neoAccess™

cross-platform

object database engine

version 5

Technical overview

neoAccess™

Technical Overview
Version 5.0

Cross-Platform Object Database Engine

neo•logic
NeoLogic Systems, Inc.
1450 Fourth Street, Suite 12
Berkeley, CA 94710
Vox: (510) 524-5897
Fax: (510) 524-4501
Email: neologic@neologic.com
Web: http://www.neologic.com

NeoAccess Technical Overview, Version 5.0

By Bob Krause, Heidi Langius, Joel VanderWerf and Alexander Vladimirsky

Special thanks to Richard Aurbach, David Bienvenu, Brian Blackman, Jean-François Brouillet, Jeff Hokit, Suresh Kumar, Paul Ossenbruggen, Peter Reeves, Scott Ribe, Mike Rockwell, Reede Stockton, Jeff Winkler and the thousands of other developers who have generously given us their criticism and praise. NeoAccess would not be the rich and robust tool that it is without their input and support.

Bob Krause would also like to offer his warm thanks to his family, Marc Bernstein, Chris Buehler, Tim Duane, Rick Hoskins, Robert Inchausti, Philip Kaake, Cindy Lee, Todd Logan, Theresa McGlashan, Tricia Parrish, Lisa Piercey, Tim Standing, Larry Zulch, and Laura Zulch for their support and encouragement.

NeoLogic Systems, Inc.
NeoAccess Developer's Toolkit
Software License Agreement and
Limited Warranty

PLEASE READ THIS LICENSE CAREFULLY. THIS IS A LEGAL AGREEMENT BETWEEN THE END USER ("LICENSEE") AND NEOLOGIC SYSTEMS, INC. ("NEOLOGIC"). THE ENCLOSED SOFTWARE AND DOCUMENTATION ARE LICENSED BY NEOLOGIC TO THE ORIGINAL INDIVIDUAL CUSTOMER FOR USE ONLY ON THE TERMS DESCRIBED IN THIS LICENSE AGREEMENT (THIS "LICENSE"). OPENING THE ENCLOSED DISKETTE ENVELOPE AND/OR USING THE SOFTWARE INDICATES THAT THE END USER ACCEPTS AND AGREES TO COMPLY WITH THESE TERMS. IF THERE IS ANY DISAGREEMENT ON THESE TERMS, (A) THE SOFTWARE MAY BE BE RETURNED IN THE UNOPENED DISKETTE ENVELOPE TO THE PLACE WHERE IT WAS OBTAINED FOR A FULL REFUND, OR (B) WRITE TO NEOLOGIC WITH A REQUEST TO MODIFY ANY TERM OF THIS LICENSE. PLEASE ALLOW 6 WEEKS FOR A RESPONSE.

1. DEFINITIONS.
(a) "NeoAccess Source Code" shall mean the source code and building blocks contained within the NeoAccess Developer's Toolkit ("Toolkit") and any future versions and derivatives developed by NeoLogic and supplied to Licensee hereunder.
(b) "NeoAccess Object Code" shall mean object code either resulting from the compilation of NeoAccess Source Code or included as object code in the Toolkit.
(c) "Licensee Program(s)" shall mean Licensee's own computer software end-user application programs resulting from the compilation of its source code with NeoAccess Source Code. It is expressly understood and agreed that Licensee Program(s) shall contain substantial added value over and above that which is contained in the Software. Licensee Program(s) may not, in any event, consist of or include a programmatic interface, software developer tools or development environments.
(d) "Software" shall mean NeoAccess Source Code, NeoAccess Object Code and related documentation, whether printed, on disk or contained in any other medium.

2. LICENSES.
(a) NeoLogic hereby grants to Licensee a personal, worldwide, non-exclusive, non-transferable, license (without the ability to sublicense) to (i) use, modify and compile NeoAccess Source Code included in the Toolkit for the development and testing of Licensee Programs, and (ii) use and distribute NeoAccess Object Code solely as part of Licensee Program(s).
(b) NeoLogic retains title to the Software in all forms whatsoever. This License allows Licensee to use the Software on a single CPU and make one copy of the Software in machine-readable form for backup purposes.
(c) All rights not expressly granted herein are reserved by NeoLogic.
(d) For the sole purpose of monitoring Licensee's compliance with the terms hereof, Licensee shall upon request deliver a copy of Licensee Program(s) to NeoLogic.

3. LABELING.
(a) Licensee shall include conspicuously in the manuals, in Licensee Program(s) "About Box", and on each copy of Licensee Program(s), a copyright notice as follows:

> "**This Program was developed using NeoAccess : © 1992-1997 NeoLogic Systems, Inc.**"

(b) Licensee shall also include in a conspicuous place in association with the Licensee Program(s), preferably in the manual, a warranty disclaimer as follows:

> "**The NeoAccess software contained within this program is proprietary to NeoLogic Systems, Inc. and is licensed to (Licensee Name) for distribution only for use in combination with the (Licensee Programs). NeoLogic Systems, Inc. makes no warranties whatsoever, expressed or implied, regarding this product, including warranties with respect to its merchantability or its fitness for any particular purpose.**"

(c) NeoLogic may, during the term of this Agreement, require revisions or additions to the notices to be placed in the manuals, including the copyright notice and the warranty disclaimer notice. Licensee shall incorporate such revisions or additions to the notices required herein, in all future printings or versions of Licensee

Program(s), but in no event shall such revisions or additions occur later than 180 days following written notice from NeoLogic.

4. SUPPORT**.** Provided that Licensee has returned to NeoLogic its product registration card fully filled out, NeoLogic shall supply Licensee with online technical support in the use of the Software for a period of thirty (30) days from date of acquisition of the Software. Application-specific support and support requested after the expiration of the initial thirty (30) day period may be provided, at the option of NeoLogic, at NeoLogic's then current standard hourly rates.

5. LICENSE FEES. This license shall have no force or effect unless and until Licensee shall have submitted to NeoLogic all applicable license fees in full. All such fees are exclusive of any taxes, duties, licenses, fees, excises or tariffs now or hereafter imposed on Licensee's production, licensing, sale, transportation, import, export or use of the Software or Licensee Programs, all of which shall be the responsibility of Licensee, other than taxes attributable to NeoLogic's net income.

6. CONFIDENTIALITY. By accepting this License, the end user acknowledge that the Software consists of information which is of a confidential and proprietary nature. Such information includes, but is not limited to know-how, techniques, processes, programs, source code, data and other trade secrets ("Proprietary Information"). NeoLogic owns and intends to maintain its ownership of all such Proprietary Information. The end user shall at all times, both during the term of this License and thereafter, maintain in the strictest confidence and trust all such Proprietary Information, and shall not use such Proprietary Information other than as authorized under this Agreement, nor shall the end user disclose any of such Proprietary Information to any third party without the express prior written consent of NeoLogic.

7. LIMITED WARRANTY**.**
(a) NeoLogic warrants that for one (1) year following delivery of the Software to Licensee, the Software, unless modified in any way by Licensee, will perform substantially the functions described in any associated product documentation provided by NeoLogic. NeoLogic does not warrant that the Software will meet Licensee's specific requirements or that operation of the Software will be uninterrupted or error-free. NeoLogic is not responsible for any problem, including any problem which would otherwise be a breach of warranty, caused by (i) changes in the operating characteristics of computer hardware or computer operating systems which are made after delivery of the Product, (ii) interaction of the Software with software not supplied or approved by NeoLogic, or (iii) accident, abuse, or misapplication.
(b) NeoLogic's entire liability and Licensee's sole remedy under the foregoing warranty during the warranty period is that NeoLogic shall, at its sole and exclusive option, either use reasonable efforts to correct any reported deviation from the relevant product documentation, replace the Software with a functionally equivalent program, or refund all license fees paid, in which case, this License shall immediately terminate. Any repaired or replaced Software will be rewarranted for an additional ninety (90) day period, unless subsequently modified by Licensee.
(c) THE ABOVE WARRANTIES ARE EXCLUSIVE AND NO OTHER WARRANTIES ARE MADE BY NEOLOGIC OR ITS LICENSORS, WHETHER EXPRESSED OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT.
(d) SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY. IN THAT EVENT, ANY IMPLIED WARRANTIES ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY OF THE SOFTWARE. THIS WARRANTY GIVES THE END USER SPECIFIC LEGAL RIGHTS. THE END USER MAY HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

8. LIMITATION OF LIABILITY**.** UNDER NO CIRCUMSTANCES SHALL NEOLOGIC BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES, EVEN IF NEOLOGIC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

In no event shall NeoLogic's total liability to Licensee for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by Licensee for the Software.

9. BREACH AND TERMINATION**.**
(a) This License is effective until terminated. This License may be terminated by the non-defaulting party if either party materially fails to perform or comply with this License or any provision hereof.
(b) Termination due to a breach of Section 6 shall be effective upon notice. In all other cases termination shall

be effective thirty (30) days after notice of termination to the defaulting party if the defaults have not been cured within such thirty (30) day period. The rights and remedies of the parties provided herein shall not be exclusive and are in addition to any other rights and remedies provided by law or this Agreement.
(c) Upon termination of this Agreement, all rights and licenses granted hereunder shall immediately terminate and all Software and other Proprietary Information of NeoLogic in the possession of Licensee or under its control, shall be immediately returned to NeoLogic. End user licenses properly granted pursuant to this Agreement and prior to termination of this Agreement shall not be diminished or abridged by the termination of this Agreement.

10. GOVERNING LAW**.** The rights and obligations under this Agreement shall be governed by the laws of the State of California excluding its conflicts of law rules and United States law and international treaties governing copyrights.

11. EXPORT LAW ASSURANCES**.** Licensee agrees and certifies that neither the Software nor any other technical data received from NeoLogic, not any direct product thereof, will be exported outside the United States, except as permitted by the laws and regulations of the United States.

12. GOVERNMENT END USERS**.**
(a) If this Software is acquired by or on behalf of a unit or agency of the United States Government through Licensee, Licensee hereby undertakes to insert in its contract with the Government the appropriate restricted rights language.
(b) If this Software is acquired by or on behalf of a unit or agency of the United States Government directly from NeoLogic, this provision applies. This Software: (i) was developed at private expense, and no part of it was developed with government funds; (ii) is a trade secret of NeoLogic for all purposes of the Freedom of Information Act; (iii) is "commercial computer software" subject to limited utilization as provided in the contract between vendor and the government entity; and (iv) in all respects is proprietary data belonging solely to NeoLogic.
(c) For units of the Department of Defense (DOD), this Software is sold only with "Restricted Rights" as that term is defined in the DOD Supplement to the Federal Acquisition Regulations ("DFARS") 52.227-7013 (c)(1)(ii) and use, duplication or disclosure is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 52.227-7013 and in similar clauses in the NASA FAR supplement. Manufacturer: NeoLogic Systems, Inc., 1450 4th Street, Suite 12, Berkeley, California 94710.

13. MISCELLANEOUS. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect. If either NeoLogic or Licensee employs attorneys to enforce any rights arising out of or relating to this Agreement, the prevailing party shall be entitled to recover its reasonable attorneys' fees, costs and other expenses. This License constitutes the entire agreement between the parties with respect to its subject matter, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of NeoLogic. This Agreement, and any rights or obligations hereunder, shall not be assigned or sublicensed by Licensee without the prior written approval of NeoLogic. Any such attempted assignment shall be void.

14. COMPLETE AGREEMENT: This License constitutes the entire agreement between the parties with respect to its subject matter, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of NeoLogic.

Any questions concerning this License and Limited Warranty should be addressed to the:

NeoLogic Systems, Inc.
1450 4th Street, Suite 12
Berkeley, California 94710

Table of Contents

Welcome

Introduction

Thank you for taking the time to review NeoAccess, the cross-platform object-oriented database engine. Applications based on NeoAccess store and retrieve even the most complex application-specific objects and data quickly and easily. NeoAccess has the features and performance you need to efficiently build powerful commercial and in-house applications.

The programming interface to NeoAccess is designed to keep visible complexity to a minimum while providing a feature-rich foundation on which to build and enhance applications. Object-oriented developers are most productive when dealing with classes and objects, not black-box procedural libraries, which most databases are. NeoAccess allows developers to access database capabilities by subclassing and function invocation.

Application frameworks include classes that you can use to build the user-interface portion of your application. NeoAccess is a set of C++ classes that extends these frameworks to facilitate the development of an application's data model, or back-end. Developers subclass and instantiate NeoAccess classes to implement those objects that need to persist across session boundaries — that time between when the user quits your application at night and starts it up again eight hours later smelling of coffee and Corn Flakes.

This manual presents you with a preliminary introduction to the capabilities of NeoAccess. Each topic discusses issues of interest. Chances are that you've probably heard the terms "object-oriented" and "database" at least five times if you've been involved in software development for more than ten minutes. While we assume you have a working knowledge of these terms, this document explains how NeoLogic has fused these two ideas into a very powerful development tool. NeoAccess is different from other database engines, object-oriented or otherwise. You can refer to this document in order to learn how NeoLogic has addressed these issues.

The Development Experience

Who's Who

A typical development team on a project using NeoAccess-oriented project is often staffed with individuals or groups that have specific responsibilities. It depends on the project, but when two or more people are involved, the project will usually have a "front-end" group and a "back-end" group. The front-end group, which we usually refer to as application developers, deal with user interface issues. The back-end group, or database developers, work on issues having to do with how persistent application objects are organized, stored and accessed.

A great deal of effort has been put into NeoAccess to make the developer experience of both of these groups as productive and enjoyable as possible. But it is particularly important that complexity and logistics be hidden from application developers. This not only allows them to remain productive, but also allows them to design and implement a front-end that is decoupled from the specifics of how objects are defined, organized and accessed in the back-end.

Design Patterns

Experienced object-oriented software designers recognize that, in a broad sense, all systems share a common set of problems. They have also found that the general set of solutions used to solve these problems are very similar, even when the design objectives, programming languages or development tools differ. Designers use the term design patterns to describe the commonalities found in the problem and solution sets of these

systems. A **design pattern** describes a problem which occurs repeatedly in different situations yet can be solved by viewing the problem as a general set of relationships, responsibilities and collaborations. The benefit to grouping general problems and their solutions in design patterns and, further, by understanding the relationships that exist between patterns, is that the solutions to new problems can then be stated in terms of these well understood patterns. It is important that a designer understand the motivation for, applicability of and consequences of a design pattern in order to use it effectively. Importantly, the name of a pattern should communicate the general problem it addresses and the perspective taken in solving it.

It is sometimes convenient to view the features provided by a database management system as consisting of three layers of functionality;

- Persistence properties
- Organizational constructs
- A high-speed search engine

These three layers are design patterns. The persistence pattern addresses the need to make data persist over time, even when the computer or application which manages the data is not running. The organization pattern is used to manage relationships that exist between persistent objects. The search pattern uses the organizational constructs to obtain fast access to the objects of interest even when the overall data set is huge.

It is useful to consider these three main design patterns and the relationships between them as you explore the features and structure of NeoAccess.

ODBMS and RDMBS Differences

Object-oriented program developers are most effective when working with classes and objects. NeoAccess takes an approach to database technology that eliminates the paradigm shift that often plagues more traditional relational database systems. However, in adopting NeoAccess, some developers with previous relational database systems experience may need some help in relating their previous experiences to the NeoAccess approach. The following definitions are included for those developers who are trying to make the transition from older, purely relational technology to NeoAccess.

Data Dictionary

In relational databases, a **data dictionary** is a description of the structure of a database. This definition can include the definition of data types, record layouts, and indexing options. Many database management systems use an internal representation of this dictionary, called a **schema**, to process database queries and updates.

The creation of a data dictionary is a database administration task begun during the initial phases of the system design process. Yet data dictionaries typically evolve dramatically during a system's lifespan. The evolution of a database dictionary is called **schema evolution**.

NeoAccess obtains much of the information that is typically contained in a data dictionary, such as class definitions, from the compiler itself. Instead of asking developers to create and maintain a data dictionary, NeoAccess uses metaclass objects to obtain information about persistent classes in the current application. The metaclass table maintained by NeoAccess contains class IDs, a list of classes that refer to the ancestors of each class, the class names, the number of index keys maintained by each class, and other pertinent information.

Tables and Records

Traditional database management systems store data in tables. A **table** is the collection of records in a database that have the same type. Indeed, the definition of a table in a schema defines the layout of the record in the database.

The definition of a NeoAccess-based persistent class is similar to a table definition, except that NeoAccess knows how classes are related to one another. This allows NeoAccess to perform deep searches on a class and its subclasses.

Individual object instances of a class are similar to records in a table. But while both records and objects contain state information, object also contain the intelligence to manipulate that state.

Query

A **query** is an abstract specification of the set of objects in a database. The relational query language supported by most traditional database systems is SQL.

The search mechanisms provided by NeoAccess use a flexible selection mechanism based on objects having a base class of CNeoSelect. Subclasses of CNeoSelect can be designed to create selection criteria objects with tremendous power. CNeoSelect is a type of **selection criterion** (also called **select key**) that is used to locate objects. Locating objects using a specific selection criterion involves instantiating an object which is a subclass of CNeoSelect and passing that object to the database's `findObject` function or the constructor of an iterator.

Cursor

A **cursor** is a construct used to obtain query results serially one record or object at a time. Cursors are most useful when the data set of a query is too large to manage effectively in memory at once.

NeoAccess includes a set of iterator classes for iterating over a set of objects that match a given selection criterion. Iterators greatly simplify the management of large sets of objects by application developers.

There are several iterator classes that come standard with NeoAccess. They are often called **keyed iterators** because of their unique ability to iterate over a subset of a collection based on an abstract select key. The set of operations supported by iterators include the ability to traverse the collection forward and backward, the ability to test whether there are more items in the collection beyond the current item and the ability to reset the iterator to the beginning again. Some subclasses of CNeoIterator can even iterate over all matching objects of a particular base class and all subclasses, or iterate over all matching items in a parts list.

NeoAccess Synergy

A car isn't a car without a minimum set of parts: wheels, a drivetrain and some way to control it. These base components work synergistically to build a higher level abstraction - a means of transportation. (Of course any car salesman will tell you that cars are much more than simply a means of transportation. They're fast, comfortable and good looking. In short, they've become status symbols.)

In much the same way, most mature application frameworks are structured as a network of subcomponents that each contribute to the synergistic whole. Major subcomponents might include event handling, document management, geometry support and, of course, views. Zooming in even further, each of these subcomponents might be further dissected.

NeoAccess is itself a collection of lightweight layered abstractions. At a base level there are simply databases and persistent objects. This is the rich soil within which higher level abstractions are rooted. At the highest level is a full featured object database. But the truly unique power of NeoAccess is that the mid and upper layer abstractions are completely accessible to developers to exploit and extend. It is this accessibility and extensibility that is the most important advantage object systems provide over procedural systems.

Naming Conventions

In order to enhance the readability of the source code and avoid naming conflicts with system software and your application code, all NeoAccess source code and header files adhere as closely as possible to the following set of naming conventions:

- Instance (non-static) member names begin in lower case.
- Class (static) member names begin in upper case.
- Instance (non-static) data member names begin with "`f`".
- Class (static) data member names begin with "`F`".

- Global variable names begin with “gNeo”.
- Parameter variable names begin with “a”.
- Constant names begin with “kNeo”.
- Type names begin with “kNeo” and end with “Type”.
- Most class names begin with “CNeo”.
- The names of embedded classes begin with “ENeo”.
- The names of template classes begin with “TNeo”.
- Property names, also called tag names, begin with “pNeo”.
- Conditional compile symbols begin with “qNeo”.

Typographic Conventions

The following typographic conventions are followed throughout this manual:

- Source code examples and the names of procedures, variables and constants are all set using Courier type.
- Important technical terms are set using **bold** type in defining sentences or first usage.
- Optional class, argument and variable names are set using *italic* type.
- SMALL CAPS style is sometimes used for emphasis.

Technical Overview

Introduction

This section contains basic explanatory information concerning NeoAccess, including basic terminology and the methodologies you can use to develop database applications with the product. Detailed information concerning each of the NeoAccess classes and member functions is provided later. For now, the intention is to arm you with the information you will need to proceed with your development efforts.

The Database

Computer data are usually stored in physical files. The operating system usually represents a file as a single stream of bytes. Typically the data stream is read in and written out serially, from beginning to end. However, many file systems also provide a mechanism for "seeking" to a particular location in the file and reading or writing data at that location.

Macintosh files consist of a data fork and a resource fork. The data fork is the same as the byte stream found on other systems. Resources are chunks of data that are identified by a 4-byte resource type and either a resource ID or a resource name. The Macintosh Resource Manager provides random access to resources. The data fork, on the other hand, contains a single stream of bytes. Other operating systems that don't support a resource fork may still have resource files that provide capabilities similar to Macintosh resources. But both mechanisms have their limitations. The data fork is without structure. The resource fork has structure but the mechanism that provides that structure, the native resource manager, is very inefficient when the number of resources begins to grow.

CNeoDatabase

NeoAccess includes a class called **CNeoDatabase**. CNeoDatabase stores objects in a **container**. A container is a repository which contains a NeoAccess database. In most cases, NeoAccess uses a file's data fork as its container. But other types of containers might also be used; OpenDoc or OLE containers are examples of this.

Objects are the basic elements of object-oriented applications. Think of them as intelligent data. The CNeoDatabase class organizes objects the way the application references them, instead of just by a resource ID or name. Fortunately, CNeoDatabase is also very efficient. It doesn't slow down when dealing with a large number of objects the way resource managers do.

Creating and Opening the Database

CNeoDatabase is a class of object. Just like any other object, an instance of this class is created by using the `new` operator. (The `new` operator is replaced by the `NeoNew` macro to provide additional debugging support.) To access objects contained in a CNeoDatabase, it must be open. However, before it can be opened, a path name must be specified. This path is the location of the database file in the file system.

Many application frameworks place the responsibility for creating the main data file for an application in an environment-specific document-derived class. Your application should derive a class from the application framework's document class to further specialize its member functions. This class will always be derived from the CNeoDoc class, which is an environment-neutral part of the NeoAccess class hierarchy.

