(userClass) nameOfLink(  
    const ooMode openMode = oocNoOpen) const;

DESCRIPTION  

Returns the handle (by value) of the object (of class userClass) associated via the one-to-one or many-to-one association link nameOfLink. A null handle is returned if no association exists on the link.

userClass  

Class of the linked objects

nameOfLink  

Name of the association link

openMode  

Optional parameter that specifies the mode in which each object is opened when the next member function is used. Its value may be any of the constants oocRead, oocUpdate, or oocNoOpen. If openMode is not specified, the default is oocNoOpen.

MEMBER FUNCTION  

userClass::nameOfLink

SYNTAX  

ooRef(userClass) &nameOfLink(  
    ooRef(userClass) &objR,  
    const ooMode openMode = oocNoOpen) const;

userClass  

Class of the linked objects

objR  

User-defined buffer

nameOfLink  

Name of the association link

openMode  

Optional parameter that specifies the mode in which each object is opened when the next member function is used. Its value may be any of the constants oocRead, oocUpdate, or oocNoOpen. If openMode is not specified, the default is oocNoOpen.
member function is used. Its value may be any of the constants oocRead, oocUpdate, or oocNoOpen. If openMode is not specified, the default is oocNoOpen.

DESCRIPTION

Returns the object reference (in the user-supplied buffer objR) of the object (of class userClass) associated via the one-to-one or many-to-one association link nameOfLink. A null object reference is returned if no association exists on the link.

MEMBER FUNCTION

userClass::nameOfLink

SYNTAX

ooHandle(userClass) &nameOfLink(
    ooHandle(userClass) &objH,
    const ooMode openMode = oocNoOpen) const;

userClass
Class of the linked objects

objH
User-defined buffer

nameOfLink
Name of the association link

openMode
Optional parameter that specifies the mode in which each object is opened when the next member function is used. Its value may be any of the constants oocRead, oocUpdate, or oocNoOpen. If openMode is not specified, the default is oocNoOpen.

DESCRIPTION

Returns the handle (in the user-supplied buffer objH) of the object (of class userClass) associated via the one-to-one or many-to-one association link nameOfLink. A null handle is returned if no association exists on the link.

MEMBER FUNCTION

userClass::nameOfLink

SYNTAX

ooStatus nameOfLink(
    ooItr(userClass) &objI,
    const ooMode openMode = oocNoOpen) const;

DESCRIPTION

Initializes an iterator to traverse the objects associated via the one-to-many or many-to-many link.
MEMBER FUNCTION  
userClass::nameOfLink

SYNTAX  
ooStatus userClass::nameOfLink(
    ooItr(userClass) &itr,
    const char *predicate);

userClass  Class of the linked objects
nameOfLink Name of the association link
itr  Name of the iterator to initialize
predicate Condition that each retrieved object must satisfy. This string must be a valid expression in the predicate language. The iterator returns only the objects that match the predicate.

DESCRIPTION  
Initializes an iterator to traverse the objects associated via the one-to-many or many-to-many link that satisfy the selection criteria specified by predicate.

MEMBER FUNCTION  
userClass::nameOfLink

SYNTAX  
ooStatus userClass::nameOfLink(
    ooItr(userClass) &itr,
    const ooMode openMode,
    const ooAccessMode access,
    char *predicate);

userClass  Class of the linked objects
nameOfLink Name of the association link
itr  Name of the iterator to initialize
openMode Optional parameter that specifies the mode in which each object is opened when the next member function is used. Its value may be any of the constants oocRead, oocUpdate, or oocNoOpen. If openMode is not specified, the default is oocNoOpen.
access Accessibility of the class members in the predicate. Its value must be one of the constants oocPublic and oocAll. The default value is oocPublic.
If you do not specify access, or if you use a value of oocPublic, you can access only public class members in the predicate, thus preserving encapsulation.

If you use oocAll, you can access any class member in the predicate. Using oocAll decreases the encapsulation of your system, so use it only within member functions of the class you are querying to preserve maximum encapsulation.

**predicate**
Condition that each retrieved object must satisfy. This string must be a valid expression in the predicate language. The iterator returns only the objects that match the predicate.

**DESCRIPTION**
Initializes an iterator to traverse the objects associated via the one-to-many or many-to-many link that satisfy the selection criteria specified by predicate. Also specifies the mode in which to open the traversed objects and specifies the accessibility of the class members in the predicate.

**RETURN VALUES**
ooStatus (oocSuccess or oocError).

**MEMBER FUNCTION**

```cpp
userClass::set_nameOfLink
```

**SYNTAX**

```cpp
ooStatus set_nameOfLink(const ooHandle(userClass) &objH);
```

**DESCRIPTION**
Sets the one-to-one or many-to-one association link `nameOfLink` with an association to the object referenced by `objH`.

**RETURN VALUES**
ooStatus (oocSuccess or oocError).
MEMBER FUNCTION  userClass::sub_nameOfLink

SYNTAX  
 ooStatus  sub_nameOfLink(
    const ooHandle(userClass) &);

DESCRIPTION  Removes the association on the one-to-many bidirectional link
nameOfLink to the object referenced by objH.

RETURN VALUES  ooStatus (oocSuccess or oocError).

SEE ALSO  ooItr(userClass)::next

MEMBER FUNCTION  userClass::sub_nameOfLink

SYNTAX  
 ooStatus  sub_nameOfLink(
    const ooHandle(userClass) & objH,
    const unit32 number = 1);

DESCRIPTION  Removes number of associations on the many-to-many link or
one-to-many unidirectional link nameOfLink to the object referenced
by objH.

RETURN VALUES  ooStatus (oocSuccess or oocError).

SEE ALSO  ooItr(userClass)::next
## Object Conversion Classes

This appendix shows the syntax of classes used for object conversion operations after schema evolution has been performed.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ooConvertInObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>The non-persistent-capable class provides member functions for you to get values of primitive data members. ooConvertInObject does not support variable-length data members (VArrays).</td>
</tr>
<tr>
<td>CLASS HIERARCHY</td>
<td>ooConvertInObject</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>ooConvertInObject::getInt8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>ooStatus ooConvertInObject::getInt8(</td>
</tr>
<tr>
<td></td>
<td>const char *memberName,</td>
</tr>
<tr>
<td></td>
<td>int8 &amp;value);</td>
</tr>
<tr>
<td></td>
<td>where</td>
</tr>
<tr>
<td></td>
<td>memberName</td>
</tr>
<tr>
<td></td>
<td>Name of data member</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Get the value of an int8 data member.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>oocSuccess if successful, or oocError if the data member name is not valid.</td>
</tr>
</tbody>
</table>
MEMBER FUNCTION  ooConvertInObject::getInt16
SYNTAX       ooStatus ooConvertInObject::getInt16(
            const char *memberName,  
            int16 &value);
where
    memberName    Name of data member
DESCRIPTION   Get the value of an int16 data member.
RETURN VALUES oocSuccess if successful, or oocError if the data member name is not valid.

MEMBER FUNCTION  ooConvertInObject::getInt32
SYNTAX       ooStatus ooConvertInObject::getInt32(
            const char *memberName,  
            int32 &value);
where
    memberName    Name of data member
DESCRIPTION   Get the value of an int32 data member.
RETURN VALUES oocSuccess if successful, or oocError if the data member name is not valid.

MEMBER FUNCTION  ooConvertInObject::getUint8
SYNTAX       ooStatus ooConvertInObject::getUint8(
            const char *memberName,  
            uint8 &value);
where
    memberName    Name of data member
DESCRIPTION   Get the value of a Uint8 data member.
RETURN VALUES oocSuccess if successful, or oocError if the data member name is not valid.
MEMBER FUNCTION ooConvertInObject::getUint16

SYNTAX ooStatus ooConvertInObject::getUint16(
    const char *memberName,
    uint16 &value);

where
memberName Name of data member

DESCRIPTION Get the value of a Uint16 data member.

RETURN VALUES oocSuccess if successful, or oocError if the data member name is not valid.

MEMBER FUNCTION ooConvertInObject::getUint32

SYNTAX ooStatus ooConvertInObject::getUint32(
    const char *memberName,
    uint32 &value);

where
memberName Name of data member

DESCRIPTION Get the value of a Uint32 data member.

RETURN VALUES oocSuccess if successful, or oocError if the data member name is not valid.

MEMBER FUNCTION ooConvertInObject::getFloat32

SYNTAX ooStatus ooConvertInObject::getFloat32(
    const char *memberName,
    float32 &value);

where
memberName Name of data member

DESCRIPTION Get the value of a float32 data member.

RETURN VALUES oocSuccess if successful, or oocError if the data member name is not valid.
### MEMBER FUNCTION ooConvertInObject::getFloat64

**SYNTAX**

```c
ooStatus ooConvertInObject::getFloat64(
    const char *memberName,
    float64 &value);
```

**where**

- `memberName` : Name of data member

**DESCRIPTION**

Get the value of a `float64` data member.

**RETURN VALUES**

- `oocSuccess` if successful, or `oocError` if the name is not valid.

### MEMBER FUNCTION ooConvertInObject::getOldBaseClass

**SYNTAX**

```c
ooStatus ooConvertInObject::getOldBaseClass(
    const char *baseClassPath,
    ooConvertInObject &existObj);
```

**where**

- `baseClassPath` : Path to the base class
- `existObj` : Existing object before schema evolution

**DESCRIPTION**

Get a reference to the existing object `existObj` that has the base class specified by `baseClassPath`.

**RETURN VALUES**

- `oocSuccess` if successful, or `oocError` if the path is not valid.

### MEMBER FUNCTION ooConvertInObject::getOldDataMember

**SYNTAX**

```c
ooStatus ooConvertInObject::getOldDataMember(
    const char *memberName,
    ooConvertInObject &existObj);
```

**where**

- `memberName` : Name of data member
- `existObj` : Existing object before schema evolution

**DESCRIPTION**

Get a reference to the existing object `existObj` that has the embedded object data member `memberName`.

**RETURN VALUES**

- `oocSuccess` if successful, or `oocError` if the r name is not valid.
CLASS NAME  ooConvertInOutObject

DESCRIPTION This non-persistent-capable class provides member functions for you to get and set data member values of primitive types. You can use these member functions to access an object's state after it is converted by object conversion operations. You can use these member functions to access variable-length data members (VArrays).

CLASS HIERARCHY  ooConvertInObject->ooConvertInOutObject

MEMBER FUNCTION  ooConvertInOutObject::getNewBaseClass

SYNTAX  

```c
ooStatus ooConvertInObject::getNewBaseClass(
    const char *baseClassPath,
    ooConvertInOutObject &convObj);
```

where

- `baseClassPath`  Path to the base class
- `convObj`  Converted object after schema evolution

DESCRIPTION  Get a reference to the converted object `convObj` that has the base class specified by `baseClassPath`.

RETURN VALUES  oocSuccess if successful, or oocError if the base class path is not valid.

MEMBER FUNCTION  ooConvertInOutObject::getNewDataMember

SYNTAX  

```c
ooStatus ooConvertInObject::getNewDataMember(
    const char *memberName,
    ooConvertInOutObject &convObj);
```

where

- `memberName`  Name of data member
- `convObj`  Converted object after schema evolution

DESCRIPTION  Get a reference to the converted object `convObj` that has the embedded object data member `memberName`.

RETURN VALUES  oocSuccess if successful, or oocError if the data member name is not valid.
MEMBER FUNCTION ooConvertInOutObject::setInt8
SYNTAX ooStatus ooConvertInOutObject::setInt8(const char *memberName, const int8 value);
where
memberName Name of data member
DESCRIPTION Set the value of an int8 data member.
RETURN VALUES oocSuccess if successful, or oocError if the data member name is not valid.

MEMBER FUNCTION ooConvertInOutObject::setInt16
SYNTAX ooStatus ooConvertInOutObject::setInt16(const char *memberName, const int16 value);
where
memberName Name of data member
DESCRIPTION Set the value of an int16 data member.
RETURN VALUES oocSuccess if successful, or oocError if the data member name is not valid.

MEMBER FUNCTION ooConvertInOutObject::setInt32
SYNTAX ooStatus ooConvertInOutObject::setInt32(const char *memberName, const int32 value);
where
memberName Name of data member
DESCRIPTION Set the value of an int32 data member.
RETURN VALUES oocSuccess if successful, or oocError if the data member name is not valid.
**MEMBER FUNCTION**  
**ooConvertInOutObject**: : **setUint8**

**SYNTAX**

```
ooStatus ooConvertInOutObject::setUint8(
    const char *memberName,
    const uint8 value);
```

*where*

- `memberName`: Name of data member

**DESCRIPTION**
Set the value of a Uint8 data member.

**RETURN VALUES**
oocSuccess if successful, or oocError if the data member name is not valid.

---

**MEMBER FUNCTION**  
**ooConvertInOutObject**: : **setUint16**

**SYNTAX**

```
ooStatus ooConvertInOutObject::setUint16(
    const char *memberName,
    const uint16 value);
```

*where*

- `memberName`: Name of data member

**DESCRIPTION**
Set the value of a Uint16 data member.

**RETURN VALUES**
oocSuccess if successful, or oocError if the data member name is not valid.

---

**MEMBER FUNCTION**  
**ooConvertInOutObject**: : **setUint32**

**SYNTAX**

```
ooStatus ooConvertInOutObject::setUint32(
    const char *memberName,
    const uint32 value);
```

*where*

- `memberName`: Name of data member

**DESCRIPTION**
Set the value of a Uint32 data member.

**RETURN VALUES**
oocSuccess if successful, or oocError if the data member name is not valid.
MEMBER FUNCTION  ooConvertInOutObject::setFloat32

SYNTAX  ooStatus ooConvertInOutObject::setFloat32(
        const char *memberName,
        const float32 value);

where

memberName  Name of data member

DESCRIPTION  Set the value of a float32 data member.

RETURN VALUES  oocSuccess if successful, or oocError if the data member name is not valid.

MEMBER FUNCTION  ooConvertInOutObject::setFloat64

SYNTAX  ooStatus ooConvertInOutObject::setFloat64(
        const char *memberName,
        const float64 value);

where

memberName  Name of data member

DESCRIPTION  Set the value of a float64 data member.

RETURN VALUES  oocSuccess if successful, or oocError if the data member name is not valid.
**ODMG Date and Time Classes**

This appendix presents ODMG date and time classes—\texttt{d\_Date}, \texttt{d\_Time}, \texttt{d\_Timestamp}, and \texttt{d\_Interval}.

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>\texttt{d_Date}</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) ODMG date class.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>\texttt{d_Date::Month}</th>
</tr>
</thead>
</table>
| SYNTAX       | enum \texttt{d\_Date::Month} {  
  January = 1,  
  February = 2,  
  March = 3,  
  April = 4,  
  May = 5,  
  June = 6,  
  July = 7,  
  August = 8,  
  September = 9,  
  October = 10,  
  November = 11,  
  December = 12  
}; |
<p>| DESCRIPTION  | (ODMG only) Enumerated type for month of the year. |</p>
<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>d_Date::Weekday</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>enum d:Date::Weekday {</td>
</tr>
<tr>
<td></td>
<td>Sunday = 0,</td>
</tr>
<tr>
<td></td>
<td>Monday = 1,</td>
</tr>
<tr>
<td></td>
<td>Tuesday = 2,</td>
</tr>
<tr>
<td></td>
<td>Wednesday = 3,</td>
</tr>
<tr>
<td></td>
<td>Thursday = 4,</td>
</tr>
<tr>
<td></td>
<td>Friday = 5,</td>
</tr>
<tr>
<td></td>
<td>Saturday = 6 }</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Enumerated type for day of the week.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>d_Date::d_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>d_Date::d_Date();</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Default constructor for d_Date class.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>d_Date::d_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>d_Date::d_Date(</td>
</tr>
<tr>
<td></td>
<td>const d_Date &amp;);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Copy constructor for d_Date class.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>d_Date::d_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>d_Date::d_Date(</td>
</tr>
<tr>
<td></td>
<td>unsigned short year,</td>
</tr>
<tr>
<td></td>
<td>unsigned short dayOfYear);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Constructor for d_Date class.</td>
</tr>
</tbody>
</table>
CONSTRUCTOR

\textbf{d\_Date::d\_Date}

\textbf{SYNTAX}

\begin{verbatim}
d\_Date::d\_Date (  
  unsigned short year,  
  unsigned short month = 1,  
  unsigned short day = 1);  
\end{verbatim}

\textbf{DESCRIPTION}

(ODMG only) Constructor for \textit{d\_Date} class.

---

CONSTRUCTOR

\textbf{d\_Date::d\_Date}

\textbf{SYNTAX}

\begin{verbatim}
d\_Date::d\_Date (  
  const d\_Timestamp &);  
\end{verbatim}

\textbf{DESCRIPTION}

(ODMG only) Constructor for \textit{d\_Date} class.

---

OPERATOR

\textbf{d\_Date::operator =}

\textbf{SYNTAX}

\begin{verbatim}
d\_Date &d\_Date::operator = (  
  const d\_Timestamp &);  
\end{verbatim}

\textbf{DESCRIPTION}

(ODMG only) Assignment operator.

---

OPERATOR

\textbf{d\_Date::operator =}

\textbf{SYNTAX}

\begin{verbatim}
d\_Date &d\_Date::operator = (  
  const d\_Date &);  
\end{verbatim}

\textbf{DESCRIPTION}

(ODMG only) Assignment operator from time stamp.

---

OPERATOR

\textbf{::operator +}

\textbf{SYNTAX}

\begin{verbatim}
friend d\_Date ::operator +(  
  const d\_Date &left,  
  const d\_Interval &right);  
\end{verbatim}

\textbf{DESCRIPTION}

(ODMG only) Addition operator that adds a date and an interval of days.
### OPERATOR ::operator +

**SYNTAX**
```cpp
friend d_Date ::operator +(
    const d_Interval &left,
    const d_Date &right);
```

**DESCRIPTION**
(ODMG only) Addition operator that adds a date and an interval of days.

### OPERATOR ::operator -

**SYNTAX**
```cpp
friend d_Date ::operator -(d_Date &left,
    const d_Interval &right);
```

**DESCRIPTION**
(ODMG only) Subtraction operator that subtracts an interval of days, `d_Interval`, from a date `d_Date`.

### OPERATOR d_Date::operator ++

**SYNTAX**
```cpp
d_Date &d_Date::operator ++();
```

**DESCRIPTION**
(ODMG only) Prefix increment operator to increment date by days.

### OPERATOR d_Date::operator ++

**SYNTAX**
```cpp
d_Date d_Date::operator ++(int);
```

**DESCRIPTION**
(ODMG only) Postfix increment operator to increment date by days.

### OPERATOR d_Date::operator --

**SYNTAX**
```cpp
d_Date &d_Date::operator --();
```

**DESCRIPTION**
(ODMG only) Prefix decrement operator to decrement the date by days.
OPERATOR  
\texttt{d\_Date::operator --}  
SYNTAX  
\texttt{d\_Date d\_Date::operator --(int);}  
DESCRIPTION  
(ODMG only) Postfix decrement operator to decrement the date by days.

OPERATOR  
\texttt{d\_Date::operator +=}  
SYNTAX  
\texttt{d\_Date \&d\_Date::operator +=(const d\_Interval \&);}  
DESCRIPTION  
(ODMG only) Adds the interval to the date.

OPERATOR  
\texttt{d\_Date::operator +=}  
SYNTAX  
\texttt{d\_Date \&d\_Date::operator +=(int ndays);}  
DESCRIPTION  
(ODMG only) Adds to the date the number of days specified by \texttt{ndays}.

OPERATOR  
\texttt{d\_Date::operator -=}  
SYNTAX  
\texttt{d\_Date \&d\_Date::operator -=(const d\_Interval \&);}  
DESCRIPTION  
(ODMG only) Subtracts the interval from the date.

OPERATOR  
\texttt{d\_Date::operator -=}  
SYNTAX  
\texttt{d\_Date \&d\_Date::operator -=(int ndays);}  
DESCRIPTION  
(ODMG only) Subtracts from the date the number of days specified by \texttt{ndays}.  

ODMG Date and Time Classes  
S-5
OPERATOR ::operator ==
SYNTAX friend int ::operator == (const d_Date & left, const d_Date & right);
DESCRIPTION (ODMG only) Equality operator to compare d_Dates.

OPERATOR ::operator !=
SYNTAX friend int ::operator != (const d_Date & left, const d_Date & right);
DESCRIPTION (ODMG only) Inequality operator to compare d_Dates.

OPERATOR ::operator <
SYNTAX friend int ::operator < (const d_Date & left, const d_Date & right);
DESCRIPTION (ODMG only) Less than operator to compare d_Dates.

OPERATOR ::operator >
SYNTAX friend int ::operator > (const d_Date & left, const d_Date & right);
DESCRIPTION (ODMG only) Greater than operator to compare d_Dates.
**OPERATOR**: 
\[ ::\text{operator} \leq \]

**SYNTAX**
friend int ::\text{operator} \leq(
    const d\_Date &left,
    const d\_Date &right);

**DESCRIPTION**
(ODMG only) Less than or equal to operator to compare d\_Dates.

**OPERATOR**: 
\[ ::\text{operator} \geq \]

**SYNTAX**
friend int ::\text{operator} \geq(
    const d\_Date &left,
    const d\_Date &right);

**DESCRIPTION**
(ODMG only) Greater than or equal to operator to compare d\_Dates.

**MEMBER FUNCTION**: 
\[ d\_Date::\text{current} \]

**SYNTAX**
static d\_Date d\_Date::\text{current}();

**DESCRIPTION**
(ODMG only) Returns current date.

**MEMBER FUNCTION**: 
\[ d\_Date::\text{day} \]

**SYNTAX**
unsigned short d\_Date::\text{day}() const;

**DESCRIPTION**
(ODMG only) Returns this date's day of the month.

**MEMBER FUNCTION**: 
\[ d\_Date::\text{day\_of\_week} \]

**SYNTAX**
Weekday d\_Date::\text{day\_of\_week}() const;

**DESCRIPTION**
(ODMG only) Returns this date's day of the week.
MEMBER FUNCTION  
**d_Date::day_of_year**

SYNTAX  
unsigned short d_Date::day_of_year() const;

DESCRIPTION  
(ODMG only) Returns this date's day of the year.

OPERATOR  
**d_Date::days_in_month**

SYNTAX  
unsigned int d_Date::days_in_month() const;

DESCRIPTION  
(ODMG only) Returns the number of days in this date's month.

OPERATOR  
**d_Date::days_in_month**

SYNTAX  
static unsigned int d_Date::days_in_month(
    unsigned short year,
    unsigned short month);

DESCRIPTION  
(ODMG only) Returns the number of days in a month.

OPERATOR  
**d_Date::days_in_year**

SYNTAX  
unsigned int d_Date::days_in_year() const;

DESCRIPTION  
(ODMG only) Returns the number of days in this date's year.

OPERATOR  
**d_Date::days_in_year**

SYNTAX  
static unsigned int d_Date::days_in_year(
    unsigned short year);

DESCRIPTION  
(ODMG only) Returns the number of days in a year.
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>d_Date::is_between</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int d_Date::is_between(</td>
</tr>
<tr>
<td></td>
<td>const d_Date &amp;,</td>
</tr>
<tr>
<td></td>
<td>const d_Date &amp;) const;</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Returns true if this date is within the specified period.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>d_Date::is_leap_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>int d_Date::is_leap_year() const;</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Returns true if this date's year is a leap year.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>d_Date::is_leap_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>static int d_Date::is_leap_year(</td>
</tr>
<tr>
<td></td>
<td>unsigned short year);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Returns true if this date's year is a leap year.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>d_Date::is_valid_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>static int d_Date::is_valid_date(</td>
</tr>
<tr>
<td></td>
<td>unsigned short year,</td>
</tr>
<tr>
<td></td>
<td>unsigned short month,</td>
</tr>
<tr>
<td></td>
<td>unsigned short day);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Returns true if the date is a valid date.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEMBER FUNCTION</th>
<th>d_Date::month</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>Month short d_Date::month() const;</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Returns this date's month.</td>
</tr>
</tbody>
</table>
**MEMBER FUNCTION**

**d_Date::next**

**SYNTAX**

```cpp
d_Date   &d_Date::next (Weekday);
```

**DESCRIPTION** *(ODMG only)* Advances this date to the next occurrence of the specified day of type `Weekday`.

---

**MEMBER FUNCTION**

**d_Date::previous**

**SYNTAX**

```cpp
d_Date   &d_Date::previous (Weekday);
```

**DESCRIPTION** *(ODMG only)* Moves this date to the previous occurrence of the specified day of type `Weekday`.

---

**MEMBER FUNCTION**

**d_Date::year**

**SYNTAX**

```cpp
unsigned short d_Date::year() const;
```

**DESCRIPTION** *(ODMG only)* Returns this date's year.

---

**FUNCTION NAME**

`::overlaps`

**SYNTAX**

```cpp
friend int ::overlaps ( 
    const d_Date &startLeft, 
    const d_Date &endLeft, 
    const d_Date &startRight, 
    const d_Date &endRight);
```

**DESCRIPTION** *(ODMG only)* For `d_Date`, determines whether two specified time periods (`d_Date` and `d_Date`) overlap. The time periods are specified by a start and end time.
FUNCTION NAME ::overlaps
SYNTAX
friend int ::overlaps(
    const d_Timestamp &startLeft,
    const d_Timestamp &endLeft,
    const d_Date &startRight,
    const d_Date &endRight);

DESCRIPTION (ODMG only) For d_Date, determines whether two specified time periods (d_Timestamp and d_Date) overlap. The time periods are specified by a start and end time.

FUNCTION NAME ::overlaps
SYNTAX
friend int ::overlaps(
    const d_Date &startLeft,
    const d_Date &endLeft,
    const d_Timestamp &startRight,
    const d_Timestamp &endRight);

DESCRIPTION (ODMG only) For d_Date, determines whether two specified time periods (d_Date and d_Timestamp) overlap. The time periods are specified by a start and end time.

CLASS NAME d_Time
DESCRIPTION (ODMG only) The time class used to denote a specific time.

TYPE NAME Time_Zone
SYNTAX
enum Time_Zone {
    GMT = 0,
    GMT12 = 12,
    GMT_12 = -12,
    GMT1 = 1,
    GMT_1 = -1,
d_Time

GMT2 = 2,
GMT_2 = −2,
GMT3 = 3,
GMT_3 = −3,
GMT4 = 4,
GMT_4 = −4,
GMT5 = 5,
GMT_5 = −5,
GMT6 = 6,
GMT_6 = −6,
GMT7 = 7,
GMT_7 = −7,
GMT8 = 8,
GMT_8 = −8,
GMT9 = 9,
GMT_9 = −9,
GMT10 = 10,
GMT_10 = −10,
GMT11 = 11,
GMT_11 = −11,
USEastern = −5,
UScentral = −6,
USmountain = −7,
USpacific = −8
);

DESCRIPTION (ODMG only) Enumerated type used to denote a specific time zone. Time zones are numbered according to the number of hours that must be added or subtracted from local time to get the time in Greenwich, England (GMT). Read the underscore as a minus.