In the following code snippet, the CNeoDocPP constructor calls a number of member functions specific to Metrowerk's PowerPlant application framework, and then it creates a new instance of the CNeoDatabaseNative class. This action creates the database object, but does not create its associated file in the file system.

```
void CNeoDocPP::CNeoDocPP(const OSType aCreator, const OSType aType,
      const Boolean aPrintable, const Boolean aNewDatabase,
      const Boolean aCreateDatabase)
{
 LGrowZone *growZone;

 NeoUsed(aPrintable);

 SetSuperCommander(gNeoApp);
 SetSuperModel(gNeoApp);

 fNewDatabase = aNewDatabase;
 mFile = NeoNew CNeoDatabaseNative(aCreator, aType);
 gNeoDatabase = ((CNeoDatabaseNative *)mFile);

 growZone = LGrowZone::GetGrowZone();
 if (growZone)
      growZone->AddListener(this);

 fOpenMode = NeoReadWritePerm;
}
```

Creation of the CNeoDatabase-derived object is similar for MFC-based applications. For example, when the user chooses the **New** command in an MFC-based application, the document object is created automatically by the framework when its `OnNewDocument` message handler is called. The `OnNewDocument` handler in the CNeoDocMFC-derived class would, in turn, call the `OnNewDocument` function in the CNeoDocMFC class. That code is as follows:

```
BOOL CNeoDocMFC::OnNewDocument()
{
 Boolean result = CDocument::OnNewDocument();

 NeoAssert(!fDatabase);
 fDatabase = new CNeoDatabaseNative;
 CNeoDatabase::SetCurrentDatabase(fDatabase);
 SetModifiedFlag(FALSE);

 return result;
}
```

The foregoing code for the `OnNewDocument` function uses the `new` operator to create an instance of the CNeoDatabase class. The `SetCurrentDatabase` member function sets the document's database (`fDatabase`) to be the currently active database and causes the `fDatabase` pointer to be stored into the `gNeoDatabase` global variable.

Once the database has been created, objects can be added, deleted or changed. But these operations will take place only in the state of the database stored in memory. In order to save these changes to a permanent file, the database must be committed.

Committing Changes to the Database

When the contents of a database change — objects have been added, deleted or changed — these changes occur only in memory. The state of the database on disk is not affected. Changes only become permanent when the on-disk state of the database is synchronized with its in-memory state. The **database commit** process involves writing the state of "dirty" objects in memory out to disk.

As was the case for creation of the database, saving its state is accomplished by code that is largely unique for each development environment. In the case of PowerPlant, committing changes to the database file is

handled by the DoAESave member function of the CNeoDocPP class. The code for this function is as follows:

```
void CNeoDocPP::DoAESave(FSSpec &aSpec, OSType aType)
{
 CNeoDatabaseNative *  database= getDatabase();
 CNeoDBFocus           dbFocus(database);

 NeoUsed(aType);

 // Specify the location of the database in the file system.
 database->Specify(&aSpec);
 mIsSpecified = TRUE;

 // Create the database file.
 database->create();

 // Open the database file.
 database->open(fsRdWrPerm);

 // If this document has a window, update its title.
 if (mWindow)
     mWindow->SetDescriptor(aSpec.name);

 // Write out all objects contained in this database.
 database->commit(TRUE);

 // Note that document is no longer dirty.
 setDirty(FALSE);
}
```

The foregoing code assumes that the file in which the data are to be saved has already been selected, and proceeds to call the Specify function of CNeoDatabase to create the CNeoContainerStream object into which the data are to be written. (See the discussion of Object I/O for more details on streams) Following this, the database file is created and opened. If a window is open on the document, its title is changed to the name of the database file. The call to the commit function causes the list of database objects in memory to be scanned, writing out only those that have been changed during the course of the application's execution.

When developing an application using MFC, saving the changed data in the database is accomplished with the OnSaveDocument function of the CNeoDocMFC class. The code for that function is as follows:

```
BOOL CNeoDocMFC::OnSaveDocument(const char *aPathName)
{
 if (fDatabase) {
     SetPathName(aPathName, fAddToMRU);
     if(!fDatabase->isOpen() || fDatabase->fNewStream) {
         fDatabase->create();
         fDatabase->open(NeoReadWritePerm);
     }
     fDatabase->commit(TRUE);
     if (!fDatabase->isDirty()) {
         SetModifiedFlag(FALSE);
         return TRUE;
     }
 }

 return FALSE;
}
```

The foregoing code performs many of the same actions as the corresponding code for the PowerPlant framework. If the database file is not yet open, then the `create` and `open` member functions inherited from CNeoDatabase are called. Then the changed data are written to the database file by virtue of the call to the `commit` function. If the operations to write out the data are successful, the `isDirty` function will return `FALSE`, the document will be marked as unmodified, and the function will return `TRUE`; otherwise, `FALSE` is returned, indicating that the "save" operation was not successful.

Closing the Database

A database object needs to be closed before the application terminates. If any objects have been added or changed, then those changes need to be committed before the database is closed.

In the case of the PowerPlant framework, the database file is closed when the `close` function of the CNeoDatabase class is called.

Key Point

When the `close` function is called, any uncommitted changes will be lost. The database file is physically closed and the state of the database on disk is left unchanged. It is therefore very important that the database's `commit` function be called before closing the database if changes are to be saved; otherwise, the changes will be lost.

When using MFC, closing the database takes place just before the document object is disposed of. This can occur as the result of closing the window in which the document's data is displayed or by the user's choice of the **Close** menu command. In either of these cases (and others), the `OnCloseDocument` function, which is overridden in the CNeoDocMFC class, is called to perform the close operation. That code is as follows:

```
void CNeoDocMFC::OnCloseDocument()
{
    BOOL bAutoDelete = m_bAutoDelete;

    m_bAutoDelete = FALSE;  // don't destroy document while closing views
    CDocument::OnCloseDocument();

    if (fDatabase) {
        if (fDatabase->isOpen())
            fDatabase->close();
        delete fDatabase;
        if (gNeoDatabase == fDatabase)
            gNeoDatabase = nil;
        fDatabase = nil;
    }

    m_bAutoDelete = bAutoDelete;

    // delete the document if necessary
    if (m_bAutoDelete)
        delete this;
}
```

As is evident from the foregoing code, after the `OnCloseDocument` function of the CDocument base class is called, the `close` function is called for the database object, if that object exists and is open. If the `bAutoDelete` flag is set for this document, then the document object is deleted.

The Object

As the name implies, object-oriented systems deal primarily with objects. Objects are pieces of intelligent data. The state of an object consists not only of data values, but also of the set of operations that are defined

for that data.

For example, an instance of the CNeoDatabase class is an object. This class of object contains state information: it may refer to an operating system file, it has a length and an object count. But the real value of a CNeoDatabase is that it performs actions for you without your having to know how it does them: for example, your application asks the database object to commit to disk the changes that have been made in memory. Note that you don't need to know the details of how this update process is performed. You just need to know how to ask the database to perform the action. Object classes centralize and isolate intelligence so that visible complexity is minimized.

Application-Specific Objects

Application-specific objects encapsulate the intelligence of your application. They are the value that you add to the user experience. The raison d'être of your application is to provide a mechanism that allows users to manipulate these objects.

Some application-specific objects are persistent objects. Users create something that they can come back to and work with again later. In order for these objects to persist, your application needs to include a mechanism that preserves the state of these objects after your application has quit, and which can be used to locate the objects again later.

It is important to remember that windows and the other components that make up the user interface to your application are not the objects that need to persist over time. Visual objects disappear when the user quits the application. These persistent objects are sometimes called model objects.

Historically, most applications that support a document architecture do so by using a fairly rudimentary mechanism called a stream. Streaming the entire contents of a file back and forth between the document and memory is sometimes called the inhale/exhale approach to object persistence. Using this approach, a persistent object must be in memory while the application is running. When the user chooses the Save… menu item, the application opens the document file and serially writes every persistent object in memory out to disk. The process of reading a document involves reopening the file and reading its entire contents back into memory.

Most database systems provide application builders with an API for reading and writing data from database tables. But the information returned by a database query is usually just a data record, not an object. Object-oriented developers need to write "wrapper routines" to copy the individual fields of a record into the data members of the application-specific objects. Developers using non-object database systems are forced to handle other bothersome logistics, like keeping track of which objects have changed: changed objects need to be kept together so that the database can be updated to reflect those changes, new objects need to be kept somewhere else so that they can be added to the database, and references to shared objects need to be tracked closely so that the objects stay consistent. Then updates necessitate that another set of wrappers be written to copy data back into the database record format for writing back in the database.

NeoAccess frees developers from worrying about these details. The approach taken by NeoAccess is that objects should be viewed as having a set of properties and a malleable state. Just as view objects have properties that allow them to be drawn on a screen or printer relative to other objects, persistent objects are provided with persistence and sharing properties by NeoAccess. These properties allow objects to maintain an association with a database. This association, which can be easily built and broken, allows objects to migrate freely between disk and memory. An object's interface to these properties deals with issues such as making and breaking the object-database connection (adding or deleting the object from a database), locating and later freeing the object in memory, object sharing, and maintaining relationships between itself and other objects in a database.

NeoAccess includes a class of object called **CNeoPersist**, on which all persistent objects are based. Built into this base class is intelligence about managing the permanence of objects. (The **leaf class** of an object is its most specific class. The leaf class and all super classes of an object are referred to as the object's **base classes**.) It's easy to define application-specific classes based on CNeoPersist, because most of the complexity involved in adding, deleting and locating objects in a CNeoDatabase has been encapsulated in the base class. All you need to add is the intelligence that makes the object useful in your application. In essence, persistence comes free, or at least at a very low cost.

NeoAccess maintains a distinction between persistent objects and permanent objects. A **persistent object** is any object that can be permanent. A **permanent object** is one which has been added to a database. For example, an application may create a new persistent object. However, if the application quits without adding this object to a database and then commits the change, then the state of the object is not permanent. So, while all permanent objects are persistent, not all persistent objects are permanent.

Key Point

As noted, persistent objects (CNeoPersist and its subclasses), by their very nature, migrate freely between the database and memory. As such, these objects can not be local variables on the stack nor can they be embedded data members of other objects.

The definition of application-specific classes are usually found in header files while the implementation of member functions are found in source files. A simplified class definition is as follows:

```
const NeoID kPersonID = 25;

const NeoTag   pNeoName     = 0x6e616d65;         /* ASCII - ‘name’ */
const NeoTag   pFather      = 0x66617468;         /* ASCII - 'fath' */
const NeoTag   pAddress     = 0x41647273;         /* ASCII - 'Adrs' */
const NeoTag   pIQ          = 0x49512020;         /* ASCII - 'IQ  ' */

class CPerson : public Compartment
{
public:
                  CPerson (const CNeoString &aName = “”,
                       const CNeoString &aFatherName = “”,
                       const CNeoString &anAddress = “”,
                       const unsigned short aIQ = 100);
 NeoID            getClassID(void) const {return kPersonID;}
 static CNeoPersist *
                  New(void) const {return NeoNew CPerson();}
 Noises           getFileLength(const CNeoFormat *aFormat) const;
 void             readObject(CNeoStream *aStream, const NeoTag aTag);
 void             writeObject(CNeoStream *aStream, const NeoTag aTag);

protected:
 CNeoString       fName;
 TNeoIDSwizzler   fFather;
 CNeoString       fAddress;
 unsigned short   fIQ;
};
```

There would be many more member functions and variable definitions in a typical class definition; however, the proceeding class definition illustrates some fundamental points. Each persistent class has an associated class ID which is unique to that class. You must define this value explicitly, as was done with the `kPersonID` constant. Notice in the foregoing definition that, in addition to the constructor, the `getFileLength`, `readObject`, and `writeObject` member functions have been overridden. All classes for which persistent data is to be stored must implement these functions. See the discussion titled “Subclassing CNeoPersist” for a complete list of member functions which need to be overridden.

Developers sometimes confuse class IDs and object IDs. The **class ID** of a persistent class must be unique among all other persistent classes of an application. The class ID should be explicitly defined in the header file containing the definition of the class. All class IDs below 20 (including negative class IDs) are reserved for NeoAccess internal classes. When NeoAccess’s dynamic object facility is enabled, all class IDs greater than or equal to `kNeoFirstPrototypeClassID` are reserved for dynamically defined classes. (See the Dynamic MetaObject Facility documentation in the NeoAccess Extras folder for more details.) The class IDs of all statically defined persistent classes should therefore be between 20 and `kNeoFirstPrototypeClassID`.

Every persistent object has an object ID. An **object ID** can be explicitly set before the object is added to the database. Alternatively, you add the an object with an object ID of zero to the database, which indicates to NeoAccess that it should assign it an object ID which is unique among all other objects in that database which were implicitly given object IDs. The ID of objects are usually unique, though they needn't be.

Tags

It is sometimes useful to be able to refer to a data member of a persistent object abstractly. Just as all persistent objects have an object ID, data members of persistent objects can be assigned tag values. A **tag** is a unique four-byte constant that can be used to symbolically refer to a data member of an object. The `getValue` and `setValue` functions of CNeoPersist and its subclasses can be used to get and set data members values using a tag. Tags are also use in the creation of relational queries and indices.

Creating an Object

Creating a persistent object is no different from creating any other type of object; use the `new` operator.

Key Point

In some cases, the `NeoNew` macro is used instead. This provides additional debugging support when objects are created. `NeoNew` is a macro which usually equates to `new`. But if the compile-time constant `qNeoDebugMemory` is defined, then this macro does the right thing to support core leak detection.

Sharing an Object

The CNeoPersist class provides a sharing property. Any persistent object can be shared using this facility whether it is permanent or not. A persistent object remembers how many references there are to it. When an object is brought into memory, either by creating it with the `new` operator, getting a reference to it from another component, or locating a pre-existing object from a database, the number of references to the object is increased. The object stays in memory until the last reference is disposed of.

Suppose your application is a personal productivity suite that includes calendar and address book functions. A user may have the calendar and address book open at the same time and both of these components may refer to a common persistent person object. Without a sharing property built into the person class, the calendar component of your application might delete the person from memory not knowing that the address book component still refers to it. One way to avoid this difficulty might be to have each component maintain a separate copy of the object in memory, but that brings up the potential for each of the copies to be slightly different.

References can be added to an object by using the `referTo` function of the object. (A reference is implicitly added to an object when it is obtained from a database using its `findObject` function or the FindByX static functions of CNeoPersist and its subclasses. References are NOT however added when an object is obtained from an iterator.) When your application has finished referring to it, it should call `unrefer` to remove the reference.

Key Point

There is an easy way to remember which NeoAccess member functions add references to objects and which don't. The static `FindByX` member functions all add a reference to the return value, if any, as does the assignment operators of swizzlers. All other functions do not add a reference.

There are many different ways for an application to manage object concurrency. (For information on how NeoAccess can more actively manage object reference counting for you, see the discussion of swizzlers below.) Consider the sample routines below:

```
void Masseuse(CNeoDatabase *aDatabase)
{
 for (long index = 0; index < 100; index++)
      MassageObject(aDatabase, index);
}

void MassageObject(CNeoDatabase *aDatabase, const NeoID aID)
{
 CNeoAppSpecific *     object;

 object = CNeoPersist::FindByID(aDatabase, kAppSpecificID, aID, FALSE);

 // Do a bunch of stuff to the object.
 // ...

 // Call unrefer to remove the reference we obtained from FindByID.
 object->unrefer();
}
```

The function `MassageObject` massages the single object that is returned by `FindByID`. If the last argument of `FindByID` were non-`nil`, then a reference would have been added to each object in the array. All of these references need to be disposed of properly.

Consider what would happen if an exception were to occur in `MassageObject` between the point where `object` was returned from the database and where the reference to `object` was removed. The object would never be deleted because it would have one more reference than it should have.

There are several possible solutions to this dilemma. The body of `MassageObject` could be enclosed in a `NEOTRY` block with a `NEOCATCH` block that removes the reference.

```
void MassageObject2(CNeoDatabase *aDatabase, const NeoID aID)
{
 CNeoAppSpecific *     object   = nil;

 NEOTRY {
      object = CNeoPersist::FindByID(aDatabase, kAppSpecificID, aID, FALSE);

      // Do a bunch of stuff to the object.
      // ...

      // Remove the reference we obtained from FindByID.
      object->unrefer();
      object = nil;
 }
 NEOCATCH {
      if (object)
           object->unrefer();
 }
 NEOENDTRY;
}
```

The overhead of `NEOTRY` blocks is relatively high, both in terms of code space and execution time. For this reason `MassageObject2` is less than optimal. NeoAccess includes a construct called a swizzler which manages reference counting automatically. Refer to the “Swizzlers” discussion later in this section for more details.

Object Concurrency and Referential Integrity

Concurrency and referential integrity are two of the more difficult issues for class designers to solve in a

general fashion. While recognizing conflicts is relatively straightforward, resolving them and avoiding deadlocks is not.

The CNeoPersist class has a property that allows developers to indicate whether an object is busy (*i.e.*, in an inconsistent state). An object's busy state is kept consistent using the same mechanism that keeps object references up to date even if there is a failure. See the topic "Sharing an Object" immediately above for more information.

An object could be marked busy and unbusy by using the `setBusy` and `setUnbusy` member functions respectively, However the use of these functions are discouraged because an exception thrown before the `setUnbusy` call would result in an object's busy state being left with an improper value.

```
void ChangeObject(CAppSpecific *aObject)
{

 // Mark the object busy so that others realize that it may be inconsistent.
 aObject->setBusy();

 // Call a routine that changes the state of the object in some round-about way.
 ThrashObject(aObject);

 // Now that it is once again consistent, mark the object as no longer busy.
 object->setUnbusy();
}
```

CNeoBusyFocus objects are used to manage the busy state of an object consistent in the event of a failure. A busy focus object records the busy state of an object before setting the object busy. The destructor of the focus, which is called even if a C++ exception is thrown, resets the object's busy state.

```
void ChangeObject(CAppSpecific *aObject)
{
 CNeoBusyFocus(aObject);

 // Call a routine that changes the state of the object in some round-about way.
 ThrashObject(aObject);
}
```

Adding an Object to the Database

At some point during the execution of your application, you will decide that an object needs to be made permanent. For example, in the personal productivity application mentioned earlier, imagine that the user has added a new person to the address book. When the time comes to add the person object to the database, your application will call the database's `addObject` function. An example of this is as follows:

```
CPerson *CMyDocument::CreateNewPerson(CNeoString aName,
      CNeoString aFather, CNeoString anAddress, unsigned short anIQ)
{
 CPerson* pPerson = NeoNew CPerson(aName, aFather, anAddress, anIQ);

 gNeoDatabase->addObject(pPerson);

 return pPerson;
}
```

When the new object is constructed, the name, father's name, address and IQ arguments to the `CreateNewPerson` function provide the means to create a complete CPerson object, as specified earlier. The call to the `addObject` function adds the object to the database.

Changing an Object

A distinction can be made between permanent data members of an object and transitory ones. **Permanent members** are those that make up the permanent state of an object. A CPerson object's address and phone number are permanent. **Transitory members** are generally used for housekeeping tasks while the object is in memory. Pointers, reference counts and the like are usually transitory values.

A property of permanent objects is their ability to track when the value of a permanent data member is changed. This allows them to update their state on disk to match the modified state in memory when changes are committed. Member functions which modify these permanent data members should use the `setDirty` function to mark the object as having been modified.

Key Point

Changes to persistent objects occur only in memory. It is only when the database is committed that all changes to dirty objects are written to disk.

```
void CAppSpecific::setPermValue(const long aValue)
{
 fPerm = aValue;
 setDirty();
}
```

Removing an Object

Inevitably, your application will need to remove objects from a database. The database's `removeObject` function does this. An object continues to exist in memory after it has been removed. It can be manipulated just like any other object. It can even be re-inserted in the same or any other database at some later point. The `removeObject` function frees the file space taken up by the object and removes it from its indices.

Deleting an Object

A persistent object is deleted from memory by using the `unrefer` function.

But `unrefer`'ing an object does not always result in the object being deleted from memory. Some other part of the application may still refer to the object in memory. NeoAccess insures that objects are deallocated from memory only after all references have been removed (by calling `unrefer`).

But there is yet another reason why objects may remain in memory even after all the application's references to it have been deleted. NeoAccess includes a very sophisticated object cache, whose purpose is to improve object access times by minimizing disk activity. NeoAccess may decide to keep an object in its cache in case the application tries to access it again. Locating the object the next time will be fast. The cache is purged when application memory is low, so that the object cache will never cause your application to run out of memory. Caching can improve access times by as much as 20 times in some situations. (Your application's mileage may vary.)

Organizing Objects in the Database

To date, object database usage has not grown at a rate comparable to that of traditional relational systems. One of the major reasons for this is that most modern database applications are designed with the assumption that data can be retrieved using relational queries (which are also called associative lookups). While first-generation object databases execute referential queries (also called parts explosions) extremely well, most do not even support relational queries *per se*.

NeoAccess is different. While NeoAccess provides very powerful referential query mechanisms, the most prevalent way to locate objects is via its relational query mechanisms.

Indexing

Objects in a NeoAccess database are organized primarily by class. This is analogous to the way that relational systems store records in tables. For example, all CCircle objects in a graphics application are grouped together, as would be all CSquares. NeoAccess also knows how classes are related to one another. It knows, for example, that the CCircle and CSquare classes have a common parent class, CShape. Knowing the genealogy of classes allows the query mechanism to be much more powerful than a system which does not support inheritance. By performing a single database query, the screen update member function of our mythical graphics application can locate all objects having a base class of CShape located in a particular update region.

All objects of a particular class are grouped together using one or more indices. By default, the primary index of each class sorts all objects in ascending order by object ID. Additional secondary indices might also be used to sort objects of a class based on other data values. The indexing defined for objects of a class in a database will depend on the application's requirements.

You might, for example, own a shoe store and manage your inventory in a NeoAccess database. All CShoe objects in the database might be sorted using one or more secondary indices in addition to the primary index – by size, by color and by manufacturer. The indexing defined for objects in a database will affect the speed with which objects are found during a database query. A search for shoes of a particular size will be much faster if shoes are sorted by size.

Indices can be added and removed from a class of objects on a per database basis at runtime. Two inventory databases, which are open at the same time in the proceeding shoe store example, might sort shoe objects differently based on the current needs of the application or the whims of the developer.

The process of designing an indexing strategy for an application involves balancing the costs and benefits of binary searches using indices and serial traverses of a list. However, except for access times, the part of an application that is trying to locate objects needn't be aware of whether a search is performed linearly or serially.

Swizzlers

While NeoAccess supports associative lookups, one of the biggest advantages it has over traditional relational systems is the ease with which it manages direct references to other permanent objects. A **swizzler** is an object which can be used to refer to another object in a database. A swizzler can be thought of as a "smart pointer." It is an object which looks and acts like a pointer, but is more powerful than a standard C pointer in that it can be used to refer to permanent objects that might not even be in memory yet. A swizzler data member of a persistent object can be used to maintain a one-to-one relationship between itself and the object referred to by the swizzler.