CONSTRUCTOR d_Time::d_Time

SYNTAX d_Time::d_Time();

DESCRIPTION (ODMG only) Default constructor for the d_Time class.
### CONSTRUCTOR
d\_Time::d\_Time

### SYNTAX
d\_Time::d\_Time (const d\_Time &);

### DESCRIPTION
(ODMG only) Copy constructor for the d\_Time class.

### CONSTRUCTOR
d\_Time::d\_Time

### SYNTAX
d\_Time::d\_Time (unsigned short hour, unsigned short minute, float second);

### DESCRIPTION
(ODMG only) Constructor for the d\_Time class.

### CONSTRUCTOR
d\_Time::d\_Time

### SYNTAX
d\_Time::d\_Time (unsigned short hour, unsigned short minute, float second, short tzhour, short tzminute);

### DESCRIPTION
(ODMG only) Constructor for the d\_Time class.

### CONSTRUCTOR
d\_Time::d\_Time

### SYNTAX
d\_Time::d\_Time (const d\_Timestamp &);

### DESCRIPTION
(ODMG only) Constructor for the d\_Time class.
### OPERATOR d_Time::operator =

**SYNTAX**
```
d_Time &d_Time::operator =(
    const d_Time &);
```

**DESCRIPTION**
(ODMG only) Assignment operator for `d_Time`.

### OPERATOR d_Time::operator =

**SYNTAX**
```
d_Time &d_Time::operator =(
    const d_Timestamp &);
```

**DESCRIPTION**
(ODMG only) Assignment operator to assign a `d_Timestamp` to a `d_Time`.

### OPERATOR 

**SYNTAX**
```
friend d_Time :: operator + (
    const d_Time & left,
    const d_Interval & right);
```

**DESCRIPTION**
(ODMG only) Addition operator to add `d_Time` and `d_Interval`.

### OPERATOR 

**SYNTAX**
```
friend d_Time :: operator + (
    const d_Interval & left,
    const d_Time & right);
```

**DESCRIPTION**
(ODMG only) Addition operator to add `d_Interval` and `d_Time`.

### OPERATOR 

**SYNTAX**
```
friend d_Interval :: operator - (const d_Time &left, const d_Time &right);
```

**DESCRIPTION**
(ODMG only) Subtraction operator that subtracts `d_Times`. 

---

**S-14 Using Objectivity/C++**
OPERATOR ::operator -
SYNTAX friend d_Time ::operator -(
    const d_Time &left,
    const d_Interval &right);
DESCRIPTION (ODMG only) Subtraction operator that subtracts d_Interval from d_Time.

OPERATOR d_Time::operator +=
SYNTAX d_Time &d_Time::operator +=(
    const d_Interval &);
DESCRIPTION (ODMG only) Adds the interval d_Interval to the time d_Time.

OPERATOR d_Time::operator -=
SYNTAX d_Time &d_Time::operator -=(
    const d_Interval &);
DESCRIPTION (ODMG only) Subtracts the interval d_Interval from the time d_Time.

OPERATOR ::operator ==
SYNTAX friend int ::operator ==(
    const d_Time &left,
    const d_Time &right);
DESCRIPTION (ODMG only) Equality operator for d_Time.
OPERATOR :: operator !=
SYNTAX friend int :: operator != (const d_Time & left, const d_Time & right);
DESCRIPTION (ODMG only) Inequality operator for d_Times.

OPERATOR :: operator <
SYNTAX friend int :: operator < (const d_Time & left, const d_Time & right);
DESCRIPTION (ODMG only) Less than operator for d_Times.

OPERATOR :: operator >
SYNTAX friend int :: operator > (const d_Time & left, const d_Time & right);
DESCRIPTION (ODMG only) Greater than operator for d_Times.

OPERATOR :: operator <=
SYNTAX friend int :: operator <=(const d_Time & left, const d_Time & right);
DESCRIPTION (ODMG only) Less than or equal to operator for d_Times.
### OPERATOR

**::operator >=**

**SYNTAX**

friend int ::operator >=(
    const d_Time &left,
    const d_Time &right);

**DESCRIPTION**

(ODMG only) Greater than or equal to operator for `d_Time`

### MEMBER FUNCTION

**d_Time::current**

**SYNTAX**

static d_Time d_Time::current();

**DESCRIPTION**

(ODMG only) Returns the current time.

**d_Time::hour**

**SYNTAX**

unsigned short d_Time::hour() const;

**DESCRIPTION**

(ODMG only) Returns this `d_Time`'s hour.

**d_Time::minute**

**SYNTAX**

unsigned short d_Time::minute() const;

**DESCRIPTION**

(ODMG only) Returns this `d_Time`'s minute.

**d_Time::second**

**SYNTAX**

float d_Time::second() const;

**DESCRIPTION**

(ODMG only) Returns this `d_Time`'s second.
### MEMBER FUNCTION
**d_Time::set_default_Time_Zone**

**SYNTAX**
```cpp
static void d_Time::set_default_Time_Zone(Time_Zone);
```

**DESCRIPTION**
(ODMG only) Sets the default time zone.

### MEMBER FUNCTION
**d_Time::set_default_Time_Zone_to_local**

**SYNTAX**
```cpp
static void d_Time::set_default_Time_Zone_to_local();
```

**DESCRIPTION**
(ODMG only) Sets the default time zone to the local time zone.

### MEMBER FUNCTION
**d_Time::tz_hour**

**SYNTAX**
```cpp
short d_Time::tz_hour() const;
```

**DESCRIPTION**
(ODMG only) Returns the current hour for a time zone.

### MEMBER FUNCTION
**d_Time::tz_minute**

**SYNTAX**
```cpp
short d_Time::tz_minute() const;
```

**DESCRIPTION**
(ODMG only) Returns the current minute for a time zone.
FUNCTION NAME ::overlaps
SYNTAX friend int ::overlaps(
    const d_Time &startLeft,
    const d_Time &endLeft,
    const d_Time &startRight,
    const d_Time &endRight);
DESCRIPTION (ODMG only) For d_Time, determines whether two specified time periods (d_Time and d_Time) overlap. The time periods are specified by a start and end time.

FUNCTION NAME ::overlaps
SYNTAX friend int ::overlaps(
    const d_Timestamp &startLeft,
    const d_Timestamp &endLeft,
    const d_Time &startRight,
    const d_Time &endRight);
DESCRIPTION (ODMG only) For d_Time, determines whether two specified time periods (d_Timestamp and d_Time) overlap. The time periods are specified by a start and end time.

FUNCTION NAME ::overlaps
SYNTAX friend int ::overlaps(
    const d_Time &startLeft,
    const d_Time &endLeft,
    const d_Timestamp &startRight,
    const d_Timestamp &endRight);
DESCRIPTION (ODMG only) For d_Time, determines whether two specified time periods (d_Time and d_Timestamp) overlap. The time periods are specified by a start and end time.
CLASS NAME  
**d_Timestamp**

DESCRIPTION  
The time stamp class, which consists of both date and time information.

### CONSTRUCTOR

**d_Timestamp::d_Timestamp**

**SYNTAX**

d_Timestamp::d_Timestamp();

**DESCRIPTION**  
(ODMG only) Default constructor for the d_Timestamp class.

### CONSTRUCTOR

**d_Timestamp::d_Timestamp**

**SYNTAX**

d_Timestamp::d_Timestamp(  
    const d_Date &);

**DESCRIPTION**  
(ODMG only) Copy constructor for the d_Timestamp class.

### CONSTRUCTOR

**d_Timestamp::d_Timestamp**

**SYNTAX**

d_Timestamp::d_Timestamp(  
    unsigned short year,  
    unsigned short month = 1,  
    unsigned short day = 1,  
    unsigned short hour = 0,  
    unsigned short minute = 0,  
    float second = 0.0);  

**DESCRIPTION**  
(ODMG only) Constructor for the d_Timestamp class.

### CONSTRUCTOR

**d_Timestamp::d_Timestamp**

**SYNTAX**

d_Timestamp::d_Timestamp(  
    const d_Date &, const d_Time &);  

**DESCRIPTION**  
(ODMG only) Constructor for the d_Timestamp class.
<table>
<thead>
<tr>
<th>CONSTRUCTOR</th>
<th>d_Timestamp::d_Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>d_Timestamp::d_Timestamp( const d_Timestamp &amp;);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Constructor for the d_Timestamp class.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>d_Timestamp::operator =</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>d_Timestamp &amp;d_Timestamp::operator = ( const d_Timestamp &amp;);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Assignment operator for d_Timestamp.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>d_Timestamp::operator =</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>d_Timestamp &amp;d_Timestamp::operator = ( const d_Date &amp;);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Assignment operator to assign a d_Date to a d_Timestamp.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator +</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend d_Timestamp ::operator +( const d_Timestamp &amp;left, const d_Interval &amp;right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Addition operator that adds d_Timestamp and d_Interval.</td>
</tr>
</tbody>
</table>
**OPERATOR** :: operator +
**SYNTAX**
friend d_Timestamp :: operator +(
    const d_Interval & left,
    const d_Timestamp & right);
**DESCRIPTION** (ODMG only) Addition operator that adds d_Interval and d_Timestamp.

**OPERATOR** :: operator -
**SYNTAX**
friend d_Timestamp :: operator -(const d_Timestamp & left,
    const d_Interval & right);
**DESCRIPTION** (ODMG only) Subtraction operator that subtracts d_Interval from d_Timestamp.

**OPERATOR** d_Timestamp:: operator +=
**SYNTAX**
d_Timestamp &d_Timestamp:: operator +=(
    const d_Interval &);
**DESCRIPTION** (ODMG only) Adds the interval d_Interval to d_Timestamp.

**OPERATOR** d_Timestamp:: operator -=
**SYNTAX**
d_Timestamp &d_Timestamp:: operator -=(
    const d_Interval &);
**DESCRIPTION** (ODMG only) Subtracts the interval d_Interval from d_Timestamp.
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator ==</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator ==(</td>
</tr>
<tr>
<td></td>
<td>const d_Timestamp &amp;left,</td>
</tr>
<tr>
<td></td>
<td>const d_Timestamp &amp;right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Equality operator for d_Timestamp.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator !=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator != (</td>
</tr>
<tr>
<td></td>
<td>const d_Timestamp &amp;left,</td>
</tr>
<tr>
<td></td>
<td>const d_Timestamp &amp;right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Inequality operator for d_Timestamp.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &lt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator &lt; (</td>
</tr>
<tr>
<td></td>
<td>const d_Timestamp &amp;left,</td>
</tr>
<tr>
<td></td>
<td>const d_Timestamp &amp;right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Less than operator for d_Timestamp.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator &gt; (</td>
</tr>
<tr>
<td></td>
<td>const d_Timestamp &amp;left,</td>
</tr>
<tr>
<td></td>
<td>const d_Timestamp &amp;right);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Greater than operator for d_Timestamp.</td>
</tr>
</tbody>
</table>

**ODMG Date and Time Classes**  
S-23
**OPERATOR**  :: operator <=
**SYNTAX**
friend int :: operator <=(  
  const d_Timestamp &left,  
  const d_Timestamp &right);  
**DESCRIPTION**  (ODMG only) Less than or equal to operator for d_Timestamps.

**OPERATOR**  :: operator >=
**SYNTAX**
friend int :: operator >=(  
  const d_Timestamp &left,  
  const d_Timestamp &right);  
**DESCRIPTION**  (ODMG only) Greater than or equal to operator for d_Timestamps.

**MEMBER FUNCTION**  d_Timestamp::current
**SYNTAX**
static d_Timestamp d_Timestamp::current();
**DESCRIPTION**  (ODMG only) Returns the current time stamp for the time zone.

**MEMBER FUNCTION**  d_Timestamp::date
**SYNTAX**
const d_Date &d_Timestamp::date() const;
**DESCRIPTION**  (ODMG only) Returns the time stamp date.

**MEMBER FUNCTION**  d_Timestamp::day
**SYNTAX**
unsigned short d_Timestamp::day() const;
**DESCRIPTION**  (ODMG only) Returns the time stamp day.
### MEMBER FUNCTION d_Timestamp::hour

**SYNTAX**

```c
unsigned short d_Timestamp::hour() const;
```

**DESCRIPTION**

(ODMG only) Returns the time stamp hour.

### MEMBER FUNCTION d_Timestamp::minute

**SYNTAX**

```c
unsigned short d_Timestamp::minute() const;
```

**DESCRIPTION**

(ODMG only) Returns the time stamp minute.

### MEMBER FUNCTION d_Timestamp::month

**SYNTAX**

```c
unsigned short d_Timestamp::month() const;
```

**DESCRIPTION**

(ODMG only) Returns the time stamp month.

### MEMBER FUNCTION d_Timestamp::second

**SYNTAX**

```c
float d_Timestamp::second() const;
```

**DESCRIPTION**

(ODMG only) Returns the time stamp second.

### MEMBER FUNCTION d_Timestamp::time

**SYNTAX**

```c
const d_Time &d_Timestamp::time() const;
```

**DESCRIPTION**

(ODMG only) Returns the time stamp time.

### MEMBER FUNCTION d_Timestamp::tz_hour

**SYNTAX**

```c
short d_Timestamp::tz_hour() const;
```

**DESCRIPTION**

(ODMG only) Returns the time zone hour for the time stamp.
MEMBER FUNCTION  
**d_Timestamp::tz_minute**

**SYNTAX**  
short d_Timestamp::tz_minute() const;

**DESCRIPTION**  
(ODMG only) Returns the time zone minute for the time stamp.

---

MEMBER FUNCTION  
**d_Timestamp::year**

**SYNTAX**  
unsigned short d_Timestamp::year() const;

**DESCRIPTION**  
(ODMG only) Returns the time stamp year.

---

FUNCTION NAME  
**::overlaps**

**SYNTAX**  
friend int ::overlaps(
    const d_Timestamp &startLeft,
    const d_Timestamp &endLeft,
    const d_Timestamp &startRight,
    const d_Timestamp &endRight);

**DESCRIPTION**  
(ODMG only) For d_Timestamp, determines whether two specified time periods (d_Timestamp and d_Timestamp) overlap. The time periods are specified by a start and end time.

---

FUNCTION NAME  
**::overlaps**

**SYNTAX**  
friend int ::overlaps(
    const d_Timestamp &startLeft,
    const d_Timestamp &endLeft,
    const d_Date &startRight,
    const d_Date &endRight);

**DESCRIPTION**  
(ODMG only) For d_Timestamp, determines whether two specified time periods (d_Timestamp and d_Date) overlap. The time periods are specified by a start and end time.
FUNCTION NAME : ::overlaps
SYNTAX friend int ::overlaps(
        const d_Date &startLeft,
        const d_Date &endLeft,
        const d_Timestamp &startRight,
        const d_Timestamp &endRight);
DESCRIPTION (ODMG only) For d_Timestamp, determines whether two specified
        time periods (d_Date and d_Timestamp) overlap. The time periods are
        specified by a start and end time.

FUNCTION NAME : ::overlaps
SYNTAX friend int ::overlaps(
        const d_Timestamp &startLeft,
        const d_Timestamp &endLeft,
        const d_Time &startRight,
        const d_Time &endRight);
DESCRIPTION (ODMG only) For d_Timestamp, determines whether two specified
        time periods (d_Timestamp and d_Time) overlap. The time periods are
        specified by a start and end time.

FUNCTION NAME : ::overlaps
SYNTAX friend int ::overlaps(
        const d_Time &startLeft,
        const d_Time &endLeft,
        const d_Timestamp &startRight,
        const d_Timestamp &endRight);
DESCRIPTION (ODMG only) For d_Timestamp, determines whether two specified
        time periods (d_Time and d_Timestamp) overlap. The time periods are
        specified by a start and end time.
### Class Name

**d_Interval**

**Description**
The interval class is used to represent a duration of time. This class conforms to the day-time interval defined in the SQL standard.

### Constructor

**d_Interval::d_Interval**

**Syntax**
```
d_Interval::d_Interval(
    int day=0,
    int hour=0,
    int minute=0,
    float second=0.0);
```

**Description**
(ODMG only) Constructor for the `d_Interval` class.

### Constructor

**d_Interval::d_Interval**

**Syntax**
```
d_Interval::d_Interval(
    const d_Interval &);
```

**Description**
(ODMG only) Copy constructor for the `d_Interval` class.

### Operator

**d_Interval::operator =**

**Syntax**
```
d_Interval &d_Interval::operator = (const d_Interval &);
```

**Description**
(ODMG only) Assignment operator for the `d_Interval` class.

### Operator

**::operator +**

**Syntax**
```
friend d_Interval ::operator + (const d_Interval &left, const d_Interval &right);
```

**Description**
(ODMG only) Addition operator that adds `d_Interval s`.

---

*S-28 Using Objectivity/C++*
d_Interval

OPERATOR d_Interval::operator -
SYNTAX d_Interval d_Interval::operator -() const;
DESCRIPTION (ODMG only) Unary minus operator for d_Interval.

OPERATOR ::operator -
SYNTAX friend d_Interval ::operator -(
    const d_Interval &left,
    const d_Interval &right);
DESCRIPTION (ODMG only) Subtraction operator that subtracts d_Interval.

OPERATOR ::operator *
SYNTAX friend d_Interval ::operator *(d
    const d_Interval &left,
    int right);
DESCRIPTION (ODMG only) Multiplication operator that multiplies d_Interval by an integer value.

OPERATOR ::operator *
SYNTAX friend d_Interval ::operator *(d
    int left,
    const d_Interval &right);
DESCRIPTION (ODMG only) Global multiplication operator that multiplies an integer value by d_Interval.
**OPERATOR** ::operator /  
**SYNTAX**  
friend d_Interval ::operator / (  
const d_Interval &left,  
int right);  
**DESCRIPTION** (ODMG only) Division operator that divides d_Interval by an integer value.

**OPERATOR** d_Interval::operator +=  
**SYNTAX** d_Interval &d_Interval::operator +=(  
const d_Interval &);  
**DESCRIPTION** (ODMG only) Adds the interval d_Interval to the current d_Interval.

**OPERATOR** d_Interval::operator -=  
**SYNTAX** d_Interval &d_Interval::operator -= (  
const d_Interval &);  
**DESCRIPTION** (ODMG only) Subtracts the interval d_Interval from the current d_Interval.

**OPERATOR** d_Interval::operator *=  
**SYNTAX** d_Interval &d_Interval::operator *= (  
int);  
**DESCRIPTION** (ODMG only) Multiplies a d_Interval by an integer value.

**OPERATOR** d_Interval::operator /=  
**SYNTAX** d_Interval &d_Interval::operator /= (  
int);  
**DESCRIPTION** (ODMG only) Divides the current d_Interval by an integer value.
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator ==</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator == ( const d_INTERVAL &amp;left, const d_INTERVAL &amp;right );</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Global equality operator for d_Intervals.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator !=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator != ( const d_INTERVAL &amp;left, const d_INTERVAL &amp;right );</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Inequality operator for d_Intervals.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &lt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator &lt; ( const d_INTERVAL &amp;left, const d_INTERVAL &amp;right );</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Less than operator for d_Intervals.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>::operator &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNTAX</td>
<td>friend int ::operator &gt; ( const d_INTERVAL &amp;left, const d_INTERVAL &amp;right );</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>(ODMG only) Greater than operator for d_Intervals.</td>
</tr>
</tbody>
</table>
OPERATOR ::operator <=
SYNTAX friend int ::operator <=(
    const d_Interval &left,
    const d_Interval &right);
DESCRIPTION (ODMG only) Less than or equal to operator for d_Interval.

OPERATOR ::operator >=
SYNTAX friend int ::operator >=(
    const d_Interval &left,
    const d_Interval &right);
DESCRIPTION (ODMG only) Greater than or equal to operator for d_Interval.

MEMBER FUNCTION d_Interval::day
SYNTAX int d_Interval::day() const;
DESCRIPTION (ODMG only) Returns the current day of the interval d_Interval.

MEMBER FUNCTION d_Interval::hour
SYNTAX int d_Interval::hour() const;
DESCRIPTION (ODMG only) Returns the current hour of the interval d_Interval.

MEMBER FUNCTION d_Interval::is_zero
SYNTAX int d_Interval::is_zero() const;
DESCRIPTION (ODMG only) Returns true if the interval is zero.
### MEMBER FUNCTION: `d_Interval::minute`

**SYNTAX**

```cpp
int d_Interval::minute() const;
```

**DESCRIPTION**

(ODMG only) Returns the current minute of the interval `d_Interval`.

---

### MEMBER FUNCTION: `d_Interval::second`

**SYNTAX**

```cpp
float d_Interval::second() const;
```

**DESCRIPTION**

(ODMG only) Returns the current second of the interval `d_Interval`. 
d_Interval
Predicate Query Language

The Query Browser and the predicate query C++ interface use a simple expression language. The language supports C and C++ operators and constant literals, and has the ability to refer to public data members. This appendix describes this language.

Overview

The query language gives you the ability to refer to a data member of the class, to use arithmetic (+, −, *, /, %), relational (<, >, <=, >=, ==, !=, ~==, !~), and logical (AND, OR, NOT) operators. It does not provide any other operators, the ability to declare variables, the use of pointers or addresses of any kind, or the ability to call functions or member functions of the scanned class. The language also does not handle enumerated constants (enum data types). The language ignores white space and new lines, which serve to separate tokens. The precedence of the operators is the same as the C and C++ equivalents of those operators.
 Operators and Literals

Objectivity/DB supports the following operators and literals.

- **AND**, `&&`  Logical conjunction operator
- **OR**, `||`   Logical disjunction operator
- **NOT**, `!`   Logical negation operator
- `>`           Greater than relational operator
- `<`           Less than relational operator
- `>=`           Greater than or equal to relational operator
- `<=`           Less than or equal to relational operator
- `=`, `==`      Equality relational operator
- `<>`, `!=`     Inequality relational operator
- `~~`         Regular expression string comparison
- `!~~`         NOT form for regular expression string comparison
- `=~~`         Case insensitive regular expression string comparison
- `!~~`         NOT form for case insensitive regular expression string comparison
- `member`      Unquoted string representing a class data member name, including to-one associations
- `intConstant` Integer constant literal
- `floatConstant` Floating point constant literal
- `characterConstant` 8-bit single-character literal (single-quoted)
- `stringLiteral` Character string literal (double quoted)
- `+`           Addition, unary plus sign operators
- `-`           Subtraction, unary minus sign operators
- `*`           Multiplication operator
- `/`           Division operator
- `%`           Remainder operator
- `(`           Left or opening parenthesis
- `)`           Right or closing parenthesis

The words **AND**, **OR**, and **NOT** are reserved words in the language; you cannot refer to a data member with one of these names in a predicate. You can mix upper and lower case in these reserved words—for example, **AND**, **and**, and **And** are all valid ways of specifying the `&&` operator.
Examples

The following expressions demonstrate some of the valid expressions you can use.

name = "ALU" AND productName = "Pegasus"
name == "ALU" && productName == "Pegasus"
pins > 3
cost <= 3.50 && NOT inDesign
((cost + 3.50 + padFactor) <= totalBudget) AND inProduction
aToOneAssoc.aMember == 5; // a to-one association member
aToOneAssoc->aMember == 5; // a to-one association member

The following expressions demonstrate invalid use of expressions.

name == ALU // ALU is not a member (you forgot quotes)
pins >* 3 // Unrecognized operator. What is >*?
((cost + 3.50 + padFactor) <= totalBudget AND inProduction
 // missing close parenthesis

Regular Expressions

For string comparisons, you can use regular expressions. The regular expression must match the entire target string. Objectivity/DB implements its regular expressions based on the POSIX extended regular expression library. Objectivity/DB also defines the following operators for doing regular expression comparisons:

 =~ String comparison.
!~ NOT form for string comparison.
==~ Case insensitive string comparison.
!==~ NOT form for case insensitive string comparison.
Supported regular expression metacharacters are summarized in the following table. For UNIX users, you can find additional information about regular expressions in the man page for `ed`. Note that no regular expression matches the newline character.

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\</code></td>
<td>Used as a prefix, <code>\</code> disables any special meaning a character has. Loses its special meaning when used within <code>[ ]</code>.</td>
</tr>
<tr>
<td>.</td>
<td>Used to match any single character. Loses its special meaning when used within <code>[ ]</code>.</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Used as a postfix, <code>*</code> matches zero or more instances of a character or range of characters. Loses its special meaning when used within <code>[ ]</code>.</td>
</tr>
<tr>
<td><code>^</code></td>
<td>When used to begin a pattern, <code>^</code> matches the beginning of the line.</td>
</tr>
<tr>
<td><code>[ ]</code></td>
<td>When used to bracket a string of characters, matches any one character in that string. If the first character of the string is a <code>^</code>, the regular expression matches any character except the remaining characters in the string. <code>[</code> stands for itself when used within <code>[ ]</code>.</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Used to indicate a range of consecutive ASCII characters. For example, <code>[0-5]</code> is equivalent to <code>[012345]</code>. Loses its special meaning if it is last in a string, or first in a string after an initial <code>^</code>.</td>
</tr>
<tr>
<td><code>$</code></td>
<td>When used to end a pattern, <code>$</code> matches the end of the line.</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>( )</code></td>
<td>Used to group patterns of characters or strings.</td>
</tr>
<tr>
<td><code>+</code></td>
<td>When used as a postfix, <code>+</code> matches one or more instances of a character or range of characters.</td>
</tr>
<tr>
<td><code>character</code></td>
<td>Matches any non-special character.</td>
</tr>
</tbody>
</table>
The following are examples of using regular expression operators for string comparisons.

```plaintext
field =~ \".*H.11.\" // anystring H anychar 11 anychar
field !~~ \"h.*o\" // h anystring o
field =~ \"a[0-9]+\" // a then digit(s)
field =~ \"(abc | def)\" // abc or def
```
Index