Using swizzlers instead of object pointers removes the need for application developers to explicitly add and remove object references to persistent objects. This is because the assignment operator of a swizzler removes its reference to the object it was referring to and adds a reference to the object being assigned to it. The swizzler destructor, which is even called when a C++ exception is thrown, also removes references as the swizzler goes out of scope.

The snippet below illustrates the use of a swizzler to manage reference counting of objects returned by an interator. Note that reference counting of objects returned by the iterator is handled automatically by the swizzler.

Like a swizzler, a tracker is a "smart pointer" which is used to refer to another object. However, trackers are not used to refer to reference counted objects. Instead, they refer to transitory objects which are to be deleted when the tracker goes out of scope. The snippet below uses a TNeoTracker object to refer to the iterator object returned by the database's `getIterator` function. This iterator is deleted when the local tracker variable's destructor is called. Not only do tracker objects handle cleanup issues, a tracker's destructor is called even when a C++ exception is thrown. Trackers can therefore be used to avoid having to set up catch blocks.

```
TNeoSwizzler<CPerson>      person;
TNeoTracker<CNeoIterator>  database->getIterator(kPersonID, nil, TRUE);

person = (CPerson *)iterator->currentObject();
while (person) {
     person->printName();
     person->skill();
     neoPrintf(NeoEOL);
     person = (CPerson *)iterator->nextObject();
}
```

Part Lists

While swizzlers are used to manage one-to-one relationships between one object and another, a **part list** is a sorted or random ordered collection of references to other permanent objects. A part list data member of a persistent object can be used to maintain a one-to-many relationship between itself and the objects referred to in the part list. Accessing objects via a part list or swizzler is sometimes called **referential access**.

Searching for Objects in the Database

A database search involves the selection of a set of objects which match a given selection criterion. Ultimately, the true value of a database is its ability to locate objects of immediate interest quickly and easily. The easy-to-use interface to NeoAccess's search mechanism balances simplicity with power with an extensible architecture.

Selection Criteria

The NeoAccess search mechanisms use a flexible selection mechanism based on objects having a base class of CNeoSelect. To support the ability to locate objects using a specific kind of selection criterion, developers simply configure selection objects to indicate those objects of interest in a collection.

CNeoTypeSelect is an abstract base class from which all type-specific selection classes are derived. The various subclasses of CNeoTypeSelect are used to compare some persistent object data member with the value of the select key. For example, CNeoLongSelect compares the long value of the key with the value of an object's data member. The type of comparison performed is defined by the CNeoTypeSelect subclass. That is, CNeoLongSelect compares long integers while CNeoStringSelect is used to compare two C strings. The **select tag** of the CNeoTypeSelect object defines which object attribute is used in the comparison. CNeoTypeSelect subclasses' `compare` function obtains the attribute value by using the object's `getValue` of `getEntryValue` member functions.

Iterators

Indices and part lists are two types of collections that NeoAccess supports. An **iterator** is an object which is used to traverse a set of entries in a collection. The use of iterators greatly simplifies access to and manipulation of collections by application developers. NeoAccess iterators have a common base class of CNeoIterator. The set of operations supported by this class includes the ability to traverse the collection forwards and backwards, to test whether there are more items in the collection beyond the current item and to reposition the iterator a specific number of entries forward or backward and position to the beginning of the list again. NeoAccess iterators are often called **keyed iterators** because of their unique capability to iterate over a subset of a collection based on an abstract select key. See the CNeoIterator, CNeoIndexIterator and CNeoPartListIterator sections for more information on NeoAccess's iterator classes.

The order in which results of a search are obtained is sometimes as important as determining which objects meet the criterion and which do not. The order in which objects are returned by an iterator is based on the sort order of the index or part list being traversed. NeoAccess includes a query optimizer which chooses which index will provide the best performance given the specified selection criterion.

It is important to note that iterators provide, in effect, incremental search capabilities. The process of initializing an iterator results in the retrieval of the first of all matching objects. It is only as the iterator's `nextObject` and `previousObject` member functions are used to move forward and backward in the set that other objects are obtained from the database cache. As a result, the cost of constructing an iterator is relatively low and fairly constant even when iterating over extremely large data sets. Further, the memory requirements of an iterator can be low as it refers to only one object at a time, not the entire set.

Finding Objects Using an Iterator

The code snippet that follows is an example of a function that iterates through a set of objects having object IDs between one and a hundred.

```
void offTheWall(CNeoDatabase *aDatabase)
{
 CNeoIDSelect                    lowTerm(1);
 CNeoIDSelect                    highTerm(100);
 CNeoAndSelect                   key;
 CNeoIndexIterator *             iterator;
 TNeoSwizzler<CBottle>           object;

 // Configure the key to match all objects between 1 and 100.
 lowTerm.setOrder(kNeoHighOrEqual);
 keyhigh.setOrder(kNeoLow);
 key.addTerm(&lowTerm);
 key.addTerm(&highTerm);

 // Get an iterator which matches all CBottle objects with
 // object IDs between 1 and 100.
 iterator = aDatabase->getIterator(kBottleID, &key, TRUE);

 // Prime the loop.
 object = (CBottle *)iterator->currentObject();
 while (object) {
      // What should happen if one should fall?
      object->shatter();

      // Another bottle within range?
      object = (CBottle *)iterator->nextObject();
 }

 // Cleanup.
 key.removeTerm(highTerm);
 key.removeTerm(lowTerm);
 delete iterator;
}
```

The snippet begins by configuring the selection terms to match the objects of interest. The `setOrder` function of a selection object is used to specify its relational operator or order. The configuration of the local variable `lowTerm` is set to match those objects with an object ID of one or greater, while the local `highTerm` will match those with object IDs less than one hundred. Both of these terms are then added to the CNeoAndSelect key to form a compound select key which matches those objects with object IDs between one and one hundred.

Having configured the selection criteria, the database's `getIterator` function is used to obtain a pointer to an initialized iterator object which matches those CBottle objects with object IDs within the given range. The argument to the `getIterator` call is the class ID of the class of objects to be searched. The second argument is the selection criteria. The third argument indicates that the CBottle class and its subclasses are to be searched. This is an example of a **deep search**.

A newly initialized iterator is set to refer to the first matching object in the collection. The iterator's `currentObject` function is used to return a pointer to that object, which is assigned to the local swizzler

`object`. The `nextObject`, `previousObject` and `currentObject` functions of an iterator do not add a reference to the object returned, though the assignment operator of a swizzler does. This insures that the object will not be purged should a garbage collection occur while the swizzler refers to the object.

The `while` loop of this snippet continues until all objects in the set have been found. The iterator's `nextObject` function is used to advance through the set of matching objects.

Having processed all objects within range, the snippet ends by cleaning up the select keys and iterator. The `removeTerm` function of the CNeoAndSelect is used to remove the terms. This is necessary because the destructor of CNeoAndSelect otherwise assumes that its terms where dynamically allocated in the heap and, as a courtesy, attempts of delete them. Finally, the iterator dynamically allocated by the database's `getIterator` function is deleted from memory and destructed.

Adding and Removing Objects Using an Iterator

Iterating over a part list is just as easy as iterating over an index. Witness the `graduation` code snippet below. The `getIterator` function of an ENeoPartMgr returns a keyed part list iterator which refers to all students on the class roster with a first name of Bob.

```
void graduation(ENeoPartMgr *aRoster, ENeoPartMgr *aAlumni)
{
 CNeoNameSelect        key("Bob");
 CNeoPartListIterator *iterator;
 TNeoSwizzler<CStudent>grad;

 // Get an iterator which matches all CStudent objects named Bob.
 iterator = aRoster->getIterator(&key, TRUE);

 // Prime the loop.
 grad = (CStudent *)iterator->currentObject();
 while (grad) {
      // Remove the graduate from the class roster.
      // <Pomp and Circumstance>
      iterator->removeCurrent(grad);

      // Brother, can you spare a million?
      aAlumni->addToList(grad);

      // Oh, it's you Bob!
      grad = (CStudent *)iterator->currentObject();
 }

 // Cleanup.
 delete iterator;
}
```

Iterators update themselves automatically when the collection they refer to changes. An iterator will continue to refer to a specific entry in a collection even when other entries are being added and removed from the collection. But if the entry that an iterator refers to is removed from a collection, then the iterator repositions itself to refer to the next matching entry. While the overall structure of the `graduation` function is similar to `offTheWall` above, note that because graduates are removed from the part list before proceeding to the next grad, the final statement of the `while` loop is a call to the iterator's `currentObject`, not `nextObject`. This is because the iterator was implicitly repositioned to the next graduate by the iterator's `removeCurrent` function.

Database Searching

The database class includes a set of functions to locate objects that match a given selection criteria. These member functions can search a specific class of objects, or a base class and all of its subclasses. (The base class of all persistent classes, CNeoPersist, also includes a set of convenience functions for locating objects. Your application-specific subclasses of CNeoPersist can include additional member functions that provide

similar capabilities, `CShape::FindShapeByRegion`, for example.)

> *Note*
>
> These search functions of the database class are best suited for situations where only one object matches the given criterion or where the order in which objects are found is not important. This is because the order in which matching objects are found by these functions is undefined.

As another example, imagine that the personal productivity application mentioned earlier includes the ability to list all those people that the user is scheduled to meet with today. Imagine also that the day object or calendar class defines a function that locates the proper set of person objects. This function, let's call it `FindTodaysPeople`, locates all the appointment objects for today, identifies the people with whom the appointments are scheduled, and then locates and returns those person objects. This may sound complicated, but, in fact, it is quite easy to implement.

Searching the database for room object having a specific room number requires only that the condition of the search be defined using a CNeoIDSelect object. The database's `findObject` function can be used to locate this object. An example of how this could be implemented is as follows:

```
CRoom *CAppointment::FindRoom(const NeoID aRoomNo)
{
 CNeoIDSelect      key(aRoomNo);
 CRoom *           room;

 room = (CRoom *)gNeoDatabase->findObject(kRoomID, &key, FALSE);

 return room;
}
```

The foregoing code assumes that rooms are located in classes that have a class ID of `kRoomID` and a data member with a tag of `pRoom`.

Apply a Function to a Set of Objects

Occasionally, an application needs to apply a function to a set of objects until a particular condition is met. Situations where this might be a useful thing to do include serially searching for a particular object, or counting objects having a specific state.

There are several different ways in which this can be accomplished with NeoAccess. The recommended way would be to create an iterator object and iterate over the set of objects or use the iterator's `doUntil` function. Another way would be to pass a function pointer to one of the `FindByX` functions of CNeoPersist. Yet another alternative would be to use the database's `doUntilObject` function. This final option is the one we will discuss in the remainder of this topic.

Imagine a situation where an application needs to count the number of objects in a database. You could do this simply and quickly by using the database's `getObjectCount` function, but assume that the application would rather iterate over each object and count each of them as they are encountered. Consider the example given below:

```
void *CountObject(CNeoCollection *aNode, const short aOffset, void *aParam)
{
 // Count this object.
 (*(long *)aParam)++;

 return nil;
}

long CountObjects(CNeoDatabase *aDatabase)
{
 long count = 0;

 // Count all objects in the database.
 aDatabase->doUntilObject(nil, kNeoPersistID, TRUE, CountObject, &count);

 return count;
}
```

Look first at the implementation of `CountObjects`. It calls the database object's `doUntilObject` function. The fourth argument of this call is a pointer to a function, `CountObject`, which is shown just above `CountObjects`.

`CountObject` ignores its first two arguments, but treats its third as a pointer to a long integer variable, which is incremented each time `CountObject` is called.

Let's examine the parameters of `doUntilObject` closely. The first argument, which is `nil`, can refer to an object in the class list to be searched. If it did refer to an object, then the function would be invoked for that object and all objects following it in its class list. The fact that `CountObjects` has set it to `nil` indicates that `CountObject` should be called for all objects of the class.

The second argument is a class ID. This indicates the class of objects to be searched. In this case, it has been set to refer to the base persistent class CNeoPersist. If the first argument had not been `nil`, then the class to search would have been the starting object's class and this second argument would be ignored.

The third argument indicates whether objects that are subclasses of the class indicated by argument one or two should also be searched. The fact that `CountObjects` has set this to `TRUE` means that all objects in the database will be counted.

As we've already seen, the fourth argument is the function to be invoked. The prototype of this function is defined by `NeoTestFunc1`. The parameters passed to this function and the value it returns will be discussed in more detail in the following example.

The fifth and final argument is a parameter value passed to `CountObject`, in this case a pointer to a long integer. This can be any value that the application and function agree upon.

Let's look at another example that makes use of the first two parameters and the return value.

```
class CMessage: public CNeoPersistNative {
public:
                          …
 Boolean                  isPriority(void);
                          …
protected:
                          /** Instance Variables **/
 long                     fPriority;
                          …
};

void *DisplayPriorityMsg(CNeoCollection *aNode, const short aOffset,
                              const NeoLockType aLock, void *aParam)
{
 Boolean                      done     = FALSE;
 TNeoSwizzler<CMessage>       msg;

 // Get a pointer to the indicated object.
 msg = (CMessage *)aNode->getObject(aOffset);
 if (msg) {
      // If it is a priority message, then present it to the user.
      if (msg->isPriority())
           done = DisplayMsg(msg);
 }

 // Stop searching upon user request.
 return (void *)done;
}

void DisplayPriorityMsgs(CNeoDatabase *aDatabase)
{
 // Present priority messages until the user says to stop.
 aDatabase->doUntilObject(nil, kMessageID, FALSE,
      (NeoTestFunc1)DisplayPriorityMsg, nil);
}
```

Consider a messaging application that a user has just launched. Its initialization process involves querying the user's message database to locate and present priority messages one at a time until the user says to stop.

This application defines a subclass of CNeoPersist, a grossly abbreviated definition of which is shown above. The routine `DisplayPriorityMsgs` searches the message database by using the database's `doUntilObject` function. It indicates that the routine `DisplayPriorityMsg` should be invoked for each message object in the database.

The interesting part of this example is what happens in `DisplayPriorityMsg`. Notice the first two arguments are a pointer to a CNeoCollection object and a `short` integer. The class CNeoCollection is the abstract base class of all collection classes. It is used internally by NeoAccess to keep track of your application-specific objects in a database.

The second argument to `DisplayPriorityMsg` indicates the specific object of interest to `DisplayPriorityMsg`. The function `CountObject` in our first example didn't need to refer to the object itself; it was only interested in its existence. However, `DisplayPriorityMsg` needs to access the object directly, so it uses the `getObject` function to obtain a pointer to it.

`DisplayPriorityMsg` is interested in the object only if it is a priority message. If not, the function simply returns.

Notice the object has an additional reference added to it while the object is referenced by `DisplayMsg`. This is because the `getObject` function of CNeoCollection does not add a reference before returning the object. It is the caller's responsibility to do so. `DisplayPriorityMsg` uses a swizzler to manage this reference.

The return value of `DisplayMsg` indicates whether the user is interested in seeing any additional priority messages. The function `doUntilObject` stops immediately and returns to its caller any non-zero value returned by the function.

Object I/O

Most C++ compilers include a standard set of classes which implement an input/output facility which is referred to as a **stream**. The most common stream class supports the transfer of basic C data types such as integers, floating-point numbers and character strings to and from a file or to a console window.

While streams have been around for some time, our understanding of them continues to evolve. We know, for example, that we need different types of streams for different purposes. Application-specific environments may benefit from the use of a stream subclass which also supports application-specific data types, imaginary numbers, for instance. Other environments may find useful a stream that transfers data not to a file but across a network pipe or an inter-process communications channel. As you can see from these two examples, there are two dimensions in which stream derivations can occur. One dimension addresses the type of data being accessed. The other defines the source/destination of the data.

NeoAccess uses streams to address the issue of where data is coming from or going to. The abstract base class CNeoStream provides an interface through which basic data types can be read in or written out. This base class is subclassed to derive another abstract stream class, CNeoContainerStream. A container stream is further subclassed to create CNeoFileStream, which is used in reading from and writing a file. Other subclasses of CNeoContainerStream might be used to create OLE, OpenDoc or JavaBean containers.

The interface to CNeoPersist, the base persistence class of NeoAccess, includes a pair of object serialization member functions, `readObject` and `writeObject`, which are used to serialize the persistent state of objects to and from NeoAccess streams. Subclasses of CNeoPersist override these member functions so that their persistent data members are also preserved and restored appropriately. Embedded classes also include `readObject` and `writeObject` member functions. For the most part, `readObject` and `writeObject` functions can be written without regard for the type of stream being used, though the isRPCStream and `isContainerStream` member functions can be used to determine the general type of stream being used. The advantage of this device-independent approach is that a single set of member functions can be used to preserve and restore a class's state to any number of different stream types.

Some application frameworks which are supported in the standard NeoAccess release include their own set of stream classes. In order to make use of these streams, persistent objects in these frameworks usually need to override their own set of serialization member functions. In order to avoid asking developers to override multiple sets of serialization member functions, NeoAccess environment-specific support for these frameworks often includes a stream class which maps the native application framework's stream class into a CNeoStream derivative, which calls an object's `readObject` and `writeObject` member functions.

The NeoAccess database class, CNeoDatabase, provides an extremely powerful mechanism for accessing persistent objects. These objects use persistence properties provided by their base class, CNeoPersist. And together these three base classes — CNeoContainerStream, CNeoDatabase and CNeoPersist — create an incredibly powerful and high performance database engine which is both extensible and easy to use.

Just as there is a need for different types of streams when reading from and writing to different types of sources and destinations, the semantics of referring to a location differs depending on where it is. For example, the process of specifying a location in the file system is probably different than specifying a host location on the Internet. A **location** is used to designate a place where a stream object can read from and/or be written to. NeoAccess includes a set of location classes, all having a base class of ENeoLocation, which are used to specify the location of data which is to be accessed using a NeoAccess stream. A container location's `getStream` function is used to create the proper type of stream given the specified location. While the CNeoDatabase class includes convenience member functions of specifying the operating system file in which the database is to reside, the database's `getLocation` and `setLocation` member functions can also be used to designate the location of the database's container.

Persistent Strings

NeoAccess includes three types of persistent strings – a character array, a native string and an embedded string. NeoAccess developers need to recognize the strengths and weaknesses of each of these persistent string types. This understanding will allow you to determine which type to use in a particular situation.

Character Arrays

Character arrays, also called 'C strings,' are a standard built-in type used by virtually every C and C++ application. C strings are represented internally in all runtime environments as an array of bytes terminated by a null character. A character array data member of an object will be declared either as a fixed sized array of characters (char []) or as a pointer to an array of characters allocated discontiguously from the object proper (char *). The amount of memory used by a character array is a constant regardless of the current length of the string. In contrast, using a character pointer may require that a separate memory allocation be done for the character value, though the size of the memory block need only be as large as the length of the string plus the terminated null character. Character arrays are usually not as limited as to how long the string value can be.

NeoAccess supports the use of character arrays in persistent objects. Pointers to strings stored discontiguously from the persistent object (char *) are supported though ENeoString embedded objects (described below). Reading and writing a C string array data member of a persistent object is done by using the `readString` and `writeString` functions of the stream.

Character arrays have a fixed maximum size. This maximum size is the amount of space allocated for the array, both in memory and in the database. Though all operating environments include very strong support for C strings, the use of native strings (described below) may offer advantages in cross-platform development situations. For example, some runtime environments don't include case insensitive C string comparison operations. Another factor to consider when deciding whether to use C strings is that the performance cost of determining the length of a C string increases linearly with the length of the string. Finally, because character arrays are stored contiguously in a persistent object, an additional i/o operation is not usually needed when reading and writing C strings.

Native Strings

Some execution environments (the Macintosh in particular) support a data type called a Pascal string. A **Pascal string** is a character array (unsigned char []) with the first byte being the length of the string value. Pascal strings are not necessarily null terminated (though whenever NeoAccess manipulates Pascal strings, it tries to put a null character at the end if there is room).

NeoAccess has a concept of a native string. A **native string** is a Pascal string in those environments which support Pascal strings. In all other environments (everything but the Macintosh), a native string is a C string. The maximum length of a native string value is 255 characters. Reading and writing a native string array data member of a persistent object is done by using the `readNativeString` and `writeNativeString` functions of the stream.

CNeoString is a class which implements a native string. The interface to this class is the same in all environments, though the implementation may differ depending on whether a particular environment supports Pascal strings.

Native strings have many of the same properties as C strings, including a fixed maximum size equal to the amount of space allocated for the array, strong support in all operating environments and efficiency from an i/o perspective. However, the maximum size of a native string is never more than 255 characters. On the plus side, the performance cost of determining the length of a native string is fixed regardless of the string length.

The use of native strings is recommended only for applications which include the Macintosh as one of the supported platforms, where the string will be used by an operating system or framework construct expecting Pascal strings.

Embedded Strings

One of the problems with both C strings and native strings is that the amount of space they occupy, both in memory and in the database, is fixed regardless of the string's value. Further, the fixed size nature of these

string types dictate a maximum length. The solution C++ developers commonly use to resolve this dilemma is to use a char * data member which refers to a non-contiguous block of memory containing the string. Yet, the logistics associated with managing this discontiguous string can be problematic, and persistence issues only complicate matters further.

NeoAccess includes an embedded string class called ENeoString which is used to manage a persistent variable-length string value much like a `char *` data member would in a non-persistent environment. The string value managed by an ENeoString object occupies a variable amount of space in the database and in memory. The maximum length of an ENeoString value is extremely large – two gigabytes. The `readObject` and `writeObject` functions of the ENeoString are used to read and write these embedded strings.

The maximum size of an embedded string is very high, making them best suited for strings that may grow over time or which will have a length greater than 32 bytes on average. Another advantage is that the amount of space used by an embedded string in memory is around 18 bytes plus the current length of string. The space used in a database by an embedded string is 14 bytes plus the current string length. Because the value of an embedded string is stored internally as a C string, all C string operations supported in a particular environment are also available for embedded strings. However, the performance cost of determining the length of an embedded string is fixed regardless of the string length.

Because the string value is stored discontiguously from the persistent object in memory and in the database, additional memory allocations and i/o operations are necessary when bringing the string into memory and when writing it out to the database. Changing the string length also necessitates that the space previously allocated for the old value in memory and, if permanent, in the database be freed and new space be allocated for the new length.

A final disadvantage is that ENeoString is an embedded class. Given the limited support C++ has for function delegation, persistent objects with embedded data members must override several member functions so as to explicitly delegate to its embedded data members.