A

abort 2-14, H-2
  ooTrans 2-14
aborting transaction 2-14, B-2
access mode 5-2
  specifying A-5
  type A-1
access_status
  d Database K-52
accessing
  container 4-19
  database 4-20
  handle 4-2
  member field 4-10
  member function 4-10
  object reference 4-2
  objects 4-1
  persistent objects 4-5
  illustration 4-2
add
  ooMap 13-6, E-1
add_allVers 16-29, D-29
add_derivatives
  ooObj D-4
add_derivativesFrom 16-39, D-4
add_mapElems D-4
  ooMap 13-8
add_nameOfLink 7-8
  userClass Q-1
add_nextVers D-4
addEntry
  ooCursor 25-7

addField
  ooKeyDesc 14-9, F-2
  ooLookupKey 14-21, F-9
adding
  association 7-8
administration
  using the programming interface 24-1
advance
  d_Iterator J-7
affected objects 22-1
allVers D-29
allVers[]
  ooGeneObj 16-22
altering (see changing)
AMS 24-12
  ooAMSUsage
    oocAMSOnly 24-12
    oocAMSPrefered 24-12
    oocNoAMS 24-12
  ooSetAMSUsage 24-12, B-15
  setting usage 24-12
anyIndex F-9
  ooLookupKey 14-23
application
  developing 1-6
  linking 1-1
  multiple transactions in 2-3
  performance statistics 21-1
  runtime speed at map resize 21-15
  tuning for runtime speed 21-9
array
  fixed
    performance 21-7
  variable-size (see VArray)
arrow operator 4-9
association 7-1
  adding 7-8, 21-20, Q-1
  basic object 16-24
  bidirectional 7-1
    performance 21-6
  cardinality 7-3
  clustering 21-11
creating D-4, D-5
  in map 13-8
default version
deleting 16-35
deleting D-6, D-7, Q-1
  all 7-9
  all on link allVers D-31
default version D-31
  from object genealogy D-33
  from to-many link 7-8
  previous version D-8
  specified number D-18, D-19
  to-one link 7-7
existence 7-11, D-9, D-10, D-31, Q-2
genealogy object 16-21, 16-29, D-18
handle Q-2
inline 7-3
link 7-1
many-to-many 7-7, 7-13
many-to-one 7-4
next version D-10, D-11
non-inlinenone
  determining size 7-3
  performance 21-6
object genealogy D-29
object reference Q-3
one-to-many 7-7, 7-13
one-to-one 7-4, 7-13
order 7-3
performance 21-17, 21-19
previous version D-11, D-18
propagating 1-17, 7-11
removing Q-6
setting Q-5
to-one link 7-6
storing 7-3
storing data with 7-13
traversing O-3, Q-3, Q-4, Q-5
unidirectional 7-1
  performance 21-6
using 1-17
versioning 16-6, 16-20

automatic recovery
  enabling from application 24-13

B

base class
  obtaining objects 22-8
basic object
  (see also object)
  access rules 11-4
  association versioning member functions 16-24
  checking in K-7
  checking out K-7
class 1-2
closing K-7
classing 3-13, 21-5, 21-9, 21-11
containing container 9-16, K-8
copying 6-1, K-8
  bit-wise 6-4
down 6-2
  shallow 6-2
  with propagation 6-2
  without propagation 6-2
creating 3-13, D-1
deleting 3-18, K-59
  performance 21-8
deleting name from scope K-16
finding in scope K-9
locking 11-2, 11-14, K-11
looking up K-12
modifying 5-2
moving 6-8, K-13
naming K-13
next version K-10
obtaining name K-10
ooHandle class K-2
ooRef class K-2
open mode K-14
opening 5-6, K-13
previous version K-10
C

Contents

Go Back  Map  Contents  Quit

printing OID K-14
tracing events 17-4
traversing 9-13
    name scope K-9
type name K-16
type number K-15
updating K-16
validity K-11
verification 17-2
versioning 16-1
    mode K-15
begin
    ooTrans H-2
bidirectional association 7-1
boot file
    changing path name K-39
bootFilePath
    to open federated database K-47
branch versioning 16-2
buffer pages 21-10, 21-19
    initial number 21-19

checkin
    ooContObj 11-20
    ooObj 11-20, K-7
checkout
    ooContObj 11-19
    ooObj 11-19, K-7
checkpoint 2-9
    ooTrans H-2
child processes
    forking 2-3
class
    array
        ooVArray 1-11
    basic object 1-2
    container 1-2
cursor 25-1
database 1-2
date, ODMG 5-1
dictionary
    ooMapElem E-7
dictionary (see class, map)
federated database 1-3
handle
    ooHandle(ooContObj) K-17
    ooHandle(ooDBObj) K-27
    ooHandle(ooFDObj) K-38
    ooHandle(ooObj) K-2
    ooHandle(userClass) P-1
index
    non-persistent F-6
    ooKeyDesc 14-3
    ooKeyField 14-5
    persistent F-1
iterator
    ool tr(ooContObj) J-3
    ool tr(ooDBObj) J-5
    ool tr(ooObj) J-1
    ool tr(userClass) O-1

C

cache
    initial number of buffer pages in 21-19
    number of pages 21-13
    optimum size 21-10
cardinality 7-3
    ooVArray 1-2
chaining
    signal handler 18-1
change
    ooDBObj 24-8, 24-10, K-28
    ooDObj 24-2, K-39
changing
    version genealogy 16-42
char *
    ooString(N) 12-12, G-9
    ooVString 12-4, G-7
map 1-19, 21-4
  header file 13-2
  ooMap 13-2, E-1
  ooMapElem 13-14
  ooMapitr 13-15, E-8
name 20-4, D-14
  basic object K-16
  database K-37
  federated database K-50
object reference
  ooRef 4-3
  ooRef(ooContObj) K-17
  ooRef(ooDBObj) K-27
  ooRef(ooFDObj) K-38
  ooRef(ooObj) K-2
  ooRef(userClass) P-1
  ooShortRef 4-3
  ooShortRef(className) P-6
obtaining 20-4, D-15, D-23, D-25
ooContObj 3-7
ooDBObj 3-3
ooGeneObj 16-18
ooMap 1-20
ooQuery 9-24
ooString(N)
  performance 21-7
ooTrans 2-7
ooVString
  performance 21-7
parameterized A-8
persistent (see persistent class)
string 1-14, 12-1, 21-3
  choosing 12-1
  ooString(N) 1-14, 1-16, 12-8, G-8
  ooVString 1-15, 12-2, G-1
  performance 21-7, 21-16
system defined
  handle K-1

  system-defined
    (see also system-defined class)
    index
      non-persistent F-6
      persistent F-1
    iterator J-1
    map E-1
    object reference K-1
    persistent base D-1
    string G-1
    transaction H-1
    VArray I-1
time, ODMG S-1
type number 20-2, 20-3, D-14
  basic object K-16
  database K-36
  federated database K-50
user-defined
  (see also user-defined class)
  iterator O-1
  object reference and handle P-1
VArray
  ooTVArray I-6
  ooVArray I-1
class name
  indexed class F-3
clear 9-27
  d_Ref_Any K-59
  ooOperatorSet M-2
clearParam
  ooMap 13-9, E-2
close 5-10
  d_Database K-52
  ooContObj K-21
  ooDBObj K-29
  ooFDObj K-41
  ooObj K-7
clustering 21-2
  basic objects 3-13, 21-5, 21-11
  containers 3-10
  directive 3-10, 3-13
  optimum 21-11
commit
  ooTrans 2-9, H-2
commitAndHold
  ooTrans 2-12, H-3
composite object 1-17
  locking 11-14
concurrency
  control 1-3
  improving 9-12
  index operations 14-14
  MROW 2-8
  rules 11-4
  transaction 2-17
  tuning for 21-16
concurrent transactions 2-17
conditional predicate 9-18
configuration
  information 1-3
consistency
  index operations 14-14
constants 1-4
constraints
  checking D-16
containedIn 9-16
  ooContObj K-21
  ooDBObj K-30
  ooObj K-8
container
  access rules 11-4
  accessing 4-19
  checking update 11-9
  class 1-2
  closing K-21
  clustering 3-10
  contained basic objects K-21
  containing database 9-17, K-21
  creating 3-7, 21-6, D-20, D-22
    multiple 3-11, 21-6, B-11
  default 1-2, 20-1, 21-5, K-32
    as name scope 3-7, 8-6
    growth percentage 3-3, 21-9
    initial number of pages 3-3, 21-5
    size 21-9
  deleting 3-18
  existence 8-5, K-22
  growth factor K-26
  growth percentage 21-9, 21-20, D-21
  handle
    array of 3-11
    hash value K-23
  hashed 3-7, 21-4, 21-11
    creating D-21
    default size 21-9
    required for name scope 3-7
  initial number of pages D-21
  initial size 21-9
  locking 11-2, 11-14, K-23
  looking up K-24
  minimizing
    growth 21-6
    number 21-5
    size 21-5
  multiple 21-12
  non-contiguous 21-9
  non-hashed 3-7, 21-4, 21-11
  number of containers in database 20-6,
    K-34
  number to create 3-11
  ooHandle class K-17
  ooRef class K-17
  open mode K-25
  opening 5-6, K-25
  page count K-24
  refreshing view 11-10, K-26
  selecting for search 14-22
  setting K-14
  setting size 21-6
  system name K-24
  tracing events 17-4
  transient 3-7
traversing 9-13, K-21
update check K-23
verification 17-3
containment hierarchy 9-13
contains 9-13
(see also iterator)
 ooContObj K-21
 ooDBObj K-30
    performance 21-14
 ooFDObj K-41
conversion function 22-5
registering 22-6
conversion operator G-7, G-9
convertObjects
 ooContObj 22-3, K-22
 ooDBObj 22-3, K-30
 ooFDObj 22-3, K-42
copy
 ooObj 6-1, K-8
    userClass P-5
copying
 basic object 6-1
  shallow and deep 6-2
  status D-14
  with and without propagation 6-2
create_iterator
 ooVArray I-2
createlndex F-2
 ooKeyDesc 14-10
creating
 (see also setting) 7-8
 association D-4, D-5
 basic object 3-13, D-1
 container 3-7, D-20, D-22
    multiple 3-11
cursor 25-4
database 3-3, B-14, D-27
federated database 1-6, 3-2
hashed container D-21
index 14-3, 14-10, F-2
key description F-1
key field F-4
keyed object 3-7, 15-5, B-12
lookup field F-6
lookup key F-9
objects 3-1
 ooVString 12-2
persistent object 3-13, D-2
string G-1
transient object 3-16, 4-3, D-3
user-defined operator set 9-27
user-defined operators 9-25, M-1
VArray 10-3, I-1
current
 d_Date S-7
d_Time S-17
d_Timestamp S-24
cursor
 architecture 25-2
 checking query completion 25-6
 creating 25-4
 declaring 25-4
 implementing 25-4
 initializing 25-5
 iterator 25-1
 retrieving object 25-6, 25-7
 visible area 25-1
cursor class 25-1
cursor VArray 25-1
 adding object 25-7
 getting size of 25-6
 resizing 25-7
D
d_Boolean 23-2
d_Char 23-2
d_Database 23-3, K-52
d_Date 23-3, S-1, S-2
d_Double 23-2
d_Float 23-2
d_Interval 23-3, S-28
d_Iterator 23-3, J-6
Contents

getLong 23-2
getOctet 23-2
gPersistentObject 23-1, D-1
gRef 23-1, 23-1, K-2, P-1
gRefAny 23-3, K-55
gShort 23-2
gString 23-1, G-1
gTime 23-3, S-11, S-12
gTimestamp 23-3, S-20
gTransaction 23-1, H-1
gULong 23-2
gUShort 23-2
gVarray 23-1, I-1
Data Definition Language (DDL) 3-2
data member
ggetting objects 22-8
getvalue 22-6, 22-7
setvalue 22-7, R-6
data modeling 1-6
data types
permitted for index 14-2
data verification 17-2
database
access rules 11-4
changing attributes 24-8
changing host name K-28
changing path name K-28
class 1-2
closing K-29
contained containers K-30
containing federated database 9-17, K-30
creating 3-3, B-14, D-27
default 4-20, C-2
default container 3-5
deleting 3-18
extistence 5-5, 8-4, K-31
file (see database file)
getting information 24-9
iterating 9-13
locking 11-15, K-33
moving 24-10
multiple 4-20

number of containers in 20-6, K-34
ooHandle class K-27
ooRef class K-27
open mode K-34
opening 5-5, K-34
replacing 3-6, B-14
system name K-33
tidy 24-11, K-35
tracing events 17-4
traversing 9-13, K-30
type name K-37
type number K-36
updating K-37
validity K-33
database file 3-4
host name K-32
name K-31
path name K-35
date
d_Timestamp
   S-24
day
d_Date S-7
d_Interval
   S-32
d_Timestamp S-24
day of week
d_Date S-7
day of year
d_Date S-8
days in month
d_Date S-8
days in year
d_Date S-8
deadlock detection 11-7
debug mode 17-1
debugging 17-2
DEC Object/DB
macros
   ooGetMemberSize 15-4
   ooNewKey 15-5
declare 10-1
  ooString(N) 12-9
declaring
  cursor 25-4
  iterator 9-3
  persistent class 3-2
  VArray 10-1
decodeSchema
  ooFDObj 24-4, K-42
decoding schema 24-4
deep copy
  basic object 6-2
default container 1-2, 3-5, 21-5
    as name scope 3-7, 8-6
    avoiding locking conflicts 20-2
    finding 20-1, K-32
default database 4-20
default version (see versioning, default version)
defaultToGeneObj D-5, D-6
  ooObj 16-22
defaultVers 16-31, D-30
  ooGeneObj 16-22
defered object conversion 22-2
del_allVers D-31
del_defaultToGeneObj D-6
del_defaultVers 16-31, D-30
  ooGeneObj 16-22
del_nameOfLink 7-7, 7-9
  userClass Q-1
del_nextVers D-7
del_prevVers D-8
delete 3-19
  ooObj D-2
delete_object
    d_Ref_Any K-59
    ooObj K-8
deleting
  association D-6, D-7, D-8
    from to-many link 7-8
    to-one link 7-7
  basic object 3-18, K-8
    performance 21-8
  container 3-18
  database 3-18
default version support 16-35
derivatives D-19
  federated database 3-17
  index 14-13
  object K-59
  objects 3-1, B-5
  persistent object 3-18, D-2
  scope name 8-7
  transient object 3-21
  version 16-42
derivative versions
  adding D-4
deleting D-6, D-7, D-18
  existence D-9, D-10
  iterating over D-8, D-9
derivatives D-8
  ooObj 16-39
derivatives[]
  ooObj 16-22
derivedFrom D-8
  ooObj 16-39
derivedFrom[]
  ooObj 16-22
dictionary (see map)
dot operator 4-9
downgrading locks 2-12
dropIndex F-2
  ooKeyDesc 14-12
dumpCatalog
  ooFDObj 24-5, K-43
E

elem 21-14
  ooTVArray 1-7
  ooVArray 10-5, 1-3
encodeSchema
  ooFDObj 24-4, K-44
encoding schema 24-4
end
  ooltr(userClass) O-2
end user tasks
  object evolution 22-14
  schema evolution 22-14
environment variable
  OO_DB_NAME 1-8
  OO_FD_BOOT 1-8
equality operator G-2, G-9, K-5, K-56
errHandler 19-10
error
  condition 19-9
  fatal 19-5
  identifier 19-2, 19-3
  level 19-4
  type A-3
  message header file 19-3
  messages 19-2
  output file B-16
  non-fatal 19-4
  number 19-2
  signal 19-7
  system 19-5
  type A-3
  user 19-4
  variables 19-6
error flags 19-5
  clearing B-15
error handler 19-9
  calling B-17
  calling interface 19-10
  pointer B-7
  registering 19-12, B-13
  writing 19-10

evaluate
  ooQuery 9-24, J-9
events
  tracing 17-4
exec 2-3
exist 1-19
  ooContObj 8-5, K-22
  ooDBObj 8-4, K-31
  ooFDObj 8-4, K-45
exist_allVers D-31
exist_defaultToGeneObj D-9
exist_defaultVers D-31
exist_derivatives D-9
exist_derivedFrom D-9
exist_geneObj D-10
exist_mapElems D-10
exist_nameOfLink 7-11
  userClass Q-2
exist_nextVers D-10, D-11
exist_prevVers D-11
extend
  ooTVArray 10-12, l-7
  ooVArray 10-7, l-3

F
fatal error 19-5
federated database
  access rules 11-4
  automatic locking 11-3
  changing attributes 24-2
  changing boot file path K-39
  changing identifier K-39
  changing lock server host K-39
  class 1-3
  closing K-41
  contained databases K-41
  creating 1-6, 3-2
  decoding schema K-42
  deleting 3-17
  encoding and decoding schema 24-4
encoding schema K-44
existence 5-5, 8-4, K-45
getting information 24-5
identifier K-46
iterating 9-12
lock server name K-46
locking 11-15, K-46
ooHandle class K-38
ooRef class K-38
open mode K-48
opening 5-3, K-47
required to save updates 5-2
page size K-48
printing information K-43
system name K-46
tidy 24-7, K-49
traversing K-41
type name K-50
type number K-50
updating K-51
validity K-45

_fileName_
  ooDBObj 24-10, K-31

_files_
  database 3-4, K-31
  schema 3-2

_finding objects (see object, looking up)_
_for statement_
  iterating 9-10

_forceAdd_
  ooMap 13-6, E-2

_forking, of child processes_ 2-3

_free B-1

_function_
  ooGetResourceOwners B-8
  ooSetLargeObjectMemoryLimit 2-5
  registering B-13, B-14

_functions 1-5_
  free B-1
  ooAbort B-2
  ooCheckVTablePointer 20-6, B-2
  ooCleanup B-2

_ooCommit B-4_
_ooCommitAndHold B-5_
_ooGetActiveTrans B-6_
_ooGetErrorHandler 19-12, B-7_
_ooGetMsgHandler 19-14, B-7_
_ooInit 2-4, B-9_
_ooNoLock 11-17, B-12_
_ooRegAtStartFunc B-13_
_ooRegErrorHandler 19-12, B-13_
_ooRegMsgHandler 19-13, B-13_
_ooRegOnExitFunc B-14_
_ooReleasemarker B-14_
_ooResetError 19-6, B-15_
_ooRunStatus B-15_
_ooSetAMSUsage B-15_
_ooSetErrorFile B-16_
_ooSetLockWait 11-6, B-16_
_ooSetMark 4-14_
_ooSignal B-17_
realloc B-19
strcat B-20
strcpy B-20
strlen B-21
strncat B-21
strncpy B-21
strtok B-22

genealogy (see version genealogy)
genealogy object 16-18
  associating D-17, D-29
  association member functions 16-25
  creating 16-29
  default version 16-31, D-5, D-6, D-30
  deleting D-7
  existence D-10
  opening D-11
  using 16-36
  using to locate all versions 16-38
geneObj
  ooObj 16-22
get_element
  d_Iterator J-7
get_object_name
  d_Database K-53
getAbsolutEntry
  ooCursor 25-7
getDefaultContObj
  ooDBObj 20-1, K-32
getDefaultVers
  ooObj 16-14, K-9
getEntry
  ooCursor 25-6
getFloat32
  ooConvertInObject R-3
getFloat64
  ooConvertInObject R-4
getInt16
  ooConvertInObject R-2
getInt32
  ooConvertInObject R-2
getInt8
  ooConvertInObject R-1
getName
  F-5
getNameObj
  8-11
  ooObj 9-7, K-9
getNameScope
  ooObj 8-10, 9-6, K-9
getNewBaseClass
  ooConvertInObject R-5
getNewDataMember
  ooConvertInObject R-5
getNextVers
  ooObj 16-11, K-10
getObjName
  ooObj 8-9, K-10
getOldBaseClass
  ooConvertInObject R-4
getOldDataMember
  ooConvertInObject R-4
getPrevVers
  ooObj 16-12, K-10
getQuerySize
  ooCursor 25-6
getTypeN
  ooKeyDesc F-2
  ooKeyField F-5
getTypeName
  F-3
getUint16
  ooConvertInObject R-3
getUint32
  ooConvertInObject R-3
getUint8
  ooConvertInObject R-2
greater than operator (>) G-5
greater than or equal to operator (>=) G-6

handle
  1-4, 4-2
  class compatibility 4-7
  container
    array of 3-11
    explicit 21-14
    implicit 21-14
    member functions 4-8
    obtaining 4-12
    associated object 7-4
    basic object D-16
    container D-24
default container D-25
Immediate object conversion 22-4
Implement 10-2
Implement macro 1-12
Index 1-21

Hour
- d_Interval S-32
- d_Time S-17
- d_Timestamp S-25

Hash
- ooContObj K-23
- overflowing 21-9
- pages 3-7, 8-6, 21-5
- with maps 21-15

Hash function
- pointer type A-6

Hash table 1-19
- (see also map)
- dynamic, in ooMap 13-1

Hash value 3-7

Hashed container 3-7
- required for name scope 3-7

Head
- ooString(N) 12-10, G-10
- ooTVArray I-8
- ooVArray 10-5, I-3
- ooVString 12-3, G-7

Header file
- cursor 25-3
- index 14-2
- map class 13-2

HostName
- ooDBObj 24-9, K-32

Hot mode 2-6
multiple user access 21-17
number of fields F-3
performance 21-8
permitted data types 14-2
scope 14-1
unique 14-2, 14-3, 14-11
uniqueness F-3
updating B-19
value of indexed character G-9
versus scope names 21-12
indexed character
obtaining value 12-5
inequality operator G-3, G-9, K-5, K-57
init
ooCursor 25-5
initializing
iterator J-2
nameOfLink() 9-20
scan 14-23
key structure 15-3
map iterator
ooMapItr 13-15
operator= 13-15
Objectivity/DB 2-4, B-9
thread data structures B-10
inline association 7-3
insert_element I-3
installing
signal handler 18-1
is_between
  d_Date S-9
is_empty I-4
is_leap_year
  d_Date S-9
is_null
  d_Ref_Any K-59
  ooObj K-11
is_valid_date
  d_Date S-9
is_zero
  d_Interval
    S-32
isActive
  ooTrans 2-16, H-3
isConsistent F-3, F-5
  ooKeyDesc 22-16
  ooKeyField 22-16
isMember
  ooMap 13-4, E-3
isNamed F-5, F-6
isQueryComplete
  ooCursor 25-6
isUnique F-3
  ooKeyDesc 14-11
isUpdated
  ooContObj 11-9, K-23
isValid
  ooDBObj K-33
  ooFDObj 24-5, K-45
  ooObj 4-5, K-11
iterating
derivative versions D-8
locking objects 9-12
map D-12
mode 9-3
name scopes for object 8-10
techniques 9-10
through database 9-13
through federated database 9-12
through to-many association 7-7
versions D-13
iterator 1-18, 9-1
accessing object with 9-9
basic object J-1
closing O-2
concurrency 11-5
closecontainer J-3
cursor 25-1
database J-5
declaring 9-3
deleting object with 9-2
in for loop 9-10
in while loop 9-10
K

initializing 9-3
to traverse associated objects
nameOfLink() 9-20
to use index 14-23
nesting 9-2, 9-11
next object O-2
setting 9-7, 13-16, E-9
using 9-2
using index O-3

K

key description 14-1
adding key field 14-9, F-2
creating 14-3, F-1
key field 14-1
adding 14-9
adding to key description F-2
checking consistency F-3
consistency F-5
creating 14-5, F-4, F-5
name F-5
type A-5
key structure
initializing 15-3
keyed object 15-1
creating 3-7, 15-5, B-12
key structure 15-3
looking up 15-6, 21-9
minimize use 21-11
using member functions with 15-8

L

large objects
setting space 2-5, B-16
length
ooString(N) 12-10, G-10
ooVString 12-2, G-7
less than operator (>) G-4
less than or equal to operator (<=) G-5

linear versioning 16-2
linking
application 1-1
lock 11-15
downgrading 2-12
ooDBObj K-33
ooDObj K-46
ooObj K-11
preserving 2-12
releasing 2-9, 2-14, 9-12, 11-2
lock manager 11-1
lock server 11-1
changing host machine K-39
lock waiting 11-5, 11-6
modes H-4
locking 11-1
automatic 11-2, 11-12, 11-14
basic object 11-14, K-11
composite object 11-14
conflicts 11-5
container 11-14, K-23
database 11-15, K-33
deadlock detection 11-7
disabling 11-17
during object creation 11-2
explicit 11-12, 11-15
performance 21-16
federated database 11-15, K-46
mode 11-13
ooSetLockWait 11-6
performance 21-16
persistent lock 11-19
removing 11-20
propagating locks 11-2
read lock 11-14
two-phase 9-12, 11-13
update lock 11-14
without propagation semantics 11-16
lockNoProp 11-16
ooContObj K-23
ooObj K-12
lockServerName
  ooFDObj 24-5, K-46
looking up objects (see object, looking up)
lookup 13-11
  ooMap 13-4, E-3
lookup field
  creating F-6
  name F-6
lookup key 14-17
  creating 14-18, F-9
  field 14-20, F-9
  number of fields F-10
lookup_object
  d_Database K-53
lookupObj 8-9, 15-6
  ooContObj K-24
  ooObj K-12
  userClass P-5

M
macros 1-5, B-1
  declare 10-1
  implement 1-12, 10-2
  ooDelete 3-19, B-5
  ooDeleteNoProp 3-19, B-5
  ooGetMemberOffset 15-3, B-7
  ooGetMemberSize 15-4, B-7
  ooNewConts 3-11, B-11
  ooNewKey 15-5, B-12
  ooReplace 3-6, B-14
  ooTypeN 20-2, B-18
many-to-many association 7-7
  adding object 7-8
  deleting all associations 7-9
  deleting association 7-8
  storing data with 7-13
many-to-one association 7-4
  deleting 7-7
  setting 7-6
map
  adding new element 13-6, D-4, E-2
  associations
    existence D-10
    checking for name 13-4, E-3
    deleting
      associations D-7, D-19
      element E-6
    forcing addition of element E-2
    hash table parameters 13-9, E-2
    header file 13-2
    iterating D-12
    looking up
      string name 13-4, 13-11, E-3
    maximum average density E-4
    name hash function 13-13, E-4
    number of bins E-4
    percent growth for bins E-5
    performance 21-4
    referential integrity E-5, E-7
    replacing element E-6
    resizing 13-11, E-5
    runtime statistics E-5
    setting name hash function E-6
    total elements E-4
map (see ooMap)
map class 1-19
map dictionary (see map)
map iterator
  initializing E-8
mapElems D-12
mark_modified D-13
maxAvgDensity
  ooMap 13-9, E-4
member field
  accessing 4-10
  offset 15-3, B-7
  size 15-4, B-7
member function 1-5
  accessing 4-10
  arrow operator 4-9
  dot operator 4-9
  exist 1-19
  handle 4-8
  name 1-19
  object reference 4-8
  open 1-19
message handler 19-13
  pointer A-6, B-7
  registering 19-13, B-13
minute
  d_Interval S-33
  d_Time S-17
  d_Timestamp S-25
mode
  access 5-2
  iteration 9-3
  open
    basic object K-14
    container K-25
    database 5-5, K-34
    determining 5-8
    federated database K-48
    iterators 21-14
    read 2-1, 5-2, 9-12, 11-3
    setting
      basic object K-15
      specifying A-7
      update 5-2, 11-3
modifying
  objects 5-2
Month
  d_Date S-1
month
  d_Date S-9
  d_Timestamp S-25
move
  ooObj 6-8, K-12
moving
  basic object 6-8

MROW
  activating 2-8, 11-9, H-3
  deactivating 2-8, 11-9, H-3
  refreshing view of container 11-10
  serialization 11-8
  update transactions 11-8
MsgHandler 19-13
multiple transactions 2-3, 2-17