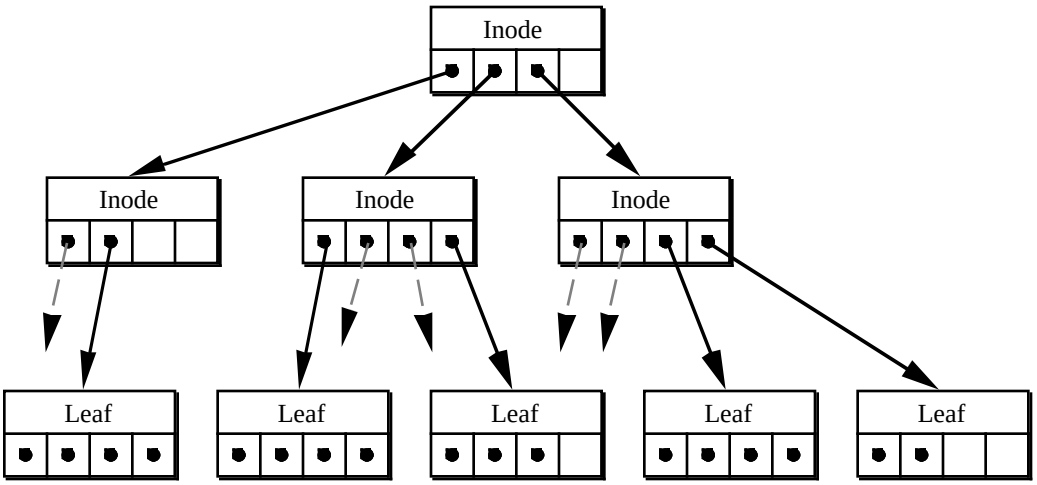
Collection Classes

Most application frameworks include a set of collection classes which are used to manage the relationships that exist between objects and other data in the application. Arrays and linked lists are two types collection classes that you might have worked with before. Database indices and part lists are two types of collections used heavily by NeoAccess-based applications.

The abstract base class from which all NeoAccess collection classes are derived is CNeoCollection. The only immediate subclass of CNeoCollection which is currently included in the standard NeoAccess release is CNeoNode, which is the abstract base class from which all **extended binary tree** (or simply **btree**) node collection classes are derived.

Btrees

Much of the power NeoAccess has for organizing objects in a database, including indexing and part lists, is obtained through the derivation and use of btree objects. As such, some of the more powerful uses of NeoAccess can’t be realized without first looking in greater detail at NeoAccess’s btree capabilities.



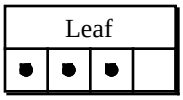
An Extended Binary Tree

A btree consists of a set of **node** objects organized into a tree structure. Each node is a separate C++ object with two or more branches (we call these branches **entries**). The set of nodes in a tree can be partitioned into two groups, **inodes** and **leaf** nodes. Inodes provide the tree with structure. They are the glue — the means through which the individual index nodes are connected. Leaf nodes provide the tree's content.

There are two general types of btree leaf classes. One whose entries refer directly to some other target object is called a **direct btree**. We call btrees whose entries do not refer to target objects **indirect btrees**. The target objects, which leaf nodes refer to, are sometimes called the **fruit** of the tree.

Note that not all node entries are always in use. The number of entries in a node that are used is called its **count**. Note that if a node has any unused entries, they are at the end of the entry array of that node. The proportion of entries in a tree that are used is referred to as the tree's **density**. The number of levels between the root node and the leaf node furthest down in the tree is referred to as the tree's **depth**.

The root of a btree can itself be a leaf node. Inodes are necessary only when the leaf entries in the tree are contained in more than one leaf object. Each leaf node in the figures shown in this discussion can have up to four entries (though NeoAccess nodes normally contain 32 entries). The figure below shows a tree with a single node having a count of three. This node is both the root and a leaf.



A Single Node Extended Binary Tree

While a tree structure may initially seem like unnecessary complexity, btrees are actually an ideal construct on which to construct database technology. While linked lists and arrays provide optimal serial access times, no construct can provide faster random access to large collections than btrees. Another advantage btrees have over other collection classes is that if a btree is persistent, then only that portion of the tree that is of immediate interest needs be in memory at one time. So persistent btrees work well in limited memory situations, even when dealing with huge collections of objects.

Using Nodes

Extended binary trees consist of a hierarchy of nodes. Each node has a header and a set of entries. CNeoNode, the base class from which all node classes are derived, makes no assumptions about how many entries are in a node, their content or even the size of each entry. This class simply provides a set of abstract member functions for manipulating the header and entries of a node. Subclasses override these member functions to implement and manage the specific capabilities of a node derivative.

Though CNeoInode is the class of node that is always used to provide structure within a btree containing more than one node, different kinds of btrees may contain different types of leaf nodes. But most searching and tree traversal member functions are unaware of the class of nodes that it may refer to. They rely on the abstract operations supported by CNeoNode to perform specific tasks. For example, there are situations where a function has a pointer to an object of an indeterminate class and that function would like to instantiate another node of that same class. It would use the `getAnother` function of the node to create an object of the same type.

Inserting and deleting entries in a node, expanding a tree to include more nodes or collapsing a tree into fewer nodes, all of these abstract operations are provided through the CNeoNode interface.

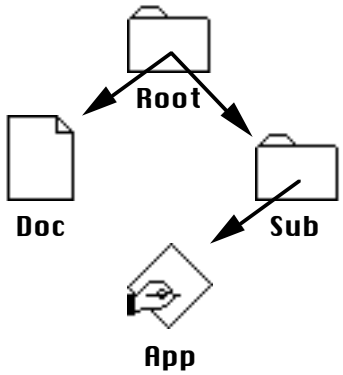
Index Classes

Indices organize objects in some sorted order. While indices are depicted at various points in the NeoAccess documentation as a single block, they may in fact consist of multiple collection objects, often btrees. NeoAccess uses btrees heavily because random searches need to be as fast as possible, and no construct can provide faster random access to large collections of persistent objects than can btrees.

The order in which objects in an index are sorted is determined by the `KeyManager` function of the index's leaf nodes. The `kNeoCanSupport` operation of the `KeyManager` function returns a select key that can be used to uniquely identify the entry (or entries in an inverted index) which refers to the given object.

Primary and Secondary Indices

Though all objects of a particular class are usually grouped by class, NeoAccess provides the ability to organize a class in more than one sorting order. To illustrate this feature, consider one way in which the Windows Explorer, the Macintosh Finder or any other hierarchical file system browser might be implemented using NeoAccess.



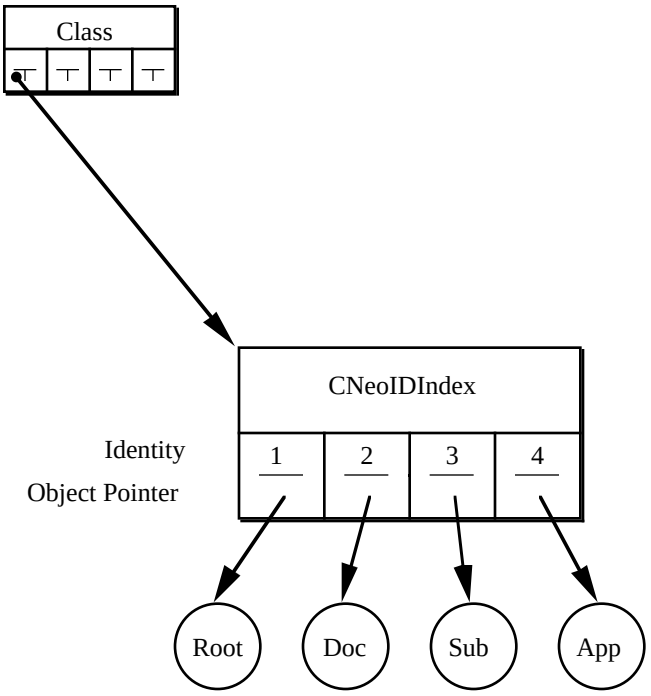
Sample File System Containment Hierarchy

Suppose the file system contains a set of four files having a configuration as depicted in the figure above. The file system assigns a 4-byte value to uniquely identity each file. Each file also tracks the identity of its parent. That is to say, the class CFile has two indices.

For example, Sub is the parent of App, and Root is the parent of both Doc and Sub. So in this example, file objects are sorted by file ID and by parent ID. The parent ID of Root is zero, because it is the root of the file system tree.

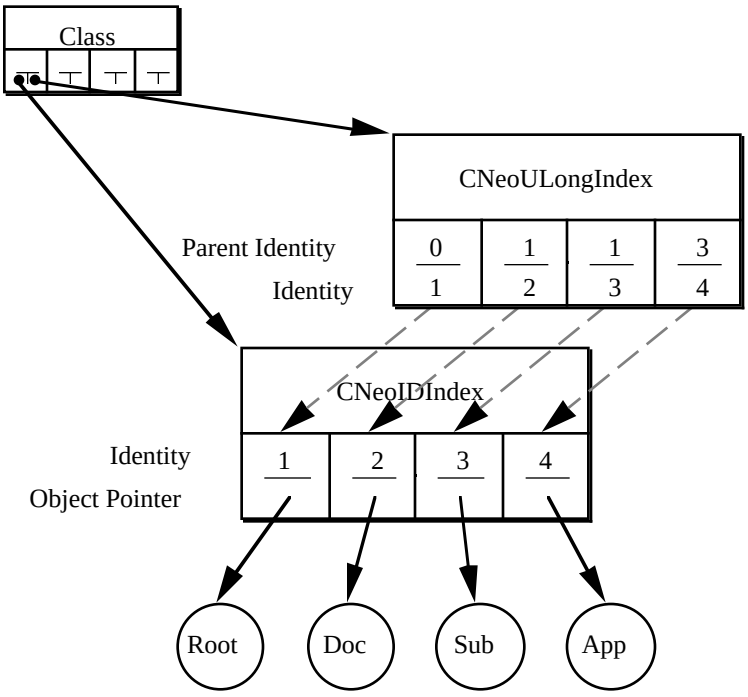
The first index, which is called the **primary index** and is always a direct btree, organizes file objects in ascending by ID. Sorting objects in ascending order by ID is in fact the default sorting order of objects in a NeoAccess database. The second index sorts file objects by parent ID. Any index beyond the primary index is called a **secondary index**. Secondary indices are always indirect btrees. (Direct and indirect btrees were discussed in greater detail in a previous discussion titled "Btree Classes.")

Classes are themselves btrees – and much of NeoAccess is implemented as btrees. The entry of the class node that refers to the CFile class keeps track of all the indices of that class.



A File System's Primary Index

The class node and primary index are depicted in the above figure. Note how the class node entry refers to the primary index tree, whose leaf entries refer directly to the file objects.



A File System's Complete Set of Indices

The second diagram shows that the CFile entry of the class node actually refers to both index trees. But note that the entries of the secondary index don't refer to the file objects directly. Instead, they contain the identity values corresponding to entries in the primary index. These identity values are used to locate the file objects by using the primary index.

Locating an object using a secondary index is a two step process. The first step involves locating the proper entry in the secondary index. The second step uses the value found in the secondary index to locate the object using the primary index.

Type-Specific Indices

There are three general characteristics of an index class which makes it different from other index classes:

- The order in which entries are sorted in the index,
- The data members that make up each entry of the index, and
- The type of data member upon which entries in the index are keyed.

As indicated earlier in this discussion, the order in which entries in an index are sorted is defined by the `KeyManager` function of the index class.

The set of data members that needs to be present in every index entry is dependent on how entries are sorted in the index and whether the index is direct or indirect. If entries are sorted by name, then the name value must either be included as a data member of each index entry or the name value will need to be obtained by asking the object that the entry refers to. If the index is indirect, the key value used to access the object using the primary index must be included in the entry. (The primary key is normally the ID of the object.)

Most NeoAccess index classes are type-specific. A **type-specific index** is a leaf index class that sorts entries based on a data member of a particular type. For example, CNeoLongIndex is a type-specific index class that sorts entries based on a long integer value.

Using a type-specific index class to organize application-specific objects involves calling the `addKey` function on the metaclass object for that class. The arguments passed to `addKey` include the class ID of the index class and a tag value identifying which data member of the application-specific object to key on. For example, a call to the metaclass for the CEmployee class to sort employees by salary, using CNeoLongSelect, would be constructed as follows:

```
meta->addKey(kNeoLongIndexID, pSalary);
```

The long index uses `pSalary` tag to obtain the salary value using CEmployee's `getValue` function.

Key Point

When using type-specific indices, it's very important that the `getValue` function be overridden by the persistent class being indexed. Indices use this function to obtain the data member value during database insertions, searches and deletions.

Developers will rarely need to create their own index classes which are direct descendents of CNeoNode. The type-specific index classes that are included in the standard NeoAccess release are indirect btree classes that assume objects are sorted primarily by object ID. Type-specific indices can be used as index leaf nodes or part list leaf nodes.

String Index Classes

NeoAccess includes three string index classes – CNeoStringIndex, CNeoNativeStringIndex and CNeoBlobStringIndex. There are costs and benefits associated with using each of these classes. Developers should refer to the discussions of each of these classes in determining which most aptly suits the requirements of their application.

Consolidated Indices

It is sometimes useful to sort all objects having a given base class in a single index. Consider the file system example once again. Users usually want to view a folder's contents in alphabetical order without regard for the type of file. Though simply adding a third index to all classes having a base class of CFile would result in all folder objects being sorted separately from all application or document objects.

All objects contained in an index normally have a common leaf class. (Once again, the leaf class of an object is its most specific class.) A **consolidated index** is one which contains objects having a common base class but whose leaf class may differ. The class to which the consolidated index is attached is called the **owner class** of that index. A consolidated index is created by configuring metaclass objects of classes having the common base class to have an index owner class ID equal to the class ID of the base class.

Dynamically Adding and Removing Indices

Experience has shown that the searching capabilities of an application may vary from one release to another. Indeed, some applications may find it desirable to dynamically add and remove indices based on the needs of individual users of an application. One user of a geographical information system (GIS) may be interested in the topographical features of a region and therefore ask that this information be indexed. Another user of the same database may be more interested in accessing seismic data quickly. The `updateIndices` function of CNeoDatabase changes the indexing of objects in a database to that which is defined by the metaclass table.

Creating Domain-Specific Index Classes

Developers can sort objects in an application-specific order by creating domain-specific index classes. The index classes included in the Developer's Toolkit serve as excellent examples of how this might be done. The default index class, CNeoIDIndex, is a primary index that sorts objects in ascending order according to object identity. Any of the type-specific index classes are good secondary index examples. A concatenated index is one which sorts objects according to a major key value and one or more minor keys. CNeoParentNNameIndex is a concatenated index which has a major key or parent ID, a minor key of name and a second minor key or object ID.

Choosing a String Index

NeoAccess includes three type-specific index classes – CNeoStringIndex, CNeoNativeStringIndex and CNeoBlobStringIndex. Several factors need to be considered when deciding which class to use to index objects based on a string value. These factors include the string type of the key attribute as well as performance, memory and file space implementations.

The first consideration in this decision is the string format in the persistent object. If the key value of the index is a C string (with a maximum length no greater than `kNeoMaxStringLength`,) then CNeoStringIndex may be the best choice. If the key is a native string, then CNeoNativeStringIndex may be appropriate. In all situations, CNeoBlobStringIndex might be the best choice.

The maximum size of the string value, as well as memory and file space usage, also needs to be considered when choosing the proper string index class. CNeoStringIndex entries have a maximum length of 31 characters. The maximum length of a CNeoNativeStringIndex entry is 255 characters. The maximum length of a CNeoBlobStringIndex is effectively two gigabytes.

Each entry in a CNeoStringIndex uses 36 bytes, both in memory and in the database. Each CNeoNativeStringIndex entry is 260 bytes. Each CNeoBlobStringIndex entry in memory will use 18 plus the length of the string, while in the database it will use 14 bytes plus the string length. (See the discussion of "Persistent Strings" for full details.)

Each CNeoBlobStringIndex entry has an ENeoString data member – with all the costs and benefits that it implies. It should be noted, once again, that the value of an embedded string is discontiguous from other persistent data and therefore requires an additional i/o operation if this value is not already cached. This may have performance implications when searching a blob string index.

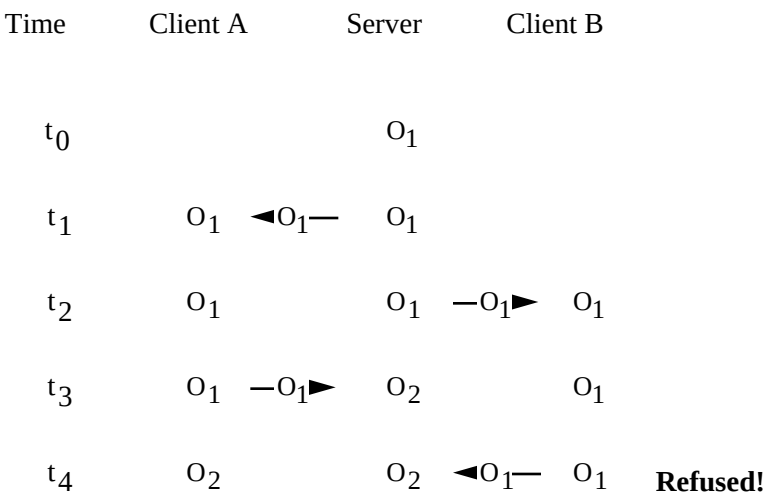
Developers actually have two additional alternatives to the three string index classes included with NeoAccess. One option is to develop a domain-specific index class which more accurately reflects the specific needs of your application.

Another possibility is to use CNeoIDList. Though the entries of this index don't include a string value, the index's `getEntryValue` function can obtain the key string value by using the primary index to find the object fruit the entry refers to then, using the fruit object's `getValue` function, obtain the string value. While there are significant performance costs associated with this approach, it would eliminate the need to duplicate the string value in each index entry, as well as in the fruit object proper.

Object Versioning

NeoAccess includes constructs to facilitate sharing of objects within an application. The object reference count and busy bit are two mechanisms that provide this support. NeoAccess includes an additional construct called versioning that provides even greater concurrency support. Here's how it works.

A persistent object's `commit` function is used to synchronize the on-disk state of dirty objects with their in-memory state. When object versioning is enabled, by defining the `qNeoVersions` symbol when compiling NeoAccess, persistent objects contain a permanent data member, `fVersion,` which is changed each time objects are updated on disk.



Object State Transition Diagram for Client/Server Application

The server portion of a client/server application based on NeoAccess might make use of object versioning to manage contention between two clients that attempt to modify a single object. Consider the transition diagram shown above. At time ***t1*** client ***A*** requests the state of object ***O***. The server responds to this request by returning the state ***O1***. At ***t2*** Client ***B*** also requests the current state of ***O***. Given that the state of the object has not been changed since ***t1***, ***O1*** is returned. Client ***A*** modifies ***O*** and commits the change at ***t3***. The change is accepted by the server because the object state given by Client ***A*** at ***tO***. However, the act of committing this change causes the state of ***O*** to be set to ***O2***. Client ***B*** attempts to submit its own change to ***O*** at ***t4***. However, the object state sent with this commitment is ***O1***. If the server were to accept this change it might cause the change submitted by client A at ***t3*** to be lost. The server recognizes the version conflict and refuses Client ***B***'s change.

Other possible uses of versioning might include a journalling-based recovery mechanism, transaction processing support and dynamic meta-object protocols.

Exception Handling

NeoAccess uses exceptions if something goes wrong during an operation. An exception is an abnormal condition that occurs during program execution. Possible exception conditions include running out of memory, or attempting to read or write beyond the end of a file. NeoAccess is designed to minimize the

occurrence of exceptions. However, should one occur, NeoAccess will clean up as best as it can and then continue to signal the exception. Your application objects should set up NEOTRY and NEOTRYTO blocks to capture and recover from an exception when it occurs (NEOTRYTO blocks are described below).

There are two types of exceptions most likely to be raised by NeoAccess: resource limits and programming errors. The frequency of these conditions can be greatly reduced by thoughtful design and implementation of your application.

Resource limits occur when your application exhausts a resource available to it. The resource most commonly exhausted is memory. NeoAccess, the class library and the operating system all provide mechanisms for optimizing memory usage and recovering from shortages. See the topics “Temporarily Permanent Objects” and “Purging Objects in the Cache” for more information on memory management in NeoAccess.

Another resource limit that is rarely encountered, though one for which developers should be prepared, is file space. The length of a database will grow as your application adds objects to it. Eventually, the database could consume all the available space on the volume. This situation is problematic because NeoAccess will usually encounter this error as it sets up to commit the changes made to a database that already has too many objects in it. The proper way to handle this situation is to advise the user to either remove some objects from the database or do a SAVE AS of the database using another volume which has enough file space to hold the database.

Programming errors may also cause exception conditions. Of course all production applications are sufficiently tested to eliminate these errors before the application ships. Your applications should, never the less, be prepared for programming-induced exceptions to occur.

The traditional exception handling construct is a NEOTRY block. A NEOTRY block takes the following form:

```
// Prepare to do something that may cause an exception.

NEOTRY {
     // Do something that may cause an exception.
}
NEOCATCH {
     // Do whatever it takes to clean up after yourself.
}
NEOENDTRY;

// Do whatever it takes to clean up after yourself.
```

The problem with this construct is that the code inside the NEOCATCH block is often similar to (or exactly the same as) the clean up code just below the NEOENDTRY statement.

Rather than duplicating this code, NeoAccess provides a NEOTRYTO, NEOCLEANUP and NEOENDTRYTO construct.

```
// Prepare to do something that may cause an exception.

NEOTRYTO {
     // Do something that may cause an exception.
}
NEOCLEANUP {
     // Do whatever it takes to clean up after yourself.
}
NEOENDTRYTO;
```

This allows your application to avoid duplicating clean up code. Object code space is therefore reduced as well.

Temporary Objects

In the course of execution, some applications generate vast amounts of intermediate results in memory. Much of this data may be disposable, as it can be reproduced at application startup time using other persistent data. Offscreen bitmaps in graphics programs and lookup tables in algorithmically-intensive applications are two situations where this might be the case. While data of this sort can be recreated, in low-memory situations it might be more efficient to cache the data to disk, rather than destroy and recreate it later.

NeoAccess makes it easy for an application to organize and access application-specific temporary data; so easy that you might want to consider using it to organize and cache objects that don't persist after the application quits. The advantage of doing this would be a potential reduction in memory requirements, because these objects could be written to disk and then be purged from memory when available memory becomes critically low.