N
name 1-19, 13-14
  element 13-14, E-7
  looking up in map hash table E-3
  obtaining D-14
  ooContObj K-24
  ooDBObj K-33
  ooFDObj 24-5, K-46
  ooMapElem E-7
name hash function
  obtaining E-4
  setting 13-13, E-6
name scope 8-6
  default container 3-7, 8-6
nameHashFunction 13-13
  ooMap E-4
nameObj
  ooObj 8-7, K-13
nameOfLink 7-4, 7-7, 9-6, 9-20
  userClass Q-2, Q-3, Q-4
naming
  system name 8-1
  version of genealogy 16-41
naming conventions
  Objectivity /DB 1-4
  system name 8-2
naming objects (see object, naming)
nBin
  ooMap 13-9, E-4
nElement
  ooMap 13-9, E-4
nesting iterators 9-11
new
  ooContObj 3-7, 3-8, D-20, D-21, D-22, D-23
  ooDBObj 3-3, D-28
  ooObj 3-13, D-2, D-3
  transient object 3-17
  userClass L-1, L-2
next
  d_Date S-10
  d_Iterator J-7
  ooItr 9-7
  ooItr(ooContObj) J-3
  ooItr(ooDBObj) J-5
  ooItr(ooObj) J-1
  ooItr(userClass) O-2
  ooMapItr 13-16, E-9
nextVers D-13
  ooObj 16-26
nField F-3, F-10
non-fatal error 19-4
non-hashed container 3-7
non-inline association
  determining size 7-3
  performance 21-6
not_done
  d_Iterator J-8
nPage
  ooContObj K-24
number
  ooDObj 24-5, K-46
numContainedObjects
  ooRefHandle(ooContObj) 20-6
numContObjs
  ooRefHandle(ooDBObj) K-34

O
object
  accessing
    (see also accessing) 4-1
    in iteration 9-9
    mode 5-2, 5-9
  affected 22-1
  associating 21-13
  checking constraints D-16
  checking in 11-20
  checking out 11-19
  class 20-4
  class name D-14
  closing 5-10
  composite 1-17
  containment hierarchy 9-13
  copying P-5
    status D-14
  copying (see also basic object, copying) 6-1
  creating
    locking during 11-2
  existence 8-3
  finding in scope 8-11
  handle or object reference
    basic object D-16
    container D-24
  identifier (see object identifier)
    in virtual memory 4-3
  keyed (see keyed object)
  large
    setting space for 2-5, B-16
  locking (see locking)
  looking up 8-2, 8-9, P-6
    performance 21-15
  moving
    status D-15
  naming 1-19
    scope name 8-6
  opening
    automatic locking 11-2
persistent (see persistent object)
printing OID 20-1
redirected 22-13
scope name (see scope name)
transient (see transient object)
traversing 9-1, D-12
type name 20-4
type number 20-2, D-14
  basic object J-2
  container J-4
updating 5-9
object cluster (OC) 21-20
object conversion 22-1
classes R-1
  container level K-22
conversion function 22-5
  registering 22-6
database level K-30
defered 22-2
end user tasks 22-14
federated database level K-42
  immediate 22-4
  on-demand 22-3
  performance issues 22-3, 22-13
  registering conversion functions K-48
  upgrade application 22-4
user-defined functions N-2
Object Database Management Group
(see ODMG)
object identifier

object identifier (OID) 1-3, 4-1
  D-C-P-S format 1-3
  long 4-3
  obtaining 13-14, E-7
  printing K-14
  replacing E-6
  reuse after object deletion 4-1
  setting 13-14, E-7
  setting handle 4-13
  setting object reference 4-13
  short 4-3

object reference 1-4, 4-2, 21-3
  class compatibility 4-7
  classes 4-3
  member functions 4-8
  obtaining 4-12
    basic object D-16
    container D-24
    default container D-25
  ooContObj K-17
  ooDBObj K-27
  ooFDObj K-38
  ooObj K-2
  passing
    performance 21-14
    setting 4-7
    validity 4-5, K-11
Objectivity/DB
  global variables C-1
  initializing 2-4, B-9
  internal statistics B-15
    associations resized 21-19
    buffers used 21-19
    forced file closes 21-20
    multiple opens 21-18
    new OCBs 21-19
    number of hash overflows 21-20
    SM objects resized 21-20
    times OCs extended 21-20
  macros B-1
  naming conventions 1-4
  parameterized types A-8
  rebuilding database 21-13
  size of database 21-4, 21-10
  types A-1

objects
  deleting B-5
  modifying 5-2

ODMG 23-1
  application development 23-4
  date classes 5-1
  enabling 23-4
example 23-5
interface 23-1
time classes S-1
offset
member field 15-3, B-7
oid
ooMapElem 13-14, E-7
OID (see object identifier)
on-demand object conversion 22-3
one-to-many association 7-7
adding object 7-8
deleting all associations 7-9
deleting association 7-8
storing data with 7-13
one-to-one association 7-4
deleting 7-7
setting 7-6
storing data with 7-13
OO_DB_NAME 1-8
OO_DEBUG_FILE 17-4
OO_DEBUG_TRACE_CONTAINER 17-4
OO_DEBUG_TRACE_DATABASE 17-4
OO_DEBUG_TRACE_OBJECT 17-4
OO_DEBUG_TRACE_PAGE 17-4
OO_FD_BOOT 1-8
ooAbort B-2
ooAccessMode A-1
ooAMSUsage 24-12
ooBoolean A-2
ooAll A-1
ooAMSPOnly 24-12
ooAMSPPreferred 24-12
ooBranchVers 16-5, A-7
ooCharArray A-5
ooCharPtrT A-2
ooError 1-4, 19-2, A-6
ooExplicitUpdate 14-15, 21-12, H-4
ooFatalError 19-5, A-3
ooFloat32 A-5
ooFloat64 A-5
ooFloat64T A-2
ooHandleToNull 2-14
ooHandleToOID 2-14
ooCheckVTablePointer 20-6, B-2
ooHostLocal A-4
ooInSensitive 2-8, 14-15, A-4, H-3, H-4
ooInt16 A-5
ooInt32 A-5
ooInt32T A-2
ooCleanup 24-13, B-2
ooLinearVers 16-5, A-7
ooLockRead 11-14, A-5
ooLockUpdate 11-14, A-5
ooNative A-4
ooNoAMS 24-12
ooNoError A-3
ooNoOpen A-5
ooNoVers 16-5, A-7
ooNoWait H-4
ooCommit B-4
ooCommitAndHold B-5
ooContObj 3-7, D-20
ooConvertToObject 22-5, 22-7, 22-8
ooConvertToObject 22-5, 22-7, 22-8, R-5
ooCopyInit 6-4, D-14
ooPublic A-1
ooRead A-5
ooSensitive 14-15, A-4, H-4
ooString A-5
ooSuccess 1-4, 19-2, A-6
ooSystemError 19-5, A-3
ooTransNoWait H-4
ooUint16 A-5
ooUint32 A-5
ooUint32T A-2
ooUpdate A-5
ooCurrentThread B-5
ooCursor 25-4
ooCursor.h 25-3
ooUserError 19-4, A-3
ooWait H-4
ooWarning 19-4, A-3
ooDataType 9-26, A-2
ooRegOnExitFunc B-14
ooReleaseMark 4-14
ooReplace 3-6, B-14
ooResetError 19-6, B-15
ooRunStatus 21-1, 21-9, 21-10, B-15
to obtain statistics 21-18
ooSetAMSUsage 24-12
ooSetAMSUsage B-15
ooSetLargeObjectMemoryLimit 2-5, B-16
ooSetLockMode H-4
ooSetLockWait 11-6, B-16
ooSetMark 4-14
ooShortRef 4-3, 21-3
ooSetErrorFile B-16
ooSwitchThread B-17
ooTermThread B-18
ooThis 4-12
  ooContObj D-24
  ooDefaultContObj D-25, D-26
  ooGeneObj D-32, D-33
  ooObj D-16
ooTrans 2-7, H-1
ooTVArray 10-11, A-9, I-6
  assigning 10-12
  extending 10-12
  resizing 10-11
ooTypeN 20-2, B-18
ooTypeNumber A-7
ooUpdate 5-9, D-16
ooUpdateIndexes 14-16, B-18
ooUseIndex 9-21, B-19
ooUserDefinedOperators 9-25
ooValidate D-16

ooVArray 1-11, 10-1, 21-2, A-10, I-1
  extending 10-7
  resizing 10-5
ooVersMode 16-5, A-7
ooLastError 19-5, C-1
ooLastErrorLevel 19-5, C-1
ooNError 19-5, C-1
ooVoidFuncPtr A-7
ooVoidPtr A-4, A-7
ooStringValue 1-15, 12-2, 21-3, G-1
  assigning string G-2
  constructors 12-2
  ooString(N) G-9
  performance 21-7, 21-16
ooTopDB 3-9, 4-20, C-2
ooTopFD C-2
ooTopTrans C-2
open 1-19
  d_Database K-53
  FDObj 5-3
  ooContObj 5-6, K-25
  ooDBObj 5-5, K-34
  ooFDObj 24-13, K-47
  ooObj K-13
opening
  basic object 5-6
  container 5-6
  database 5-5
  federated database 5-3
  required to save updates 5-2
  persistent object 5-1
openMode 5-8
  ooContObj K-25
  ooDBObj K-34
  ooFDObj K-48
  ooObj K-14
operator - S-4, S-14, S-15, S-22, S-29
d_Interval S-29
operator --
d_Date S-4, S-5
operator != G-3, G-4, K-57, K-58, S-6, S-16, S-23, S-31
  ooObj K-5
  ooString(N) 12-13, G-9
  ooVString 12-5
operator * S-29
  ooContObj K-19
  ooObj K-4
  userClass P-3
operator *=
  d_Interval S-30
operator + S-3, S-4, S-14, S-21, S-22, S-28
operator +=
  d_Date S-4
  d_Iterator J -6, J -7
operator <=
  d_Date S-5
  d_Interval S-30
  d_Time
    S-15
  d_Timestamp S-22
  ooString(N) 12-12, G-9
  ooVString 12-4, G-2
operator . 4-9
operator / S-30
operator /=
  d_Interval S-30
operator < S-4, S-6, S-16, S-23, S-31
operator <= G-5, G-6, S-7, S-16, S-24, S-32
operator == G-5, G-6, S-7, S-16, S-24, S-32
operator != G-3, G-4, K-57, K-58, S-6, S-16, S-23, S-31
  ooObj K-5
  ooString(N) 12-13, G-9
  ooVString 12-5
operator * S-29
  ooContObj K-19
  ooObj K-4
  userClass P-3
operator *=
  d_Interval S-30
operator + S-3, S-4, S-14, S-21, S-22, S-28
operator +=
  d_Date S-4
  d_Iterator J -6, J -7
operator <=
  d_Date S-5
  d_Interval S-30
  d_Time
    S-15
  d_Timestamp S-22
  ooString(N) 12-12, G-9
  ooVString 12-4, G-2
operator . 4-9
operator / S-30
operator /=
  d_Interval S-30
operator < S-4, S-6, S-16, S-23, S-31
operator <= G-5, G-6, S-7, S-16, S-24, S-32
operator == G-2, G-3, K-56, K-57, S-6, S-15, S-23, S-31
  ooObj K-5
  ooString(N) 12-13, G-9
  ooVString 12-5
operator -> 4-3, 4-9, 21-14
  ooContObj K-19
  ooObj K-4
  userClass P-3
operator > G-5, S-6, S-16, S-23, S-31
operator >= G-6, G-7, S-7, S-17, S-24, S-32
operator [] 21-14
  ooString(N) 12-12, G-9
  ooTVArray I-7
  ooVArray 10-4, I-2
  ooVString 12-5, G-2
operator char *
  ooString(N) 12-12, G-9
  ooVString 12-4, G-7
operator className*
  userClass P-4
operator const ooContObj* K-20
operator const ooObj* K-6
operator d_Ref_Any
  ooContObj K-20
  ooObj K-6
  userClass P-4
operator delete 3-19
  ooObj D-2
operator int
  ooObj K-6
**operator new**
- ooContObj 3-7, D-20, D-21, D-22, D-23
- ooDBObj 3-3, D-28
- ooObj D-2, D-3
- userClass L-1, L-2

**operator ooRef(ooObj) K-7**

**operator ooVString**
- ooString(N) G-9

**operator set**
- creating user-defined 9-27

**operators**
- on user-defined classes L-1
- predicate query T-2
- user-defined 9-25, M-1
  - clearing 9-27
  - creating 9-25
  - registering 9-27
  - using 9-28

**overlaps**
- d_Date S-10, S-11
- d_Time S-19
- d_Timestamp S-26, S-27

**pathName**
- ooDBObj 24-9, K-35

**percentGrow**
- ooContObj K-26
- ooMap 13-10, E-5

**performance 21-1**
- after object conversion 22-13

**persistent class 1-3**
- declaring 3-2
- ooContObj D-20
- ooDBObj D-27
- ooDefaultContObj D-25
- ooFieldObj D-28
- ooGeneObj D-29
- ooKeyDesc F-1
- ooKeyField F-4
- ooMap 13-2, E-1
- ooMapElem 13-14, E-7
- ooMapItr 13-15, E-8
- ooObj D-1
- ooVString G-1
- type number 20-2

**persistent lock (see locking)**

**persistent object 3-1**
- (see also object)
- accessing 4-2, 4-5
- clustering 21-2
- containing object 9-16
- creating 3-13, D-2
- deleting 3-18, D-2
  - OID reuse after 4-1
- handle 4-7
- kinds of 3-2
- locking 11-15
- modifying 5-3
- object reference 4-7
- opening 5-1
- printing OID 20-1

**page**
- buffer 21-10
  - initial number 21-19
- hash 3-7, 8-6
- size 2-5
  - and performance 21-13
- tracing events 17-4
- verification 17-3

**pageSize**
- ooF Obj 24-5, K-48

**parameterized class**
- ooHandle A-8
- ooInt A-8
- ooRef A-8, A-9
traversing 9-6
updating D-16
virtual table 20-6, B-2

**predicate query 9-18**
language T-1
literals T-2
operators T-2
regular expressions T-3

**previous**
d_Date S-10
prevVers D-17
ooObj 16-27

**primitive types**
getting data member values R-5
setting data member values R-5

**print**
ooObj 20-1, K-14

**printStat**
ooMap 13-10, E-5

**propagation**
association 7-11

**ptr**
ooContObj K-26
ooObj K-14
userClass P-6

**Q**

query language (see predicate query language) T-1

**R**

read lock 11-14, 11-17
read mode 2-1, 5-2
realloc B-19
recovery
application 24-13
application error messages 24-17
automatic 24-13

**recovery tasks**
creating recovery application 24-13
writing applications for 24-13

**redirected objects 22-13**

**refEnable**
ooMap 13-12, E-5

**referential integrity 7-2**
flag 13-12, E-7
maintaining E-5
maintaining, in map hash table 13-20

**refreshOpen**
ooContObj 11-10, K-26

**registering**
error handler B-13
function B-13, B-14
message handler B-13
operators 9-27

**registerOperator 9-27**
ooOperatorSet M-1

**regular expressions**
for predicate query T-3

**rehash**
ooMap 13-11, E-5

**relational operator functions**
user-defined N-1

**remove**
ooMap 13-8, E-5

**remove_all**
ooVArray I-4

**removal indexes F-4**
ooKeyDesc 14-13

**removing (see deleting)**

**rename_object**
d_Database K-53

**replace**
ooMap 13-5, E-6

**replace_element_at**
ooVArray I-4

**replacing**
database 3-6

**reset**
d_Iterator J-8
resize
  ooCursor 25-7
  ooString(N) 12-10, G-10
  ooTVArray 10-11, I-8
  ooVArray 10-5, I-5
    minimizing calls to 21-14
  ooVString 12-3, G-7
resizing
  VArray for cursor 25-7
retrieve_element_at
  ooVArray I-5
return value conventions 1-4

S

scan 9-4, 9-18
  ooltr 14-23
  ooltr(ooContObj) J-3
  ooltr(ooDBObj) J-5
  ooltr(ooObj) J-2
  ooltr(userClass) O-3
performance 21-14
scanning objects (see iterator)
schema
  (see also data modeling)
    decoding K-42
    encoding K-44
    encoding and decoding 24-4
    file 3-2
schema evolution
  end user tasks 22-14
  object conversion 22-1
scope
  index 14-1
  searching
    using lookup key 14-18
scope name 8-6
  basic object K-10
  default container as scope 3-7, 8-6
  deleting 8-7
  hashed container required 3-7
minimizing use 21-11
  object lookup 8-9, 21-9
  obtaining 8-9
  setting 8-7, K-13
  storage requirement 21-4
  valid 8-6
second
  d_Interval S-33
  d_Time
  S-17
  d_Timestamp
  S-25
set
  ooTVArray I-8
  ooVArray 10-4, I-5
set_container
  ooObj K-14
set_default_Time_Zone
  d_Time
  S-18
set_default_Time_Zone_to_local
  d_Time S-18
set_defaultToGeneObj 16-29, D-17
set_defaultVers D-33
set_geneObj 16-29, D-18
set_nameHashFunction
  ooMap 13-13, E-6
set_nameOfLink 7-6
  userClass Q-5
set_object_name
  d_Database K-53
set_oid 13-14
  ooMapElem E-7
set_prevVers D-18
set_refEnable
  ooMap 13-12, 13-16, E-7
setConversion
  ooFDObj 22-6, K-48
setDefaultVers
  ooObj 16-13, K-15
setenv command 17-2
setFloat32  ooConvertInOutObject R-8
setFloat64  ooConvertInOutObject R-8
setInt16  ooConvertInOutObject R-6
setInt32  ooConvertInOutObject R-6
setInt8  ooConvertInOutObject R-6
setlocale 14-26
setting
  association
to-one link 7-6
default version 16-13
iterator 9-7
scope name 8-7
setting large object space 2-5, B-16
setUInt16  ooConvertInOutObject R-7
setUInt32  ooConvertInOutObject R-7
setUInt8  ooConvertInOutObject R-7
setup
  ooQuery 9-24, J-9
setVersStatus 16-5
  ooObj K-15
shallow copy
  basic object 6-2
SIGKILL 18-2
signal
  error 19-7
  handlers 18-1
  ignoring 18-2
sigvec 18-2
size
  database 21-4, 21-10
  ooTVArray I-8
  ooVArray 10-5, I-5
sprint
  ooObj K-15
start
  ooTrans 2-8, H-3
starting
  transaction 2-7
statistics
  application 21-1
  internal 21-18, B-15
  performance 21-1
  runtime 13-10, E-5
status
  code 1-4, 19-2
  return type A-6
storage
  deallocating B-1
strcat B-20
strcoll 14-26
strcpy B-20
string
  adding characters B-21, G-8
  assigning to ooVString G-2
class 1-14, 12-1, 21-3
  ooString(N) 12-8, G-8
  ooVString 12-2, G-1
concatenating 12-4, 12-12, G-2
considered as token sequence B-22
converting
  to const char* G-7, G-9
  to ooVString G-9
converting to const char* 12-4
creating B-20
declaring 12-9
equality 12-5, 12-13, G-9
implementing 12-9
inequality G-9
length 12-2, 12-10, G-7, G-10
looking up in map hash table E-3
name look up 13-11
obtaining length B-21
performance 21-16
pointer to first character 12-3, 12-10, G-7, G-10
replacing characters 12-11, G-8
resizing 12-3, 12-10, B-20, G-7, G-10
storing efficiently 21-16

`strlen B-21`
`strncat B-21`
`strncpy B-21`
`strtok B-22`
`sub_allVers D-33`
`sub_derivatives D-18`
`sub_derivedFrom D-19`
`sub_mapElems D-19`
  `ooObj 13-9`
`sub_nameOfLink 7-8`
  `userClass Q-6`
`sub_nextVers D-19`
`system error 19-5`
`system name 1-19, 8-1`
  `container K-24`
  `database K-33`
  `federated database K-46`
  `naming conventions 8-2`
  `obtaining 8-2`
  `setting 8-2`
  `valid 8-2`
`system-defined class`
  `ooContObj D-20`
  `ooDBObj D-27`
  `ooDefaultContObj D-25`
  `ooEqualLookupField F-6`
  `ooFDobj D-28`
  `ooGeneObj D-29`
  `ooGreaterThanOrEqualLookupField F-8`
  `ooGreaterThanOrEqualLookupField F-7`
  `ooHandle(ooContObj) K-17`
  `ooHandle(ooDBObj) K-27`
  `ooHandle(ooFDObj) K-38`
  `ooHandle(ooObj) K-2`
  `ooItr(ooContObj) J-3`
  `ooItr(ooDBObj) J-5`
  `ooItr(ooObj) J-1`
  `ooKeyDesc F-1`
  `ooKeyField F-4`
  `ooLessThanOrEqualLookupField F-8`
  `ooLessThanOrEqualLookupField F-7`
  `ooLookupFieldBase F-6`
  `ooLookupKey F-9`
  `ooMap E-1`
  `ooMapElem E-7`
  `ooMapItr E-8`
  `ooObj D-1`
  `ooRef(ooContObj) K-17`
  `ooRef(ooDBObj) K-27`
  `ooRef(ooFDObj) K-38`
  `ooRef(ooObj) K-2`
  `ooString(N) 12-8, G-8`
  `ooTrans H-1`
  `ooTVArray I-6`
  `ooVArray I-1`
  `ooVString 12-2, G-1`

`thread`
  and concurrent transactions 2-17
  current B-5
  initializing data structures B-10
  switching B-17
  terminating B-18
`tidy 24-7, 24-11`
  `ooDBObj 24-11, K-35`
  `ooFDObj 24-7, K-49`
`time`
  d_Timestamp
  S-25
`time command 21-1`
`Time_Zone`
  d_Time S-11
`to-many association (see one-to-many, many-to-many)`
`tools`
  `oonewfd 1-6, 2-5`
to-one association (see one-to-one, many-to-one)

**transaction** 2-7
- aborting 2-9, 2-14, B-2, H-2
- effect on ooRef and ooHandle 4-7
- active 2-16, H-3
- checkpoint 2-9, 2-12
- class H-1
- commit and hold 2-12, B-5, H-3
- committing 1-10, 2-9, B-4, H-2
- concurrency of 2-17
- getting information 24-6
- holding resources B-8
- iterator-based 9-12
- listing active B-6
- maintaining 2-7
- model 2-1
- multiple 2-3
- multiple in application 2-3, 2-17
- starting 2-7, H-2, H-5
- steps in 2-1
- update and MROW 11-8

**transaction object** 2-7

**transient object**
- creating 3-16, 4-3, D-3
- deleting 3-21
- limitations 3-16

**transient memory**
- d_Database K-54

**two-phase locking** 9-12, 11-13

**type**
- key field class F-5
- number
- obtaining B-18

**type name**
- obtaining 20-4, D-14
  - basic object K-16
  - database K-37
  - federated database K-50

**type number**
- obtaining 20-2, D-14
  - basic object J -2, K-15
  - container J -4
  - database K-36
  - federated database K-50
  - of persistent class A-7

**typeName** 20-4
- ooDBObj K-36
- ooFDObj K-50
- ooItr(ooContObj) J -4
- ooItr(ooObj) J -2
- ooObj K-15

**types**
- ooAccessMode A-1
- ooBoolean A-2
- ooError A-3
- ooErrorHandlerPtr A-3
- ooErrorLevel A-3
- ooFileNameFormat A-4
- ooIndexMode A-4
- ooKey A-4
- ooKeyType A-5
- ooLockMode A-5
- ooMode A-5
- ooMsgHandlerPtr A-6
- ooStatus A-6
- ooTypeNumber A-7
- ooVersMode A-7
- ooVoidPtr A-7
- parameterized A-8

**tz_hour**
- d_Time S-18
- d_Timestamp S-25
tz_minute
  _d_Time S-18
  _d_Timestamp
  S-26

U

uint32 A-4
underscore 1-5
unidirectional association 7-1
unnameObj
  ooObj 8-7, K-16
update
  ooDBObj 5-5, K-37
  ooFieldObj K-51
  ooObj K-16
  ooTVArray I-9
  ooVArray 10-5, I-5
update lock 11-2, 11-14, 11-17
update mode 5-2, 11-3
upgrade
  ooTrans 22-5, H-5
upgrade application 22-4
upgrade interface 22-4
upgradeObjects
  ooFieldObj 22-5, K-51
upper_bound I-6
user error 19-4
UserClass L-1, Q-1
userConversionFunction 22-6, N-2
user-defined class L-1
  ooHandle(userClass) P-1
  ooTR(className) O-1
  ooRef(userClass) P-1
user-defined classes
  member functions Q-1
user-defined object conversion
  functions N-2
user-defined operator set
  creating 9-27
user-defined operators 9-25, M-1
  clearing 9-27
  creating 9-25
  registering 9-27
  using 9-28
userRelOperFunction N-1

V

validity
  database K-33
  federated database K-45
variable 1-4
  (see also global variable)
variable-size array (see VArray)
VArray 1-11, 10-1, 21-2
  (see also ooTVArray, ooVArray)
  allocating new element I-3, I-7
  assigning 10-9
  class ooTVArray 10-11
  class ooVArray 10-11
  copying I-2, I-7
  creating 10-3, I-1
  cursor 25-1
  declaring 10-1
  elements 10-5
  extending 10-7, I-3, I-7
  implementing 10-2
  new element 10-7
  of handles 10-12
  performance 21-7
  pointer to first element 10-5, I-3, I-8
  resizing I-5, I-8
    performance 21-14
  returning element I-3, I-7
  saving modifications 10-5, I-5, I-9
  setting element value 10-4, I-5, I-8
  size 10-5, I-5, I-8
    initial 21-14
system default association 7-3
temporary I-7
transient 10-9, 10-11
updating 10-5, I-6, I-9
verification
basic object 17-2
container 17-3
data 17-2
page 17-3
version genealogy 16-3
changing 16-42
changing order 16-43
default version 16-31
traversing 16-26
versioning 16-1, D-19
adding derivatives D-4
advanced topics 16-16
associating
  to genealogy object D-18
  to previous version D-18
association 16-20
association links 16-6
association member functions 16-24
branch 16-2
copy semantics 16-10, D-15
creating version 16-6
default object D-17
default version 16-3, D-6, D-9, D-30, K-9
  changing 16-33
deleting 16-33, D-31
deleting support 16-35
locating 16-14, 16-31, 16-36, D-5, D-6,
  D-30
  obtaining K-9
  setting 16-13, 16-33, D-33, K-15
  specifying 16-29
deleting
  derivatives D-6, D-7, D-18
genealogy object D-7
  version 16-42
derivatives 16-40, D-8, D-9
  existence D-9, D-10
derived version 16-39
  disabling 16-5, 16-8
  enabling 16-5
genealogy (see version genealogy)
genealogy object D-9, D-29
  existence D-10
  opening D-11
interface 16-1
iterating D-13
linear 16-2
locating all versions D-29
merging version branches 16-39
mode 16-5, K-11
naming version 16-41
next version 16-11, D-4, D-10, D-11, D-13,
  K-10
  deleting D-7, D-19
persistent objects 16-2
previous version 16-12, 16-27, D-11,
  D-17, K-10
  deleting D-8
  semantics 16-20
setting mode 16-5, K-15
traversing genealogy 16-26
version branches 16-39
virtual function 1-8
virtual member function 1-7
virtual memory
  pointer 4-3
virtual table
  persistent object 20-6, B-2
visible area
  cursor 25-1
void
  pointer A-7
VString (see ooVString)
W

Weekday
  d_Date S-2
while loop
  iterating 9-10

Y

year
  d_Date S-10
  d_Timestamp
    S-26