Marking an object temporary is done by setting its `fTemporary` bit. Temporary objects are managed just like permanent objects. As such, they can be added and removed from a database using `addObject` and `removeObject`, respectively. And, they need to be marked dirty when their in-memory state is changed from their state on disk.

NeoAccess provides the ability to mark an entire class of objects temporary by using the database's `markClassTemporary` function. All objects of a temporary class are removed when the database is opened and closed — whether or not the `fTemporary` bit of each object is set. Objects belonging to a non-temporary class, but having their `fTemporary` bit set, will be deleted when the application calls the database's `removeTempObjects` function with a non-zero parameter value.

Object Caching

NeoAccess supports a sophisticated object caching mechanism, which greatly improves access times by minimizing disk activity. NeoAccess keeps objects in memory even after an application deletes its references to it. If the application tries to access the object again later, NeoAccess can locate it without having to reread it from disk. Caching can improve access times by as much as 20 times in some situations. (Your application's mileage may vary.)

The object cache uses memory not otherwise being used by the application. NeoAccess limits the size of the object cache to something close to the amount specified by the static variable `CNeoPersist::FCacheSize`. If an allocation causes the cache to exceed this limit, the `purge` function of all open NeoAccess databases will then be called to "garbage collect" objects already in the cache so that the allocation can proceed. The single argument to `purge` is a pointer to a long integer that indicates how much memory is needed. The database will attempt to free up at least this much memory. Depending on how NeoAccess has been configured, `purge` may free more memory than is currently needed. This is done in order to reduce the frequency of low memory situations, while simultaneously maintaining the usefulness of the object cache.

Each database object has an associated "purging action" state. The **purging action** of a database defines whether to commit the database before or during garbage collection. The purging action state of a database can be set using the `setPurgeAction` function of the database. The default value is `kNeoNoPurgeAction`, which indicates that the database shouldn't be committed during garbage collection. A value of `kNeoCommitBeforePurge` causes NeoAccess to purge the database before attempting to purge objects from the cache. A value of `kNeoCommitBeforeGrowZone` causes NeoAccess to purge the database if the required amount of space to be purged can not be satisfied.

> *Note*
>
> Due to concurrency issues associated with committing and garbage collecting in a multi-threaded environment, the purge action state of a database must be `kNeoNoPurgeAction` when NeoAccess has been built with the compile-time symbol `qNeoThreads` defined.

Schema Evolution

C++ is a static language. That is, the definition of classes is defined at compile time. Yet an unavoidable reality of software development is that things change. The expectations of users are dynamic. NeoAccess includes several mechanisms that application developers can use to evolve the performance and feature set of their applications at runtime and in future updates. This support includes the ability to add and remove indices on a class and even to change or remove persistent data members of persistent classes. This process of changing the structure of a pre-existing database over time is called **schema evolution**.

The header at the beginning of every NeoAccess database contains a user format and NeoAccess format. The user format is obtained by calling the database's `getUserFormat` function. This value indicates which version of the application this database was written by. The database's `getNeoFormat` function returns the NeoAccess release the database was written by. Taken together, these two values can be used to determine the format of NeoAccess and application-specific objects in the file.

Format Objects

Format objects define the format of persistent objects read from and written to the stream. Every NeoAccess stream object refers to a pair of CNeoFormat objects. The `fInputFormat` data member of a stream refers to the object which defines the format of objects read from the stream. The `fOutputFormat` data member of a stream refers to the object which defines the format of objects written to the stream.

The database header is read into memory when the database is opened. Using the information obtained from the header, the database's `open` function uses the application's `getFormat` function to create and assign the input and output format objects, which will be used by the database's container stream to read and write objects in the database. The domain-specific application class can override the `getFormat` function to instantiate and return domain-specific objects, which are subclasses of CNeoFormat. These subclasses may override or extend the capabilities to the base format class to support domain-specific formatting constructs.

Once input and output format objects are assigned to the stream, the input stream's `compareClasses` function is called to determine whether the database's `convert` function should be called to update the format of objects in the database.

Having done this, the input stream's `compareIndices` function is called to determine whether the database's `updateIndices` function should be used to update the indexing of objects in the database to reflect that of the application's metaclasses.

Converting the Format of Objects in a Database

NeoAccess's schema evolution support includes the ability to convert the format of objects in a database written by different versions of a NeoAccess-based application. As new versions of an application are released, the format of objects in databases written by other versions of the application may change. Developers may want to update the format of these objects to reflect those changes.

Key Point

Newer versions of an application are not REQUIRED to convert a database created by an earlier release of the application, as long as it does not require that new persistent data members be preserved if modified. The `readObject` and `writeObject` member functions of persistent classes in these applications that don't convert old databases need to be capable of reading and writing objects in the old file format.

Once input and output format objects are assigned to the stream of a database, the input stream's `compareClasses` function is called to determine whether the database's `convert` function should be called to update the format of objects in the database. The database's `convert` function calls the `convert` function of every object in the database.

If the format of an object is the same in both the input and output formats, then the object's `convert` function does nothing.

If the format of an object is different, yet the file space an object uses in the file is the same in both input and output formats, then the object's `convert` function simply marks the object dirty, so that it is written in the new format the next time changes in the database are committed.

If the amount of space an object occupies in the database differs between formats, then the object's `convert` function will free the space allocated for the object in the input format and then allocate file space the object needs given the output format.

Changing Object Indexing

NeoAccess supports a second form of schema evolution, which is the ability to change the indexing of objects in a database. After the input and output format objects are assigned to the stream of a newly opened database and the format of objects in the database are converted if necessary, the input stream's `compareIndices` function is called to determine whether the database's `updateIndices` function should be used to update the indexing of objects in the database to reflect the state of the application's metaclasses.

The database's `updateIndices` function compares the indexing of the classes in the database to the indexing defined by the metaclass table state. If any changes are found, then `updateIndices` will add or remove indices or re-index objects in the database.

Note

The primary key of a persistent class can not be changed in a database which contains permanent objects.

While `updateIndices` may be called automatically in response to differences in the input and output formats of a database, application developers can change the configuration of metaclass objects and call `updateIndices` at other times during execution of the application.

DynaObjects

C++ is a statically compiled language. C++ classes are therefore defined at compile time. NeoAccess stores, organizes and retrieves statically defined persistent C++ objects and data in a database. To the extent that these objects are statically defined at compile time, so is their layout in a database. To the extent that C++ class definitions can evolve from one release of an application to another, NeoAccess includes schema evolution constructs which allows the layout of objects in a database to also evolve between releases.

Some object-oriented languages, such as SmallTalk and some Lisp-based environments, include the ability to dynamically define and evolve classes at runtime. These environments also support the ability to add and remove data members from individual objects. The programming interface to these dynamic object and class evolution capabilities is sometimes referred to as a **dynamic meta-object protocol**.

Object-oriented environments that support a dynamic meta-object protocol implement classes and objects as a collection of attributes. An object containing a variable-length collection of attributes is called a **dyna-object**. A dynamic class is a dyna-object the members of which constitute the default data members of objects of that class. A dyna-object which represents a class is called a **prototype**. Creating an instance of a particular class involves cloning the class's prototype. Default values of the newly created object are taken from the attribute values of the prototype. Every attribute of a dyna-object has a symbolic name or **tag**. Resolving a reference to an attribute value involves searching the collection of data members looking for one with that tag value.

The dynamic object support of NeoAccess, which is enabled when the compile-time symbol `qNeoDynaObject` is defined, implements a persistent dynamic meta-object protocol. A dynamic class inherits data members from its super class, if it has one. A dynamic class can even have a set of static attributes. Some static attributes, such a class name and super class ID, are specific to the leaf class. But most static attributes are shared by a base class and all subclasses. Both types of static attributes are supported.

As is the case with statically defined C++ classes in NeoAccess, the indexing of objects of a dynamic class can be defined and even changed at runtime.

Persistent dynamic classes are defined on a per database basis. The class ID of a dynamic class is assigned when the prototype of the class is created in the database. Because different databases can contain a unique set of dynamic classes, the metaclass table is not shared by all open databases when `qNeoDynaObject` is defined. Instead, a new metaclass table is created and initialized by the constructor of the database object.

Defining a Dynamic Class

The C++ base class of all NeoAccess dyna-objects is CNeoDynaObject, which is a subclass of CNeoPartMgr. A dyna-object's part list is used to maintain direct references to the attributes of the object. The `getClassID` method of CNeoDynaObject returns the value of the dyna-object's `fClassID` data member. A prototype is a dyna-object with a class ID of `kNeoPrototypeID`. The object ID of a prototype is the class ID of the dynamic class the prototype defines.

```
NeoID                 classID;
CNeoDynaObject *      prototype;
CNeoAttribute *       attribute;
CNeoMetaClass *       meta;

// Create a prototype for a dynamic person class.
prototype = database->createPrototype("DPerson");

// Add a couple of attributes to the prototype.
attribute = NeoNew CNeoLongAttribute(pIQ, 100);
prototype->addAttribute(attribute);
attribute->unrefer();
attribute = NeoNew CNeoULongAttribute(pAge, 38);
prototype->addAttribute(attribute);
attribute->unrefer();

// Sort people by IQ. Is this P.C.?
classID = prototype->getID();
meta = CNeoMetaClass::GetMetaClass(classID);
meta->addKey(kNeoLongIndexID, pIQ);
database->updateIndices();
```

The database's `createPrototype` method creates a CNeoDynaObject object with a class ID of `kNeoPrototypeID`. The object ID of this prototype is obtained by calling the metaclass's `GetUnusedID` method, which returns an unused class ID greater than or equal to `kNeoFirstPrototypeClassID`. A metaclass is created for this newly defined class. The attributes of the super class are added to the prototype and the prototype is added to the database. A reference to the prototype is returned to the caller of `createPrototype`.

All dynamic attributes have a C++ base class of CNeoAttribute. NeoAccess's dynamic object support includes type-specific subclasses of this base attribute class. The `addAttribute` method is used to add an attribute to a dyna-object. The symbolic name of an attribute is a tag used to refer to the attribute in the dynamic object. The value given to an attribute will be the default value of the attribute for new instances of this dynamic class.

C++ classes include support for static data members. Some static attributes, such a class name and super class ID, are specific to the leaf class. But most static attributes are shared by a base class and all subclasses. Both types of static attributes are supported. An attribute's `setStatic` method is used to indicate whether the attribute is static and if so whether is specific to the leaf class or that class and all subclasses.

```
// Add a static attributes to the prototype.
attribute = NeoNew CNeoLongAttribute(pPopulation, 0);
attribute->setStatic();
prototype->addAttribute(attribute);
attribute->unrefer();
```

Key Point

The value of a static attribute is shared by all instances of a class. As such, static attributes should be added only to prototypes, not dyna-object instances.

Defining the order in which objects of this dynamic class are to be sorted is done by using the `addKey` method of the dynamic class's metaclass object. As is the case when changing indexing of a static class, the `updateIndices` method should be called on the database if it contains objects of that class.

Working with Dynamic Objects in a Database

Once a prototype of a dynamic class is configured in a database, working with instances is very straightforward. The process of creating a dynamic object is similar to creating any other type of C++ object; you use the `new` operator to create a CNeoDynaObject object. A pointer to the database containing the prototype and the class ID of the object to be created are the only arguments passed to the constructor. The object created contains the attributes defined by the prototype, with their respective default values.

```
long                    IQ;
CNeoDynaObject *        genius;

// Give birth to a true genius.
genius = NeoNew CNeoDynaObject(database, classID);

// She's not a genius unless her IQ is greater than 160, right?
genius->getValue(pIQ, kNeoLongType, &IQ);
if (IQ < 160)
     genius->setValue(pIQ, kNeoStringType, "180");

// Add her to the database.
database->addObject(genius);
genius->unrefer();
```

A dyna-object can be treated just like any other C++ object. The attribute values of both statically defined persistent classes as well as dynamically defined ones can both be accessed using the `getValue` and `setValue` virtual functions. In the dyna-object case, the fact that values are obtained from CNeoAttributes stored in a direct part list is hidden using this generic interface.

The internal representation of an attribute value can sometime be hidden from an application. The `getValue` and `setValue` methods will make an effort to convert data between the internal format which the attribute is stored in the database and the format referred to by application code.

As is the case with statically defined persistent classes, dyna-objects are added to the database using the `addObject` method of the database. If the prototype of the dyna-object indicates that the object should be sorted on an attribute which the object does not have, then the object is not added to that index.

KEY POINT

If a dyna-object does not contain a key attribute value, then seaches which use the index which the object is missing from will not match this object, even if the select key is configured to match all objects.

Adding and Removing Attributes from a Dyna-Object Instance

Attributes can be added and removed from individual dyna-object instances at runtime. If an attribute on which an object is indexed is removed, then the entry for this object is removed from that index. Conversely, if a key attribute is added to a dynamic object, then the object will be added to the index at that time.

The internal format of an attribute value can change over time as attributes are added and removed from an object or prototype. Care must be taken though to avoid asking for attribute values in incompatible formats. There may also be some performance implications associated with converting to and from internal and requested formats.

```
// As we grow older, our experiences count more than IQ.
// This will result in the object being removed from the IQ index.
genius->removeAttribute(pIQ);

// She's a little sensitive about her age.
genius->removeAttribute(pAge);
attribute = NeoNew CNeoStringAttribute(pAge, "30-something");
genius->addAttribute(attribute);
attribute->unrefer();
```

Changing the Class ID of an Object

The class ID of a dyna-object is defined by the value of its `fClassID` data member. The `setClassID` method can be used to change the class ID of a dyna-object. Because objects are organized in a NeoAccess database by class, changing the class ID also involves re-indexing the object as defined by the metaclass of that class.

```
// Change the class ID of a dyna-object to be a person.
meta = CNeoMetaClass::FindByName("DPerson");
classID = meta->getID();
dyna->setClassID(classID);
```

Threads and Asynchronous I/O

Personal computer operating systems are becoming ever more sophisticated. Modern execution environments support asynchronous i/o operations and multiple cooperative threads of execution in a single process.

An **asynchronous i/o function** is one which schedules i/o which may not be completed until after the i/o function returns to its caller. The application continues to execute during the time between the operation being scheduled and its completion. The parameters passed to the scheduling function includes a pointer to an **i/o completion routine**, a function called by the operating system when the i/o operation completes. An i/o completion routine typically releases resources used while the i/o operation is in progress.

A **thread** is an execution context within a process. The execution environment of a traditional application includes such things as the current instruction pointer (also called PC value), an execution stack, a dynamic memory pool (also called a heap), a set of static memory values (also called globals), a set of open files and so forth. Each thread in a multi-threaded process has a separate PC value, execution stack and set of globals, though all threads in a process share the same address space and set of open files.

There are two general classes of threads, **cooperative threads** and **preemptive threads**. A cooperative thread operates much as cooperative processes do: each thread runs without interruption until it yields the processor to some other thread of process.

NeoAccess includes optional support for execution environments which allow for asynchronous write operations to a file. When enabled, this compile time option, named `qNeoAsyncIO`, can increase NeoAccess's overall throughput during the commit process. The file stream class maintains a free list of write buffers. The stream obtains a buffer from the free list, fills it with data, schedules the write operation and then continues execution while the write to the file takes place. When a write option completes, the completion routine returns the buffer to the free list. In this way the file stream is able to schedule as many asynchronous write operations as there were write buffers. If the stream requests another write buffer when all of them are in use by previously scheduled writes, the stream waits in a tight loop until the completion routine of one of the earlier scheduled write operations returns a buffer to the free list.

While asynchronous write operations are possible in this environment, asynchronous reads are not. This is because the application can not continue execution until a read operation completes because it needs the results of that read in order to proceed. However, it is possible to take advantage of asynchronous reads in a multi-threaded environment because only the thread performing the read operation needs that information on order to proceed. Other threads are able to proceed. The potential exists for dramatically increased overall throughput through NeoAccess in such an environment so long as other issues such as concurrency, scheduling and context switching (which are collectively referred to as **friction**) don't consume throughput gains.

NeoAccess's cooperative multi-threading support is enabled when built with the compile time symbol `qNeoThreads` defined. In this environment container streams obtain read and write buffers from a free list shared by all open container streams. Asynchronous write operations are performed pretty much as described above, with the one exception that threads yield instead of looping when waiting for a buffer to become available.

When operating in a multi-threaded environment, database objects are protected using a multiple-reader/single-writer semaphore. Each function that enters the database must first obtain a reference lock of a type appropriate to the kind of database operation being performed. Database query operations begin by obtaining a read reference. Database update operations need a write lock before they can proceed. Attempting to obtain a database lock may cause a thread to block. Blocked threads will be made ready as the resource they are trying to obtain becomes available. The database's `lock` and `unlock` member functions are used to obtain and free database lock references.

Thread objects in a multi-threaded applications which use NeoAccess must be derived from CNeoThread subclass which is native to the development environment being used. For example, if the application is built using the PowerPlant application framework, then application-specific thread classes should has a base class of CNeoThreadPP.

NeoAccess thread objects preserves the state of various NeoAccess global variables between the time the thread yields and when it regains control. For example, the value of the global variable `gNeoDatabase`, which refers to the current database, can be different for each active thread. These globals are preserved when the thread yields the processor and restored when the thread regains control.

Laundry

NeoAccess automatically keeps track of which objects have changed in memory and therefore need updating on disk. NeoAccess, in fact, supports two mechanisms which it can use to determine which objects need updating.

The most straightforward way is for it to simply mark the object dirty and then at commit time traverse the class list and index trees writing dirty objects to disk.

Another scheme uses a construct called a laundry list to keep track of all dirty objects. Using a laundry list at commit time can be much more efficient than not using one.

Laundry lists are enabled when NeoAccess is compiled with the symbol `qNeoLaundry` defined.

Configuring NeoAccess

NeoAccess is very configurable so that your application operates at peak performance in its unique execution environment. Much of this configurability is due to the fact that you have complete source code. We believe that NeoAccess is the best object-oriented database solutions available. However if you don't agree, then you can change whatever you don't like. How's that for power?

kNeoMarkSize

NeoAccess supports the ability to create databases larger than 4GB in this runtime environments that support extended file lengths. This feature is enabled by defining the value of `kNeoMarkSize` to be 8 bytes. By default `kNeoMarkSize` is 4 bytes.

qNeoAsyncIO

NeoAccess includes optional support for execution environments which allow for asynchronous write operations to a file. When enabled, this compile time option named `qNeoAsyncIO`, can increase NeoAccess's overall throughput during the commit process. The file stream class maintains a free list of write buffers.

This symbol should be undefined unless `qNeoThreads` is defined.

qNeoByteSwap

Define this constant when building NeoAccess for use on a machine swaps the internal order of short integer and long integer values. This symbol is either defined or undefined automatically in the platform-specific header file.

qNeoDebug

NeoAccess source code has been instrumented with countless assertions which verify the integrity of the system. These assertions are enabled when NeoAccess is built with `qNeoDebug` defined. If for some reason you experience a problem when doing development with this symbol undefined, you will save yourself countless hours of heartache by rebuilding with `qNeoDebug` defined.

A significant downside to having `qNeoDebug` defined is that NeoAccess will operate much slower than when it is not defined. However most applications will continue to function even at these slower speeds.

Make sure that this compile time symbol is defined throughout the development process but undefined before building the final application.

qNeoDebugFreelist

When `qNeoDebugFreelist` is defined NeoAccess maintains a dynamically allocated bit map in memory each bit of which refers to a quantum (typically 8 bytes) of file space. If a bit is on this indicates that the file space this bit refers to is allocated. When `qNeoDebugFreelist` is defined NeoAccess will rigorously verify that the bit map matches the state of the free list.

qNeoDebugIO

Jeff Winkler, a user of NeoAccess, once spent an afternoon tracking down a file corruption problem which ended up being caused by the `writeObject` function of one of his application-specific persistent classes writing more bytes of data to the file than the value returned by that class's `getFileLength`. In order to simplify the process of finding such problems in the future, he designed a class we now call CNeoDebugIO. Jeff was kind enough to contribute this class for the benefit of all NeoAccess developers.

The debugging properties of CNeoDebugIO objects are enabled by defining the compile time symbol qNeoDebugIO. Objects of this class are used to verify that the `readObject` and `writeObject` member functions of a concrete persistent class never read or write more bytes than specified by their `getFileLength` function. Consider the implementation of CMyClass::readObject below:

```
void CNeoMyClass::readObject(CNeoStream *aStream, const NeoTag aTag)
{
 CNeoFormat *format= aStream->fInputFormat;
 CNeoDebugIOchecker(aStream, FALSE, this);

 NeoInherited::readObject(aStream, aTag);

 NeoAssert(format);

 fTeacherID = aStream->readLong();
 if (format->fNeoFormat <= kNeo2P0FileFormat)
     (void)aStream->readLong();// Old data member, no longer used.
}
```

By default, this compile time symbol is defined whenever `qNeoDebug` is defined.

qNeoDebugMemory

Some development environments include debugging mechanisms which track memory usage. This support is enabled in NeoAccess-based applications by defining the compile time symbol `qNeoDebugMemory`. See the “Core Leaks” discussion in the Debugging Tips portion of this manual for more information.

qNeoDynaObject

Though C++ is a statically compiled language, NeoAccess includes a feature which allows applications to dynamically define persistent classes at runtime, create instances of these classes and even add and remove data members from instances. This feature is called the DynaObject facility and it is enabled when the compile-time symbol `qNeoDynaObject` is defined. See the documentation of the DynaObject facility in the Extras folder of the NeoAccess release.

qNeoThreads

NeoAccess’s cooperative multi-threading support is enabled when built with the compile time symbol `qNeoThreads` defined. In this environment container streams obtain read and write buffers from a free list shared by all open container streams. Threads performing asynchronous write operations yield instead of looping when waiting for a buffer to become available. This symbol undefined by default.

qNeoVersions

When the `qNeoVersions` compile-time symbol is defined, every persistent object has a four-byte data member which is used as a version number. Every time an object is marked dirty, the object’s version number may change to reflect the fact that the object’s state has changed. This construct is most useful in shared environments. We recommend that applications be built with `qNeoVersions` defined.

Tutorial

Introduction

The structure and flow of control in applications built using NeoAccess is different in some ways from other applications. One major difference is that most other applications use the “inhale/exhale” approach to object persistence. That is, the entire contents of a file are read into memory as the document is created. As soon as the contents are in memory the file is closed, only to be reopened when changes are saved, at which point the entire file is totally rewritten. NeoAccess-based applications, on the other hand, leave the file open during the life of the document. Objects are brought into memory as needed. Updating a database involves writing out only those objects that have changed, not the entire database.

A great deal of effort has gone into making NeoAccess as easy to use as possible. The structure of NeoAccess allows complexity to be hidden from developers so they can be as productive as possible.

The fact that NeoAccess is a cross-platform database engine also contributes to some differences in the way that an application is structured. Every effort has been made to keep the interface to NeoAccess classes as consistent as possible across platforms. So even if you are developing a cross-platform application using different application frameworks, the NeoAccess interface will stay pretty much the same.

This section reviews a sample application, called Laughs, which comes with the NeoAccess Developer’s Toolkit. Carefully reading with section will help you to see how NeoAccess is integrated into and used in an application. While the sample document and application classes referred to in this discussion may not be the same as your development environment, many of the principles shown are applicable in all environments.

Note

It might be a good idea to bring up the Laughs project on your computer screen as you read through this tutorial. This will give you the opportunity to peruse the source code in its entirety as you progress through the tutorial.

Laughs

The NeoAccess Developer’s Toolkit includes several versions of a sample application called Laughs. The value of Laughs is its utter simplicity. Its implementation is uncomplicated because of its almost total lack of a user interface; some versions doesn’t even have an event loop. We’ll exploit this clarity in the discussion that follows to illustrate the steps developers take in building an application that uses NeoAccess. While from a user-interface perspective Laughs is simplistic, it actually uses some of the more advanced features of NeoAccess such as secondary indices, consolidated indices, blobs, part lists, swizzlers, different database formats and iterators.

As you proceed through this tutorial you should keep in mind the fact that our objective in the development of Laughs was to provide a sample application which uses a cornucopia of NeoAccess features but which is also easy to understand. Some constructs are more complex than they might have been had we not tried to show the multitude of NeoAccess features in a single application as we have.

Laughs is a simple database application. It creates a database to which it adds some objects. It then finds a pre-existing database from which it retrieves and displays some of the objects the database contains. As is the case with most C++ applications, `main` is a trivial routine. If you ignore some initialization that the application framework requires, `main` simply creates an application object, runs it, and finally deletes it before heading out the door.

```
int main(void)
{
 CLaughsApp *app;

 // PowerPlant asks that the following be done before creating an app.
 InitializeHeap(1);
 InitializeToolbox();
 (void)new LGrowZone(20000);

 // Create an application object, run it, then delete it.
 app = NeoNew CLaughsApp();
 app->Run();
 delete app;

 return 0;         // Time to go home.
}
```

The application class for Laughs is itself fairly plain. It is a subclass of the native NeoAccess application class, CNeoAppNative (Refer to the framework-specific header file of NeoAccess to determine how CNeoAppNative is defined.). It includes a constructor and an override of the `Run` function from the native application class.

The constant `kLaughsSig` is the signature of the application. The Macintosh Finder and file system both use this in assigning the appropriate icon to the application file and for matching documents with the applications that created them and are capable of opening them. Developers in other environments may use this value to refer to constructs unique to this application.

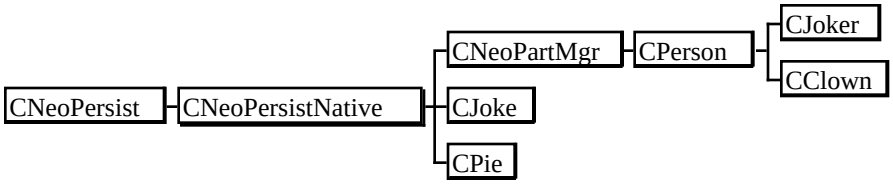
```
const OSType kLaughsSig         = 'Neo6';

class CLaughsApp : public CNeoAppNative {
public:
                        /** Instance Member Functions **/
                        CLaughsApp(void);

 virtual CNeoDoc *      createDocument(void);
 virtual void           openDocument(FSSpec *aSpec);
};
```

We'll look at the application class in more detail later. But first, let's examine the bizarre personalities one finds in Laughs.

The Persistent Classes



Inheritance Tree for Persistent Laughs Classes

Laughs defines seven persistent classes: CNeoPersist, an optional class generically referred to as CNeoPersistNative, CPerson, CJoker, CClown, CJoke and CPie. These classes have the ancestral relationships depicted in the above diagram.

CNeoPersist

CNeoPersist is an important NeoAccess class. It provides general persistence properties to its subclasses. CNeoPersist provides a comprehensive set of features. NeoAccess developers should be sure to review the discussion of this powerful class in the reference section of this document. Just to understand what the basic issues are, let's take a cursory look at a few elements of this class. An abbreviated class definition of CNeoPersist might appear as follows:

```
class CNeoPersist
{
public:
                        /** Instance Member Functions **/
 virtual                ~CNeoPersist(void) {}

 virtual NeoID          getClassID(void) const;
 static CNeoPersist *   New(void);

 virtual long           getFileLength(const CNeoFormat *aFormat) const;

                        /** I/O Member Functions **/
 virtual void           readObject(CNeoStream *aStream, const NeoTag aTag);
 virtual void           writeObject(CNeoStream *aStream, const NeoTag aTag);

                        /** Searching Member Functions **/
 static void *          FindByID(CNeoDatabase *aDatabase,
                             const NeoID aClassID, const NeoID aID,
                             const Boolean aDeeply, NeoTestFunc1 aFunc,
                             void *aParam, const NeoLockType aLock);
 static void *          FindEvery(CNeoDatabase *aDatabase,
                             const NeoID aClassID, const Boolean aDeeply,
                             NeoTestFunc1 aFunc, void *aParam,
                             const NeoLockType aLock);

                        /** Persistence Member Functions **/
 virtual void           setID(NeoID aID);
 void                   setDirty(const NeoDirty aReason = kNeoChanged);

                        /** Schema-Evolution Member Functions **/
 virtual NeoMark        convert(CNeoFormat *aOldFormat, CNeoFormat *aNewFormat);

                        /** Concurrency Member Functions **/
 void                   referTo(void);
 NeoRefCnt              unrefer(void);
 void                   setBusy(void);
 void                   setUnbusy(void);

#ifdef qNeoDebug
                        /** Debugging Member Functions **/
 virtual const void *   verify(const void *aValue) const;
#endif
};
```

The default destructor is significant because its virtual definition means that the destructors of all subclasses of CNeoPersist will also be virtual.

Most persistent classes can be referred to by a unique **class ID**. A class ID is simply a four-byte value much like a resource type. The pair of member functions, `getClassID` and `New`, provide a two-way mapping between a class ID and an object of that class. The instance function `getClassID` will return the class ID of a particular object. The static function `New` is a factory function used to create an object of a particular class. Concrete subclasses of CNeoPersist should override these functions to return the proper class ID and instance, respectively.

NeoAccess needs to determine the amount of file space taken up by objects of a particular class. It figures out an instance's file space requirements by calling `getFileLength`. Concrete subclasses will override this function. These overrides will simply add to the requirements of the parent class the amount of file space needed to preserve the persistent attribute values of that class. The format object is passed to this function so that a different value can be returned by `getFileLength` depending on the layout of this class of object in a database having this format.

NeoAccess utilizes a streams-based object serialization mechanism to preserve and restore the persistent state of objects. Persistent classes override the `readObject` and `writeObject` member functions to serialize and retrieve the persistent values of the class.

The most significant feature that NeoAccess provides that simple object persistence mechanisms can not is quick and efficient access to specific objects in a potentially huge set of objects. The most common interface for accessing objects in NeoAccess is through static functions such as `FindEvery` and `FindByID`. Subclasses may provide other static functions similar to these which provide simple access based on other selection criteria. The CShoe class of a shoe store management application might, for example, include a static function, `FindBySize`, which locates shoe objects by size.

Every persistent object is assigned an **object ID**. This ID can be unique among all objects of a particular class, or there can be multiple objects having the same ID. NeoAccess organizes objects in a database primarily by class. The default sorting order of all objects of a class is in ascending order by object ID. The ID of an object is set, not surprisingly, by using the `setID` function. If the ID of an object isn't set by the application at the time the object is passed to the database's `addObject` function, then NeoAccess will assign the object an ID unique to that database.

Running applications are often dynamic systems. The state of persistent objects change in response to user actions. NeoAccess provides a simple mechanism to manage change during execution. Member Functions that modify persistent attribute values use the `setDirty` function to indicate that the object's state in memory has changed and that that state needs to be committed to disk. NeoAccess keeps track of all dirty objects and commits changes all at once. This means that the contents of a NeoAccess database on disk is always consistent between one commit and the next. Some applications commit changes only when the user chooses the Save or Save As… menu items. Others implement a much more urgent and frequent policy.

A significant factor in limiting complexity in an application is the management of concurrency. One way to view concurrency control sees it as a combination of: 1) shared access, 2) serialization of change and 3) cooperation to accomplish a task. CNeoPersist includes two sets of member functions which facilitate shared access and serialized change. Getting software components to cooperate in order to finish a task is the charter of dependency mechanisms which are often provided by application frameworks and sophisticated collaborative computing constructs.

NeoAccess keeps track of which persistent objects the application has pointers to. This is done through the use of an object reference count in every persistent object. The static member functions `FindEvery` and `FindByID`, for example, add a reference to every object they find. Application code uses the `referTo` and `unrefer` member functions to keep this reference count consistent. When one component of an application passes a persistent object pointer to another component, the reference count of the object needs to be incremented accordingly. This is done by calling `referTo`. Conversely, references can be deleted by calling the `unrefer` function. NeoAccess will take care of deleting the object from memory once the last reference to it has been removed. You never use the `delete` operator on a persistent object.

An easier way of managing reference counting issues is to use a swizzler to point to a persistent object rather than a simple object pointer. A **swizzler** is an object which can be used to refer to another object in a database. A swizzler can be thought of as a "smart pointer." Using swizzlers instead of object pointers removes the need for application developers to explicitly add and remove object references to persistent objects. This is because the assignment operator of a swizzler removes its reference to the object it was referring to and adds a reference to the object being assigned to it. The swizzler destructor, which is even called when a C++ exception is thrown, also removes references as the swizzler goes out of scope.

Serializing change is necessary in order to avoid putting an object in an inconsistent state due to concurrent updating by two independent tasks. NeoAccess provides a simple mechanism for avoiding this type of inconsistency through the use of the `setBusy` and `setUnbusy` member functions. A task should check the busy state of the object before trying to modify its state. If the object is not busy, the task should then mark

the object busy to signal that it is in an inconsistent state during the update process. The object should be marked unbusy once the update is complete.

CNeoBusyFocus objects are used to manage the busy state of an object consistent in the event of a failure. A busy focus object records the busy state of an object before setting the object busy. The destructor of the focus, which is called even if a C++ exception is thrown, resets the object's busy state.

CNeoPersistNative

The core of NeoAccess was written to be environment-neutral so as to facilitate portability. NeoAccess portability is implemented in part through the measured use of environment-neutral and environment-specific classes.

Consider the class diagram for persistent classes in Laughs. CNeoPersist is an environment-neutral class. That means that the interface to CNeoPersist makes no assumptions about what environment it might be compiled or executed. However, providing a seamless integration of CNeoPersist into some application frameworks might require adding additional overrides beyond the general support provided in CNeoPersist.

Borland's ObjectWindows Library (OWL) application framework, for example, asks that all concrete subclasses of TObject override the `isA` function. One way to provide this support would be to compare an object's class ID with the given argument value. A basic implementation could do this simply by calling the object's `getClassID` function. This would eliminate the need for other persistent classes from also having to override `isA`.

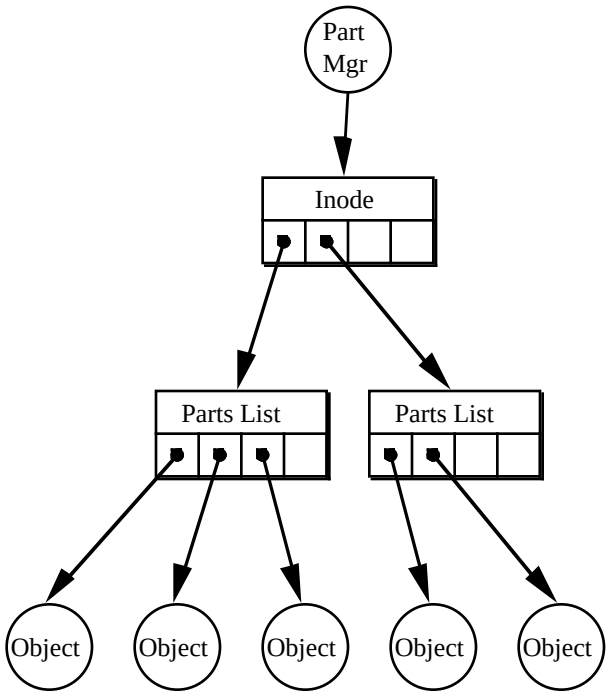
NeoAccess provides this intimate level of support in numerous environments by optionally defining environment-specific subclasses of CNeoPersist. The compile-time symbol `CNeoPersistNative` is defined to refer to this subclass. All other persistent classes are meant to be subclasses of CNeoPersistNative. In environments which don't require an environment-specific subclass, CNeoPersistNative is defined to be the same as CNeoPersist.

CNeoPartMgr

Objects in memory often refer to other objects. In order to be effective, a database engine needs to provide mechanisms for preserving these relationships between objects. NeoAccess includes several such mechanisms. A class called CNeoPartMgr can be used to group objects into a collection called a **part list**. Application-specific subclasses inherit this one-to-many grouping capability by deriving from CNeoPartMgr or by including an ENeoPartMgr data member in a persistent class definition.

CNeoPartMgr is in fact a "convenience class." It is simply a subclass of CNeoPersistNative which has a data member of type ENeoPartMgr and function overrides to manage the persistence of this data member. ENeoPartMgr is a class of object designed to be embedded as a data member of a CNeoPersist subclass. Part management responsibilities are delegated to this data member. For example, a sales processing application might include a persistent CProduct class which refers to the set of suppliers of this product and to the set of customers who have purchased this product. While CProduct could be implemented as a subclass of CNeoPartMgr, it would more likely be an immediate subclass of CNeoPersistNative with two ENeoPartMgr data members – one of which manages the set suppliers and the other the set of customers.

A part object can refer to zero or more subparts. These subparts may themselves be parts from which further subsubparts descend. This hierarchical structure, which is sometimes called a **containment hierarchy**, is incredibly useful construct to use in the design of an application. Indeed, one of the initial motivations for object database systems was just such a construct for use in CAD/CAM applications. In order to illustrate part lists and their use in an application, the personalities in Laughs descend from CNeoPartMgr.



Part Manager with Attached Part List

CPerson

CPerson is a subclass of CNeoPartMgr. CPerson is an **abstract** class, which means that no objects of this class are ever actually created in memory. Its purpose is to define those properties and responsibilities that all people have in common. All people, for example, have a name, an IQ and are able to pronounce their name on demand. Names can be a maximum of 255 characters and are stored in the instance variable `fName`. Taking an optimistic view of the world, everybody also has a skill, though CPerson leaves the definition of a particular person's skill to specific subclasses.

Every person maintains a reference to their father using a swizzler. As mentioned earlier, **swizzlers** are smart pointers. They are objects that act like pointers, but they're more powerful that standard C pointers in that they can be used to refer to objects that might not even be in memory.

Developers refer to the reference material in the NeoAccess Reference Manual for each of the superclasses to determine which functions of these superclasses need to be overridden. CPerson has a base class of CNeoPartMgr, which is itself a subclass of CNeoPersist. The sections titled "Subclassing CNeoPartMgr" and "Subclassing CNeoPersist" articulate the complete set of functions to be overridden.

An abbreviated definition of CPerson is as follows:

```
class CPerson : public CNeoPartMgr
{
public:
                        /** Instance Member Functions **/
                        CPerson(const CNeoString &aName = "",
                            const CNeoString &aFatherName = "",
                            const unsigned short aIQ = kDefaultIQ);
 virtual long           getFileLength(const CNeoFormat *aFormat) const;

                        /** Access Member Functions **/
 CPerson *              getFather(void);
 void                   getFatherName(CNeoString &aName);
 virtual Boolean        getValue(const NeoTag aTag, const NeoTag aType,
                            void *aValue);
 void                   printName(void) const;
 void                   setFatherName(const CNeoString &aName = "");
 virtual Boolean        setValue(const NeoTag aTag, const NeoTag aType,
                            const void *aValue);
 virtual void           skill(void) = 0;

                        /** I/O Member Functions **/
 virtual void           readObject(CNeoStream *aStream, const NeoTag aTag);
 virtual void           writeObject(CNeoStream *aStream, const NeoTag aTag);

                        /** Persistence Member Functions **/
 virtual Boolean        revert(void);
 virtual void           update(CNeoPersist *aObject);

                        /** Schema-Evolution Member Functions **/
 virtual NeoMark        convert(CNeoFormat *aOldFormat, CNeoFormat *aNewFormat);

                        /** Purge Member Functions **/
 virtual Boolean        purge(NeoSize *aNeeded) const;

protected:
 CNeoString             fName;
 TNeoIDSwizzler         fFather;
 CNeoString             fAddress;
 unsigned short         fIQ;
};

const NeoIDkPersonID= 25;
```

Note that `kPersonID` is the unique class ID for this class. All negative class IDs and those between 0 and 19 are reserved by NeoAccess for use by the core engine. Class IDs for an application should begin at 20. Class IDs can never exceed the value of `kNeoClasses` (which is defined in the header file for CNeoMetaClass). Some applications may need to increase the value of `kNeoClasses` in order to meet this requirement.

The implementation of the CPerson constructor is simplicity itself. It simply sets the state of the person according to the arguments of the constructor.

```
CPerson::CPerson(const CNeoString &aName, const CNeoString &aFatherName,
 const unsigned short aIQ)
{
 setName(aName);
 setAddress(kDefaultAddress);
 setIQ(aIQ);
 setFatherName(aFatherName);
}
```

NeoAccess includes a rather powerful and flexible streams mechanism which is similar to the *iostreams* library which is a standard part of most C++ development environments. The two i/o functions of CPerson, `readObject` and `writeObject`, are used primarily by NeoAccess to serialize data members. Each class which contains data member values which must be preserved, overrides these two member functions to read in and write out these values. The `readObject` function should also be overridden by classes which contain "transient" data members which must be initialized each time an object is read in from disk.

NeoAccess streams are "typed". This means that a datum is read in and written out according to its type. You would use a stream's `readLong` function to read a `long` integer and `writeShort` to write a `short` integer to the stream. There are a number of advantages to this approach, the most significant is that it allows issues such as buffering and byte-swapping to be encapsulated in the stream's implementation and away from the view of application developers.

CPerson's implementations of `readObject` and `writeObject` are as follows:

```
void CPerson::readObject(CNeoStream *aStream, const NeoTag aTag)
{
 CNeoFormat *      format       = aStream->fInputFormat;
 CNeoDebugIO       checker(aStream, FALSE, this);

 NeoAssert(format);
 NeoInherited::readObject(aStream, aTag);

 aStream->readNativeString(fName, sizeof(fName), pNeoName);
 if (format->getUserFormat() == kLaughsFormat2 ||
     format->getUserFormat() == kLaughsFormat4)
     aStream->readNativeString(fAddress, sizeof(fAddress), pAddress);
 else
     // If fAddress was not persistent in this version of the database
     // then we'll use the default value.
     fAddress = kDefaultAddress;
 if (format->getUserFormat() == kLaughsFormat3 ||
     format->getUserFormat() == kLaughsFormat4)
     fIQ = aStream->readShort(pIQ);
 else
     // If fIQ was not persistent in this version of the database
     // then we'll use the default value.
     fIQ = kDefaultIQ;

 fFather.readObject(aStream, pFather);
}

void CPerson::writeObject(CNeoStream *aStream, const NeoTag aTag)
{
 CNeoFormat *      format       = aStream->fOutputFormat;
 CNeoDebugIO       checker(aStream, TRUE, this);

 NeoAssert(format);
 NeoInherited::writeObject(aStream, aTag);

 aStream->writeNativeString(fName, sizeof(fName), pNeoName);
 if (format->getUserFormat() == kLaughsFormat2 ||
     format->getUserFormat() == kLaughsFormat4)
     aStream->writeNativeString(fAddress, sizeof(fAddress), pAddress);
 if (format->getUserFormat() == kLaughsFormat3 ||
     format->getUserFormat() == kLaughsFormat4)
      aStream->writeShort(fIQ, pIQ);

 fFather.writeObject(aStream, pFather);
}
```

The CNeoDebugIO object at the top of these two member functions is a debugging construct used to verify that the number of bytes read from and written to the stream is less than or equal to the value returned by the class's `getFileLength` function. This check is performed only when the `qNeoDebug` and `qNeoDebugIO` compile-time symbols are defined. Developers should always use CNeoDebugIO objects in the implementation of `readObject`/`writeObject` member functions of concrete classes.

In real applications the layout of objects on disk may change from release to release – some data members which used to be transient might become persistent, others which were persistent may no longer be, and so on. NeoAccess provides a powerful schema-evolution mechanism using CNeoFormat objects. Format objects contain enough information for the application to decide which data members are persistent in the given version of the database. By using format objects, developers can support the reading of older versions of their databases as well as converting databases between formats.

The implementation of CPerson is based on the following assumptions:

❖ There are 4 different user-formats,

- The `fName` data member is persistent in all user-formats,
- The `fAddress` member is persistent only in user-formats `kLaughsFormat2` and `kLaughsFormat4`,
- The `fIQ` member is persistent only in user-formats `kLaughsFormat3` and `kLaughsFormat4`

The implementations of the `readObject` and `writeObject` member functions of CPerson begin by giving the parent class, CNeoPartMgr, an opportunity to read/write its data members. Based on the format of the given stream, it then calls the stream's i/o functions to restore/preserve the persistent data members of this person. Note that the `readObject`/`writeObject` member functions of the `fFather` are called to restore/preserve the persistent state of the swizzler.

When you read the more complete discussions of streams elsewhere in this document you'll see that `readObject` and `writeObject` can also be used to read and write streams other than just file streams. The `aTag` value of the stream's i/o functions is given to facilitate the use of I/O streams that require these values. We'll ignore this issue in this tutorial other than to say that the tag value of each data member in a class should be unique within that class and its parent classes.

Virtually all the complexity of the `getFileLength` function is motivated by the desire to support multiple database versions, as described above.

```
long CPerson::getFileLength(const CNeoFormat *aFormat) const
{
 long     length       = NeoInherited::getFileLength(aFormat)
                             + sizeof(CNeoString)
                             + fFather.getFileLength(aFormat);

 switch (aFormat->getUserFormat()) {
 case kLaughsFormat1:
      break;

 case kLaughsFormat2:
      length += sizeof(CNeoString);
      break;

 case kLaughsFormat3:
      length += sizeof(unsigned short);
      break;

 case kLaughsFormat4:
      length += sizeof(CNeoString) + sizeof(unsigned short);
      break;

 default:
      NeoAssert(FALSE); // unknown format!
 }

 return length;
}
```

Object-oriented programming purists will tell you that all data members of a class should be protected. Some will even tell you that they should all be private. You should access data values using accessor functions. CNeoPersist includes a pair of member functions, `getValue` and `setValue`, which are the mother of all accessor functions.

The `getValue` function can be used to obtain the value of a data member of an object. The `aTag` argument is a tag which identifies the data member, and `aType` refers to the format in which the value is to be returned. The final argument to this function, `aValue`, is a pointer to a buffer into which the value should be returned. If `aType` is the same as the type of the data member specified by `aTag`, the direct assignment is performed. Otherwise, `ConvertType` is called to convert the data member's native type to the requested type. CPerson overrides this function to allow its data members to be accessible via this mechanism.

```
Boolean CPerson::getValue(const NeoTag aTag, const NeoTag aType, void *aValue)
{
 Boolean  result   = TRUE;

 switch (aTag) {
 case pNeoName:
      if (aType == kNeoNativeStringType)
           *(CNeoString *)aValue = fName;
      else
           ConvertType(&fName, kNeoNativeStringType, aValue, aType);
      break;

 case pAddress:
      if (aType == kNeoNativeStringType)
           *(CNeoString *)aValue = fAddress;
      else
           ConvertType(&fAddress, kNeoNativeStringType, aValue, aType);
      break;

 case pIQ:
      if (aType == kNeoUShortType)
           *(unsigned short *)aValue = fIQ;
      else
           ConvertType(&fIQ, kNeoUShortType, aValue, aType);
      break;

 default:
      result = NeoInherited::getValue(aTag, aType, aValue);
 }

 return result;
}
```

Just as `getValue` can be used to obtain the value of an abstract data member, the `setValue` function can be used to set a value. Once again, the `aTag` argument is a tag which identifies the data member, and `aType` refers to the format of the value as given by the `aValue` argument. If `aType` is the same as the native type of the data member specified by `aTag`, the direct assignment is performed; otherwise, `ConvertType` is used to perform any possible conversion.

```
Boolean CPerson::setValue(const NeoTag aTag, const NeoTag aType,
                              const void *aValue)
{
 Boolean  result   = TRUE;

 switch (aTag) {
 case pNeoName:
      if (aType == kNeoNativeStringType)
           fName = *(CNeoString *)aValue;
      else
          ConvertType(aValue, aType, &fName, kNeoNativeStringType);
      break;

 case pAddress:
      if (aType == kNeoNativeStringType)
          fAddress = *(CNeoString *)aValue;
      else
          ConvertType(aValue, aType, &fAddress, kNeoNativeStringType);
      break;

 case pIQ:
      if (aType == kNeoShortType)
          fIQ = *(short *)aValue;
      else
          ConvertType(aValue, aType, &fIQ, kNeoShortType);
      break;

 default:
      result = NeoInherited::setValue(aTag, aType, aValue);
 }

 return result;
}
```

NeoAccess automatically tracks changes to objects in memory. These changes may later be committed to disk or reverted using the database object's `commit` or `revert` member functions, respectively. An object's `revert` function is called during the reversion process so that it's state can be reinitialized and read back into memory from disk. CPerson overrides `revert` so that the `fFather` swizzler is also reverted.

```
Boolean CPerson::revert(void)
{
 return (Boolean)(fFather.revert() && NeoInherited::revert());
}
```

An object's `update` function is called to copy the state of one object to another. This is some times useful when transferring data from one database to another or when otherwise duplicating data. The `aObject` argument is assumed to be the same class of object as `this`. The immediate data members of CPerson assume the values found in `aObject` by overriding this function.

```
void CPerson::update(CNeoPersist *aObject)
{
 NeoInherited::update(aObject);

 fName = ((CPerson *)aObject)->fName;
 fAddress = ((CPerson *)aObject)->fAddress;
 fIQ = ((CPerson *)aObject)->fIQ;
 fFather.update(&((CPerson *)aObject)->fFather);
}
```

Most of the remaining member functions of CPerson are simple accessor functions. But a couple of them are interesting enough to note here.

The `getFather` function uses the `fFather` swizzler data member to return a pointer to the CPerson who is the father of this person. This function simply returns a doubly casted pointer. The first cast, to a CNeoPersistNative pointer, is necessary in order to invoke the casting operator of the swizzler object. The second cast, to a CPerson pointer, is needed in order to match the return value of the `getFather` function.

```
CPerson *CPerson::getFather(void)
{
 return (CPerson *)(CNeoPersistNative *)fFather;
}
```

The `printName` function uses the `neoPrintf` macro to print the person's name on `stdout`.

```
void CPerson::printName(void) const
{
 char str[256];

 sprintf(str, "Name is %s" NeoEOL, (char *)fName);
 neoPrintf(str);
}
```

It's probably also worthwhile to note that the `getClassID` and `New` member functions have not been overridden by CPerson. This is because only concrete subclasses of CNeoPersist need these overrides.

CJoker

CJoker and CClown are two **concrete** subclasses of CPerson. Concrete classes are those for which objects can be allocated during execution.

A joker is a type of person that is particularly good at telling jokes. (Actually, the jokes they tell aren't that funny!) The definition of CJoker is as follows:

```
const NeoID kJokerID = 26;

class CJoker : public CPerson
{
public:
                        CJoker(const CNeoString &aName = "",
                               const CNeoString &aFatherName = "");

 static CNeoPersist *New(void);
 NeoID                  getClassID(void) const;

                        /** Joker Member Functions **/
 void                   skill(void) const;
 void                   forgetJoke(const CJoke *aJoke);
 CJoke *                getJoke(const long aOffset) const;
 long                   getJokeCount(void) const {return getListCount();}
 void                   learnJoke(CJoke *aJoke);
};
```

The constructor passes its arguments on to its parent, CPerson. It also specifies that the default IQ of a joker is 150 – generous at best. As we saw earlier, CPerson is a subclass of CNeoPartMgr. Hence, it includes a part list data member. The constructor for CJoke specifies the base class of all subparts to be `kJokeID` (the class ID of joke objects) by calling `setObjClassID`.

```
CJoker::CJoker(const CNeoString &aName, const CNeoString &aFatherName)
 : CPerson(aName, aFatherName, 150)
{
 setObjClassID(kJokeID);
}
```

Concrete persistent classes override the `New`, `getClassID` and `getFileLength` member functions to create the proper type of object and return an object's class ID, respectively. CJoker doesn't override `getFileLength` because it doesn't have any additional persistent data members other than those inherited from CPerson.

```
CNeoPersist *CJoker::New(void)
{
 return NeoNew CJoker();
}

NeoID CJoker::getClassID(void) const
{
 return kJokerID;
}
```

Teaching a joker a new joke is accomplished by using the function `learnJoke`. (Teaching it a joke that is actually funny is beyond the scope of this tutorial!)

We already know that jokes are kept in the joker's part list. By default, parts are sorted in a list in ascending order by ID, though other types of part lists could be used instead to sort objects in some other order. Adding a joke to a joker's part list involves simply calling the joker's `addToList` function.

```
void CJoker::learnJoke(CJoke *aJoke)
{
 // Add the joke to this joker's part list
 addToList(aJoke);
}
```

Getting a joker to forget a joke is similar to the process of teaching it one. The `forgetJoke` function simply calls the joker's `deleteFromList` function. Note though that removing a joke from a joker's part list does not remove it from the database entirely. Jokes can be shared (read: stolen) by other jokers. Just because one joker decides to forget a joke doesn't mean that all other jokers must also do so.

```
void CJoker::forgetJoke(const CJoke *aJoke)
{
 // Remove the joke from this joker's part list
 deleteFromList(aJoke);

 // Note: This joke is still in the database!
 // To remove the joke completely, we would say...
 // if (fMark)
 //     gNeoDatabase->removeObject(joke);
}
```

The `skill` function for jokers tells a joke. The joke it tells is chosen at random from its vast repertoire, which it keeps track of using the part list of the joker object. The function begins by determining how many jokes are in the joker's part list. If you look at the implementation of `getJokeCount` , it simply calls CNeoPartMgr's `getListCount` function, which returns the number of entries in the part list. If it has any jokes to tell, it selects one by passing a random number between one and the number of jokes it knows to `getJoke`, which locates the joke object in the database and returns a pointer to it. After the joke is delivered we remove our reference to it.

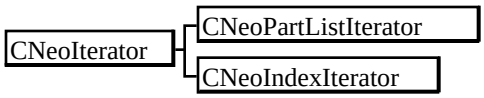
```
void CJoker::skill(void) const
{
 long count= getJokeCount();
 CJoke *joke;

 if (count) {
      // Randomly pick a joke
      joke = getJoke((rand()&0x7FFFFFFF) % count);
      NeoAssert(joke);

      neoPrintf("Tells jokes : ");
      joke->printJoke();

      // Don't forget to remove our reference to the joke.
      joke->unrefer();
 }
 else
      // This guy should probably retire.
      neoPrintf("Has no jokes to tell." NeoEOL);
}
```

Now let's take a look at what's involved in retrieving a joke from a joker's part list. The joker's `getJoke` function uses another powerful construct found in NeoAccess called an iterator. NeoAccess actually includes three iterator classes. The base class, CNeoIterator, provides the base capabilities for iterating over NeoAccess btrees in memory. A CNeoPartListIterator is used to traverse the joker's repertoire of jokes. We use the base part class's `getIterator` function to obtain such an iterator.



NeoAccess Iterator Classes

A newly initialized iterator is positioned by its constructor to just before the first object in the list (or immediately after the last object if the direction of the iterator is backwards). The `leap` function takes a signed value which indicates how many objects to move forward or backward in the list. After leaping to the proper position in the list, `getJoke` uses the iterator's `currentObject` function to obtain a pointer the object which the iterator now refers to. Be sure to note that `currentObject` does not add a reference to the object, so one needs to be added before a pointer to the object can be returned to the caller of `getJoke`.

```
CJoke *CJoker::getJoke(const long aOffset) const
{
 CJoke *               joke;
 CNeoPartListIterator *iterator     = getIterator();

 // Select the aOffset-th joke known to this CJoker.
 iterator->leap(aOffset);
 joke = (CJoke *)iterator->currentObject();

 // Iterators don't add references to objects. So we add one ourselves.
 if (joke)
     joke->referTo();
 delete iterator;

 return joke;
}
```

CJoke

Perhaps we've put the cart before the horse. Having looked at the powers of a joker, let's now go back and look at what a joke really is. Its abbreviated definition is as follows:

```
const NeoIDkJokeID      = 28;

class CJoke : public CNeoPersistNative
{
public:
                        CJoke(const char *aText = “”);

 static CNeoPersist *   New(void);
 virtual NeoID          getClassID(void) const;
 virtual long           getFileLength(const CNeoFormat *aFormat) const;

                        /** I/O Member Functions **/
 virtual Boolean        commit(CNeoContainerStream *aStream,
                            const Boolean aCompletely,
                            const Boolean aCompress, const Boolean aDeeply);
 virtual void           readObject(CNeoStream *aStream, const NeoTag aTag);
 virtual void           writeObject(CNeoStream *aStream, const NeoTag aTag);

                        /** Persistence Member Functions **/
 virtual void           add(void);
 virtual Boolean        getValue(const NeoTag aTag, const NeoTag aType,
                            void *aValue);
 virtual void           remove(void);
 virtual Boolean        revert(void);
 virtual Boolean        setValue(const NeoTag aTag, const NeoTag aType,
                            const void *aValue);
 virtual void           update(CNeoPersist *aObject);

                        /** Joke Management Member Functions **/
 void                   getJoke(char *aText) {aText = (char*)fJoke.getBlob();}
 long                   getJokeLength(void) const {return fJoke.getLength();}
 void                   printJoke(void) const;
 void                   setJoke(const char *aText) {fJoke = aText;}

protected:
 ENeoString             fJoke;
};
```

Some persistent classes have string data members. These string values are preserved and restored using a stream’s `readString` and `writeString` member functions. The CNeoString class can be used to manage a “native” string. In most environments, a native string is simply a C string, but on the Macintosh it is a Pascal string. (The first byte of a Pascal string is the length, followed the string itself. Note: Pascal strings are not necessarily null-terminated.)

The problem with strings and CNeoStrings is that the amount of space these data members occupy, both in memory and in the database, is fixed regardless of the string’s current length. This is an inefficient use of space and limits the maximum length that these strings can be. The common solution to this dilemma is to use a `char *` data member which refers to a non-contiguous block of memory to contain the string. The logistics associated with managing this discontiguous string value can be problematic.

NeoAccess includes an ENeoString class which is used to manage a persistent variable-length string value much like a `char *` data member would in a non-persistent environment. The string managed by ENeoString objects occupy a variable amount of space in the file and in memory depending on the string’s length. ENeoString is a subclass of ENeoBlob, which is the NeoAccess class that manages non-object persistent data.

CJoke has only one data member specific to this class, `fJoke`, which is an ENeoString. CJoke is a concrete persistent class. As such, it is a derivative of CNeoPersistNative, and overrides the `New`, `getClassID` and `getFileLength` member functions to create the proper type of objects and return the persistence particulars for the class, respectively. Having seen the implementations of these member functions in the classes we’ve seen earlier, the CJoke implementations should come as no surprise.

```
CNeoPersist *CJoke::New(void)
{
 return NeoNew CJoke();
}

NeoID CJoke::getClassID(void) const
{
 return kJokeID;
}

long CJoke::getFileLength(const CNeoFormat *aFormat) const
{
 return NeoInherited::getFileLength(aFormat) + fJoke.getFileLength(aFormat);
}
```

Unfortunately, C++ handles delegation less elegantly than it does inheritance. That is, a data member can override a function of the class that contains it. The data member's member function needs to be called explicitly. As such, CJoke overrides the `commit` function so that `fJoke`'s `commit` function gets called. The `add`, `remove`, `revert`, `update`, and `verify` member functions are overridden for the same reason.

```
Boolean CJoke::commit(CNeoContainerStream *aStream, const Boolean aCompletely,
 const Boolean aCompress, const Boolean aDeeply)
{
 Boolean  moved    = fJoke.commit(aStream, aCompletely, aCompress);

 moved |= NeoInherited::commit(aStream, aCompletely, aCompress, aDeeply);

 return moved;
}

void CJoke::add(void)
{
 NeoInherited::add();
 fJoke.add();
}

void CJoke::remove(void)
{
 fJoke.remove();
 NeoInherited::remove();
}

Boolean CJoke::revert(void)
{
 return (Boolean)(fJoke.revert() && NeoInherited::revert());
}

void CJoke::update(CNeoPersist *aObject)
{
 NeoInherited::update(aObject);
 fJoke.update(&((CJoke *)aObject)->fJoke);
}

const void *CJoke::verify(const void *aValue) const
{
 fJoke.verify(aValue);

 return NeoInherited::verify(aValue);
}
```

The `getValue` and `setValue` member functions are overridden to provide symbolic access to the `fJoke` data member. Their implementations are as expected:

```
Boolean CJoke::getValue(const NeoTag aTag, const NeoTag aType, void *aValue)
{
 Boolean      result   = TRUE;
 ENeoBlob *   blob;

 if (aTag == pJoke) {
      if (aType == kNeoBlobType)
           *(ENeoBlob **)aValue = &fJoke;
      else {
           blob = &fJoke;
           ConvertType(&blob, kNeoBlobType, aValue, aType);
      }
 }
 else
      result = NeoInherited::getValue(aTag, aType, aValue);

 return result;
}

Boolean CJoke::setValue(const NeoTag aTag, const NeoTag aType, const void *aValue)
{
 Boolean      result   = TRUE;
 ENeoBlob *   blob;

 if (aTag == pJoke) {
      if (aType == kNeoBlobType)
           fJoke.update(*(ENeoBlob **)aValue);
      else {
           blob = &fJoke;
           ConvertType(aValue, aType, &blob, kNeoBlobType);
      }
 }
 else
      result = NeoInherited::setValue(aTag, aType, aValue);

 return result;
}
```

As we saw in the implementation of CPerson, the `readObject` and `writeObject` member functions of CJoke begin by calling `NeoInherited`, then read/write its single data member, the text of the joke.

```
void CJoke::readObject(CNeoStream *aStream, const NeoTag aTag)
{
 CNeoDebugIO   checker(aStream, FALSE, this);

 NeoInherited::readObject(aStream, aTag);
 fJoke.readObject(aStream, pJoke);
}

void CJoke::writeObject(CNeoStream *aStream, const NeoTag aTag)
{
 CNeoDebugIO   checker(aStream, TRUE, this);

 NeoInherited::writeObject(aStream, aTag);
 fJoke.writeObject(aStream, pJoke);
}
```

All other member functions of the CJoke class are everything you would expect them to be.

CClown

Clowns are every bit as entertaining as jokers. Their unique skill is throwing pies. The class is defined something like this:

```
const NeoID kClownID    = 27;

class CClown : public CPerson
{
public:
                        /** Instance Member Functions **/
                        CClown(const CNeoString &aName = "",
                            const CNeoString &aFatherName = "",
                            const long aShoeSize = 15);

 static CNeoPersist *   New(void);
 virtual NeoID          getClassID(void) const;
 virtual long           getFileLength(const CNeoFormat *aFormat) const;

                        /** I/O Member Functions **/
 virtual void           readObject(CNeoStream *aStream, const NeoTag aTag);
 virtual void           writeObject(CNeoStream *aStream, const NeoTag aTag);
 virtual void           update(CNeoPersist *aObject);

                        /** Clown Member Functions **/
 CPie *                 bakePie(const char *aFilling = "Custard");
 CPie *                 getPie(const long aIndex);
 long                   getPieCount(void) const {return getListCount();}
 long                   getShoeSize(void) const {return fShoeSize;}
 virtual Boolean        getValue(const NeoTag aTag, const NeoTag aType,
                            void *aValue);
 void                   setShoeSize(const long aShoeSize)
                            {fShoeSize = aShoeSize;}
 virtual Boolean        setValue(const NeoTag aTag, const NeoTag aType,
                            const void *aValue);
 virtual void           skill(void);
 void                   throwPie(CPie *aPie);

protected:
 long                   fShoeSize;
};
```

The constructor passes its arguments on to its parent, CPerson. Clowns have a default IQ of 120. (Clowns are good cheaters!)

```
CClown::CClown(const CNeoString &aName, const CNeoString &aFatherName,
 const long aShoeSize)
 : CPerson(aName, aFatherName, 120)
{
 fShoeSize = aShoeSize;
 setObjClassID(kPieID);
}
```

Once again, the `New`, `getClassID` and `getFileLength` member functions create the proper type of objects and return the persistence particulars for the class, respectively. The implementations of these functions, shown below, are no surprise.

```
CNeoPersist *CClown::New(void)
{
 return NeoNew CClown();
}

NeoID CClown::getClassID(void) const
{
 return kClownID;
}

long CClown::getFileLength(const CNeoFormat *aFormat) const
{
 return NeoInherited::getFileLength(aFormat) + sizeof(fShoeSize);
}
```

The `skill` function causes a pie to be thrown.

```
void CClown::skill(void) const
{
 printf(“Throws %s pies\n”, fPietype);
}
```

And they said that you couldn’t teach an old clown new tricks. The `skill` function of clowns picks a pie at random form the clown’s arsenal and “throws” it. The `getPie` function uses a part list iterator to locate the pie in the list. A pie can obviously only be thrown once. So throwing it involves removing it from the clown’s pie part list and from the database.

```
void CClown::skill(void)
{
 long     count    = getPieCount();
 CPie *   pie;

 if (count) {
      // Randomly pick a pie
      pie = getPie((rand()&0x7FFFFFFF) % count);
      NeoAssert(pie);

      neoPrintf("Throws pies: ");
      throwPie(pie);

      // Don't forget to remove our reference to the pie.
      pie->unrefer();
 }
 else
      // This guy should probably retire.
      neoPrintf("Has no pies to throw." NeoEOL);
}

CPie *CClown::getPie(const long aIndex)
{
 CPie *                pie;
 CNeoPartListIterator *iterator     = getIterator();

 iterator->leap(aIndex);
 pie = (CPie *)iterator->currentObject();

 // Iterators don't add references to objects. So we add one ourselves.
 if (pie)
      pie->referTo();
 delete iterator;

 return pie;
}

void CClown::throwPie(CPie *aPie)
{
 char     Filling[kMaxFillingName];
 char     str[256];

 // Remove the pie from this clown's part list
 deleteFromList(aPie);

 // Now remove it from the database completely.
 gNeoDatabase->removeObject(aPie);

 // Throw it!
 aPie->getFilling(Filling);
 sprintf(str, "Here's a %s pie in your face!" NeoEOL, Filling);
 neoPrintf(str);
}
```

CPie

The definition and implementation of CPie is every bit as straightforward as one would expect of a concrete persistent class. The class is defined as follows:

```
class CPie : public CNeoPersistNative
{
public:
                        /** Instance Member Functions **/
                        CPie(const char *aFilling = "Custard");

 static CNeoPersist *   New(void);
 virtual NeoID          getClassID(void) const;
 virtual long           getFileLength(const CNeoFormat *aFormat) const;

                        /** Access Member Functions **/
 virtual Boolean        getValue(const NeoTag aTag, const NeoTag aType,
                             void *aValue);
 virtual Boolean        setValue(const NeoTag aTag, const NeoTag aType,
                             const void *aValue);

                        /** I/O Member Functions **/
 virtual void           readObject(CNeoStream *aStream, const NeoTag aTag);
 virtual void           writeObject(CNeoStream *aStream, const NeoTag aTag);
 virtual void           update(CNeoPersist *aObject);

                        /** Pie Member Functions **/
 void                   getFilling(char *aText) const
                             {strncpy(aText, fFilling, kMaxFillingName);}
 void                   setFilling(const char *aText)
                             {strncpy(fFilling, aText, kMaxFillingName);}

protected:
 char                   fFilling[kMaxFillingName];
};
```

CPie is a derivative of CNeoPersistNative, and overrides the `New`, `getClassID` and `getFileLength` member functions to create the proper type of objects and return the persistence particulars for the class, respectively. The implementations of these member functions are as follows.

```
CNeoPersist *CPie::New(void)
{
 return NeoNew CPie();
}

NeoID CPie::getClassID(void) const
{
 return kPieID;
}

long CPie::getFileLength(const CNeoFormat *aFormat) const
{
 NeoUsed(aFormat);

 return NeoInherited::getFileLength(aFormat) + kMaxFillingName;
}
```

The `readObject` and `writeObject` member functions of CPie are as one would expect; call `NeoInherited` then read/write its data member.

```
void CPie::readObject(CNeoStream *aStream, const NeoTag aTag)
{
 CNeoDebugIOchecker(aStream, FALSE, this);

 NeoInherited::readObject(aStream, aTag);
 aStream->readString(fPieType, sizeof(fPieType));
}

void CPie::writeObject(CNeoStream *aStream, const NeoTag aTag)
{
 CNeoDebugIOchecker(aStream, TRUE, this);

 NeoInherited::writeObject(aStream, aTag);
 aStream->writeString(fPieType, sizeof(fPieType));
}
```

The Laughs Application Class

Now that the introductions are complete, let’s look at the implementation of the CLaughsApp class.

The Constructor

```
CLaughsApp::CLaughsApp(void)
{
 CNeoMetaClass *   meta;

 // Note our file type so that we can be particular in get file dialog.
 FFileType = kLaughsFileType;

 fFormat0 = nil;
 fFormat1 = nil;
 fFormat2 = nil;
 fFormat3 = nil;

 // Add the application-specific metaclasses to the metaclass table.
 (void)NeoNew CNeoMetaClass(kNeoIDListID, kNeoNullClassID, "CNeoIDList",
      NeoNewGetOnePersist(CNeoIDList::New),
      NeoNewKeyManager(CNeoIDList::KeyManager));
 (void)NeoNew CNeoMetaClass(kPersonID, kNeoPersistID, "CPerson",
      NeoNewGetOnePersist(CPerson::New));
 (void)NeoNew CNeoMetaClass(kJokerID, kPersonID, "CJoker",
      NeoNewGetOnePersist(CJoker::New));
 (void)NeoNew CNeoMetaClass(kJokeID, kNeoPersistID, "CJoke",
      NeoNewGetOnePersist(CJoke::New));
 (void)NeoNew CNeoMetaClass(kClownID, kPersonID, "CClown",
      NeoNewGetOnePersist(CClown::New));
 (void)NeoNew CNeoMetaClass(kPieID, kNeoPersistID, "CPie",
      NeoNewGetOnePersist(CPie::New));

 meta = CNeoMetaClass::GetMetaClass(kPersonID);
 meta->addKey(kNeoNativeStringIndexID, pNeoName, kNeoNullClassID, TRUE);

 // Let the games begin.
 printf("Start Laughing...\n\n");
}
```

The constructor of the Laughs application begins with the allocation of metaclass objects for the persistent classes of the application. The metaclasses for CNeoPersist and CNeoIDIndex, two persistent classes used by virtually all NeoAccess-based applications, are created by the constructor of the base NeoAccess application, CNeoApp.

A **metaclass** is a class that describes other classes. Some object developers would call a metaclass a **factory class**. Each metaclass being allocated in Laughs describes important properties of an application-specific persistent class. The first argument to the metaclass constructor is the class ID, followed by the class ID of its parent's class. The name of the class is also given, as is a pointer to the `New` function for the class. The `NeoNewGetOnePersist` and `NeoNewKeyManager` macros need to be used in some runtime environments in order to convert a function pointer to what is sometimes called a **universal proc pointer**, or **UPP**. A UPP is a pointer to a subroutine that resides in a DLL, code fragment or shared library. If NeoAccess is statically linked into your application or if the environment you're developing in does not make a distinction between function pointers and UPPs, then you may not need to use these macros.

It's very important that a metaclass object be allocated for every persistent class in an application. This would seem to be a reasonable request given the straightforward nature of their construction. There's no need to keep pointers to the newly created metaclasses; they are automatically inserted in a metaclass table by their constructor.

Note that a deep secondary index key is added to the CPerson class. This is done by calling its metaclass's `addkey` function. The first two arguments to `addKey` is the class ID of the collection leaf class and the tag of the data member to be sorted on – `kNeoNativeStringIndexID` and `pNeoName`, respectively. CPerson's `getValue` function is called with this tag to obtain the sort key value.

If a third argument is given to `addKey`, it must either be `kNeoNullClassID` or the class ID of some base class. The third argument of this call is `kNeoNullClassID`. This is an example of a consolidated index. A **consolidated index** is one which contains objects having a common base class but whose leaf class may differ. All concrete person objects, be they jokers or clowns, are all sorted together in a single index in ascending order by name. See the discussion of “Consolidated Indices” in the Preliminaries section and the CNeoMetaClass’s “Adding to the Metaclass Table” discussion for more information on consolidated indices.

The end result of this metaclass configuration is that, at least initially, every joker and clown object added in the database will be indexed primarily by ID (which is the default), while the secondary consolidated index will sort all CPerson subclasses in the same index by name. As we will see later in this tutorial, it’s possible to dynamically change the indexing of objects in the database at runtime.

Creating a Document

Laughs’ `createDocument` function is called to create a new document. NeoAccess’s environment-specific support in other frameworks may include a different member function which performs a function very similar to the `createDocument` function found in the PowerPlant implementation. Most often this is because the application is just starting up or because the user has selected the New… menu item. Your application should include a `createDocument` (or compliment) function very much like this one. (The `OnCreate` function of MFC-based application classes serves a similar purpose.)

The function begins by setting up a `NEOTRYTO` block to capture and recover from any failures which might occur as the document is allocated or initialized. The `NEOCLEANUP` handler for this `NEOTRYTO` block cleans up before propagating the failure to the next handler on the failure stack. The default handler usually displays a dialog box indicating that the command could not be completed because of an error.

The arguments passed to the document’s constructor indicate whether the document supports printing, whether the database being opened is a new database and whether it is a remote database. The last argument is reserved for NeoShare support and should be FALSE for our purposes.

Note that the NeoAccess database object is allocated by the constructor of CNeoDocNative, not by any application-specific code.

Another interesting thing to note is that the construction of a database object is not the same as creating an operating system file. In order to clarify this distinction, let’s consider for a moment the standard user experience when creating a new word processing document... The user begins by launching the application, opening a new document window and starting to type. It’s only after enough of the document has been typed and the user worries about losing it to a system crash that the Save As… menu item is chosen. It’s at this point that a file on disk is created and the persistent objects that have been added to the database are committed. NeoAccess supports this user experience by allowing developers to add, remove and search for objects in a database object which is not yet open or even specified on the disk.

The only thing left to is to call the document’s `newDatabase` function to initialize the newly created document and database.

```
CNeoDoc *CLaughsApp::createDocument(void)
{
 CLaughsDoc *      document = nil;

 NEOTRYTO {
      const Boolean printable    = FALSE;
      const Boolean newFile      = TRUE;
      const Boolean remote       = FALSE;

      // Create your document.
      //
      // The arguments indicate whether the document is...
      //  printable (FALSE),
      //  a new file (TRUE),
      //  remote (FALSE).
      document = new CLaughsDoc(printable, newFile, remote);
      NeoFailNil(document);
      document->SetSuperCommander(this);

      // Call the document's newDatabase function.
      document->newDatabase();
 }
 NEOCLEANUP {
      // This exception handler gets executed if a failure occurred
      // anywhere within the scope of the above NEOTRY block. Since
      // this indicates that a new doc could not be created, we
      // check if an object had been allocated and if it has, delete
      // it. The exception will propagate up to the next exception
      // handler, which displays an error alert.
      if (document &&
           !document->getDatabase()->isOpen()) {
           delete document;
           document = nil;
      }
 }
 NEOENDTRYTO;

 return document;
}
```

Opening an Existing Database

CNeoLaughsApp `openDocument` function is called to open an existing database. As we discussed in the discussion of CNeoLaughsApp's `createDocument` function, NeoAccess's environment-specific support in other frameworks may include a different member function which performs a function very similar to the `openDocument` function found in the PowerPlant implementation.Your application should include an `openDocument` (or compliment) function very much like this one. (The `OnOpen` function of MFC-based application classes serves a similar purpose.)

The function begins by setting up a `NEOTRYTO` block to capture and recover from any failures which might occur as the document is allocated or initialized. The `NEOCLEANUP` handler for this `NEOTRYTO` block cleans up before propagating the failure to the next handler on the failure stack. As was the case earlier, the default handler usually displays a dialog box indicating that the command could not be completed because of an error.

The arguments passed to the document's constructor indicate whether the document supports printing, whether the database being opened is a new database and whether it is a remote database. The last argument is reserved for NeoShare support and should be FALSE for our purposes. Note that the NeoAccess database object is allocated by the constructor of CNeoDocNative, not by any application-specific code.

Once again, note the difference between the construction of a database object and the opening an operating system file. It is only after the document object is created that the database is opened using the document's `openFile` function.

```
void CLaughsApp::OpenDocument(FSSpec *aSpec)
{
 Boolean           remote;
 CLaughsDoc *      document     = nil;

 VOLATILE(document);

 NEOTRYTO {
     const Boolean printable    = TRUE;
     const Boolean newFile      = FALSE;

     // Our parent will check to see if this is a document we have
     // already opened. If we find the file already open by the app,
     // then fDocument will refer to it. If the file is opened but
     // not by this app, then the parent will signal a failure.
     NeoInherited::OpenDocument(aSpec);
     if (fDocument)
          return;

     remote = FALSE;

     // Create your document.
     document = new CLaughsDoc(printable, newFile, remote);
     NeoFailNil(document);
     document->SetSuperCommander(this); // PowerPlant logistics

     // Call the document's openFile function. The document will open
     // a window, open the file specified in the file specification
     // record, and display it in its window.
     document->openFile(aSpec);
 }
 NEOCLEANUP {
     if (document &&
          !document->getDatabase()->isOpen())
          delete document;
 }
 NEOENDTRYTO;
}
```

The Laughs Document Class

In application frameworks that support them, it is the charter of the document class to consolidate the various information sources that the document's clients need to operate properly. In practice, this is a rather broad charter. Most documents have only a single database to work with. But the document classes of many new Macintosh applications also manage Publish & Subscribe editions as well. Microsoft's OLE 2 (Object Linking and Embedding) and Apple's OpenDoc push the evolution of document classes further still.

Some frameworks don't include a document class. Regardless of the support for documents provided by the application framework, NeoAccess includes a document class to manage most of the issues associated with opening, closing and otherwise manipulating a NeoAccess database.

It's interesting to note that applications built using NeoAccess rely much more heavily on the services of the database class, CNeoDatabase, than do applications which simply "inhale" the entire contents of a file as it is opened and "exhale" it back out in response to a Save command. NeoAccess applications bring objects into memory on demand and purge them when memory reserves are low. A NeoAccess database is kept open for as long as objects from that database are being accessed. And updating a NeoAccess database doesn't

necessarily involve rewriting the entire database; only those objects that have changed in memory need to be written.

```
const OSType kLaughsFileType= 'Ne6d';

class CLaughsDoc : public CNeoDocRoot {
public:
                        CLaughsDoc(const Boolean aPrintable,
                             const Boolean aNewFile, const Boolean aRemote);

 virtual void           buildWindow(void);
 virtual void           newDatabase(void);
 virtual void           openFile(FSSpec *aSpec);

 void                   createObjects(void);
 void                   printOut(void);
};
```

The constant `kLaughsFileType` is used, in combination with the application's signature, by the Macintosh Finder in choosing the appropriate icon for the document file. These values are ignored in most other environments.

The constructor and `buildWindow` member functions of the Laughs document class are trivial. The constructor simply calls the parent's constructor which initializes the document and database object. Most applications will include a more substantial `buildWindow` function which creates and initializes a window object in which to present the contents of the document. Laughs is such a simple application that it doesn't explicitly create a window. Instead, a text window is created automatically the first time `printf` is called.

The real meat of the Laughs document class is implemented in its `newDatabase`, `openFile`, `createObjects` and `printOut` member functions. Let's take a look at those now.

Creating a New Database

Earlier in this tutorial we saw that in PowerPlant the document's `newDatabase` function is called by the application's `createDocument` function immediately after creating the document. The parent class's `newDatabase` function is given an opportunity to do its thing before the file is actually opened. Then it specifies a default location for the database on disk. The way in which a database is specified is often environment-specific. Each framework and operating system provides different interfaces for specifying the file system location. The PowerPlant database class, CNeoDatabasePP, includes a function, `setFileSpec`, which takes a Macintosh file spec. The name of the Laughs database file is initially "Laughter", though the user can change it when the standard file dialog box is presented. A volume reference number and directory ID of zero refers to the directory containing the Laughs application. Again, other environments might handle the specification of a file location by different means.

Having opened the file with the proper permissions, the `createObjects` function is called to add objects to the database.

```
void CLaughsDoc::newDatabase(void)
{
 CNeoDatabase *    database     = getDatabase();
 NeoFileSpec       spec;

 fNewDatabase = TRUE;
 fOpenMode = NeoReadWritePerm;
 NeoInherited::newDatabase();

 // Set the default name which appears in the get file dialog.
 spec.vRefNum = 0;
 spec.parID = 0;
 NeoStringCopy("\pLaughter", spec.name);
 database->setFileSpec(spec);

 // Stuff the database with objects.
 createObjects();

 // Commit the changes that we made to the database.
 DoSave();

 // That's all folks!
 printf("\nDone!\n\n");
}
```

NeoAccess uses global variables sparingly. However the global `gNeoDatabase` must always be set to refer to “the current database”. The document sets this value in its constructor. An application which has only one database open at a time can always be assured of this variable being set properly. (While Laughs is, in fact, this simple, most *real* applications are not.) In frameworks which use document classes, the NeoAccess document class often sets this value when a document is activated. Developers should make sure that it is set properly in their applications.

Opening an Existing Database

Earlier in this tutorial we saw that when developing using PowerPlant the document’s `openFile` function is called by the application’s `openDocument` function immediately after creating the document object. After permissions are set the parent class’s `openFile` function is given an opportunity to do its thing before the file is actually opened. This function is passed the file specification `aSpec`. Having opened the file with the proper permissions, the `printOut` function is called to display objects in the database.

```
void CLaughsDoc::openFile(FSSpec *aSpec)
{
 CNeoDatabase *    database     = getDatabase();

 fOpenMode = NeoReadPerm;

 NeoInherited::openFile(aSpec);

 // Find each of the objects in the database in turn.
 printOut();

 // That's all folks!
 printf("\nDone!\n\n");
}
```

Adding Objects to the Database

OK, let's get some work done. The `newDatabase` function of the Laughs document class creates a new database object and then populates the database by calling the document's `createObjects` function. The `createObjects` implementation is long, so let's look at it one piece at a time. We begin by telling the user what we're about to do...

```
// Tell them what we're about to do.
database->getName(string);
if (!string[0])
     string = "Untitled";

sprintf(str,"Storing 4 Jokers and 3 Clowns in \"%s\"." NeoEOL, (char *)string);
neoPrintf(str);
```

Adding an object to the database is pretty straightforward. Simply create the joke in memory, then call the database's `addObject` function. We add four jokes in a row...

```
// Each joke is created and added to the database.
// Note: An object ID is assigned automatically by addObject.

// Know any good jokes? How 'bout this one...
joke1 = NeoNew CJoke("The world's shortest poem: Flees. Adam had'em.");
database->addObject(joke1);

// Is this a joke???
joke2 = NeoNew CJoke("My dog's got no nose?");
database->addObject(joke2);

// OK, two more...
joke3 = NeoNew CJoke("Cogito ergo spud - I think therefore I yam");
database->addObject(joke3);
joke4 = NeoNew CJoke("And so I said to the guy...");
database->addObject(joke4);
```

Having defined the overall schtick, we in turn create three joker objects, teach each of them a repertoire of jokes and add them to the database. Teaching a joker a joke is easy, just call `learnJoke`. As we complete our work with a particular joke we're careful to delete our reference to the joker by using it's `unrefer` function. We do the same thing with the joke objects once all the jokers have been educated.

NeoAccess persistent objects are created just as all C++ objects are, by using the `new` operator. Each joker object is initialized with its name and its father's name. Having done that, the object is added to the database using the database's `addObject` function. Note that adding joker objects, which have multiple indices and part lists, is just as easy as adding the more basic joke and pie objects. What could be easier?

```
// Create a joker object.
 joker = NeoNew CJoker("Adam");

 // Teach it a couple of jokes.
 joker->learnJoke(joke1);
 joker->learnJoke(joke2);

 // Add it to the database.
 database->addObject(joker);
 // Don't need this guy any more. Remove our reference to it.
 joker->unrefer();
 joker = nil;

 // Another joker
 joker = NeoNew CJoker("John", "Adam");
 joker->learnJoke(joke3);
 joker->learnJoke(joke4);
 database->addObject(joker);
 joker->unrefer();
 joker = nil;

 // This one steals others' jokes
 joker = NeoNew CJoker("Steve", "John");
 joker->learnJoke(joke1);
 joker->learnJoke(joke2);
 joker->learnJoke(joke3);
 joker->learnJoke(joke4);
 database->addObject(joker);
 joker->unrefer();
 joker = nil;

 // Create another joker.
 joker = NeoNew CJoker("Harry", "Adam");

 // Add it to the database.
 database->addObject(joker);

 // This guy steals jokes.
 joker->learnJoke(joke2);

 // Remove our reference to the joker and the jokes.
 joker->unrefer();
 joker = nil;
 joke1->unrefer();
 joke1 = nil;
 joke2->unrefer();
 joke2 = nil;
 joke3->unrefer();
 joke3 = nil;
 joke4->unrefer();
 joke4 = nil;
```

Now we're going to add a few clowns to the database and arm them with pies. The process is pretty much the same as with jokers. A clown's `bakePie` function creates a CPie object, adds it to the database and then returns a reference to the caller. In this particular use of `bakePie` we don't actually need the pie reference so we remove it using `unrefer`. We need to do this to all references to persistent objects so that the NeoAccess garbage collector know that we're not referring to these objects any more.

```
// Create a clown.
clown = NeoNew CClown("Fred", "John");

// Add it to the database.
database->addObject(clown);

// Build up its arsenal.
pie = clown->bakePie("Jello");
pie->unrefer();
pie = clown->bakePie("Marshmellow");
pie->unrefer();
pie = clown->bakePie("Custard");
pie->unrefer();
pie = clown->bakePie("Cool Whip®");
pie->unrefer();
pie = clown->bakePie("Yogurt");
pie->unrefer();

// Remember to remove our reference when we're done.
clown->unrefer();
clown = nil;

// Create another clown.
clown = NeoNew CClown("Donald", "Steve");
database->addObject(clown);

// Six pies all the same.
for (i = 1; i <= 6; i++) {
     pie = clown->bakePie("Lemon-meringue");
     pie->unrefer();
}
// Remove our reference
clown->unrefer();
clown = nil;

// And another clown.
clown = NeoNew CClown("Eve");
database->addObject(clown);

// Four pies all the same.
for (i = 1; i <= 4; i++) {
     pie = clown->bakePie("Banana cream");
     pie->unrefer();
}
// Remove our reference
clown->unrefer();
clown = nil;
}
```

NeoAccess makes a distinction between changing the state of a database in memory and updating the database on disk to reflect those changes. The `addObject` function marks each object dirty so that its state is saved to disk when changes are committed. These changes are committed by the caller of `createObjects` by using the database's `commit` function.

Notice how simple an application built using NeoAccess can be. There are absolutely no database administration tasks to worry about. Application code doesn’t need to keep track of how objects are indexed, which ones are dirty, or where in the database an object is actually located. NeoAccess takes care of all the details so that developers can focus on the fun stuff.

Locating Objects in a Database

When the Laughs database is initially created, all the CPerson, CJoker, and CClown objects are indexed on the `fName` field. However, all objects of a particular class are sorted separately from all objects of other classes (e.g., CJokers are sorted separately from CClowns). Unfortunately, this does not allow us to iterate through all objects having a base class of CPerson in the alphabetical order. It's also possible in NeoAccess to create a consolidated index in which CPersons, CJokers and CClowns are sorted together alphabetically by `fName`. Moreover, such an index (as well as any other NeoAccess index) can be created dynamically at run time.

After printing a message indicating the name of the database and how many objects it contains, `printOut` changes the indexing of the database then uses the new indexing to iterate over all CPerson objects in the database by name. The evolutionary change in the way in which objects in this database are organized illustrates one aspect of NeoAccess's schema evolution support.

```
void CLaughsDoc::printOut(void)
{
 CPerson *                  person;
 CPerson *                  father;
 CNeoNativeStringSelect     key(pNeoName, "");
 CNeoDatabase *             database     = getDatabase();
 CNeoIterator *             iterator;
 CNeoMetaClass *            meta;
 CNeoString                 string;
 char                       str[256];

 // Tell them what we're about to do.
 database->getName(string);
 sprintf(str, "Restoring %ld Jokers and %ld Clowns from \"%s\"." NeoEOL,
      database->getObjectCount(kJokerID, FALSE),
      database->getObjectCount(kClownID, FALSE), (char *)string);
 neoPrintf(str);

 meta = CNeoMetaClass::GetMetaClass(kPersonID);
 // Remove the existing non-consolidated indices from the metaclass table.
 meta->removeKeyByID(kNeoNativeStringIndexID, pNeoName, TRUE);
 // Add the new consolidated index to the metaclasses table.
 meta->addKey(kNeoNativeStringIndexID, pNeoName, kPersonID, TRUE);
 // Ask database to update indices to correspond to the metaclasses table.
 database->updateIndices();

 key.setMatchAll(TRUE);
 // We want to iterate through all objects having base class of CPerson
 // in the alphabetical order.
 iterator = database->getIterator(kPersonID, &key, TRUE);
 person = (CPerson *)iterator->currentObject();
 while(person) {
      person->autoReferTo();
      person->printName();
      father = person->getFather();
      if (father) {
           neoPrintf("Father's ");
           father->printName();
      }
      else
           neoPrintf("This is an orphan" NeoEOL);
      person->skill();
      neoPrintf(NeoEOL);
      person->autoUnrefer();
      person = (CPerson *)iterator->nextObject();
 }

 delete iterator;
}
```

The clarity of Laughs has allowed us study the steps a developer must take in order to build a very simple application that uses NeoAccess. This tutorial has been unencumbered by user interface issues and many of the other logistics which often complicate applications.

Index