neoAccess™

Introductory Reference Manual
Version 3.0

Cross-Platform Object Database Component

neo logic

1450 Fourth Street, Suite 12
Berkeley, CA 94710
(510) 524-5897
AppleLink: NeoLogic
CompuServe: 71762,214
AOL: NeoLogic
Internet: neologic@holonet.net

NeoAccess Introductory Reference Manual, Version 3.0.5

By Bob Krause and Alexander Vladimirsky

Special thanks to Brian Blackman, Jean-François Brouillet, Jeff Hokit, Suresh Kumar, Paul Ossenbruggen, Mike Rockwell, Reede Stockton and the hundreds 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, Tim Duane, Robert Inchausti, Theresa McGlashan, Lisa Piercey, Tim Standing, Larry Zulch, and Laura Zulch for their support and encouragement.

copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from NeoLogic Systems.

NEOLOGIC SYSTEMS MAKES NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY, OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Table of Contents

Introduction

Welcome

Thank you for considering to develop your applications using 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 is a full featured object-oriented database engine with incredible performance.

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. Developers of object-oriented systems 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 method

invocation.

Standard 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 provide the facilities for 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.

The first section of this manual presents you with an introduction to the capabilities of NeoAccess. 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 10 minutes. While we assume you have a

working knowledge of these terms, this section explains how NeoLogic has fused these two ideas into a very powerful development tool. NeoAccess is very different from other database engines, object-oriented or otherwise, so this first section is required reading for everybody.

Following the introduction is a section titled Preliminaries. Each topic in this section discusses issues of interest which don't relate specifically to a NeoAccess class. You should refer to these discussions when you need to understand those aspects of the NeoAccess Developer's Toolkit.

The next section is a tutorial which discusses, in great detail, the PowerPlant implementation of the Laughs sample application. First time NeoAccess developers are encouraged to read this section carefully.

The final section includes a reprint of an article written for Frameworks, the journal for object-

oriented developers on the Macintosh. This is a factual, though rather tongue-in-cheek, recounting of the development and implementation of the Photographer's Assistant sample NeoAccess application.

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`".
- Most class names begin with "`CNeo`".

• The names of delegation class begin with "`ENeo`".

• 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.

Overview

The Database

Information on computers today is usually stored in files. The operating system presents a file as a single stream of bytes. This data stream is typically read in and written out serially. However, the file system also provides a mechanism for “seeking” to particular locations in the file.

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; like an OpenDoc or OLE container.

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.

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. 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 where the file resides in the file system.

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.

Closing the Database

A database object needs to be closed before the application terminates. If any objects have been added or changed, then the database needs to be updated before it is closed.

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 the set of operations that are defined for that data.

For example, a CNeoDatabase is an object. This class of object contains state information: it refers to an operating system file, it has a length and an object count. But the real value of a CNeoDatabase is that it does things for you

without your having to know how it does them. For example, your application will ask 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 do it. Object classes centralize and isolate intelligence so that 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 object 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 all 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. What remains is the application-specific content.

Historically most applications that support a document architecture do so by using a fairly rudimentary mechanism called a stream. Streaming data back and forth between the document and memory is sometimes called the inhale/exhale approach to object persistence. All persistent objects 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. Non-object database systems also force developers 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. Updates

necessitate that another set of wrappers be written to copy data back into the database record format for writing back in the database.

Thankfully, applications written using NeoAccess don’t need to worry about these details. The approach taken by NeoAccess is that objects should be viewed as having a set of properties and a pliable 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 by NeoAccess with persistence and sharing properties. 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 API to these properties deal 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. 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.

Creating an Object

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

Sharing an Object

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 other concurrency issues.

The class CNeoPersist provides a sharing property. Any persistent object, whether or not it is permanent, can be shared using this facility. 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.

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 theoretical application used above, a user adds a new person to the address book. When the time comes to add an object to a database, your application calls the database's `addObject` method.

Locating Objects in the Database

Ultimately, the true value of a database is its ability to locate objects quickly and easily when you need them. The interface to the database query mechanism needs to balance simplicity with power. NeoAccess has an extensible interface that provides both.

If the truth be told, object database usage so far has not grown at a rate comparable to that of relational systems a decade earlier. One of the biggest reasons is that database applications are

designed under the assumption that data can be retrieved using relational queries (which are also called associative lookups). While object databases execute referential queries (also called parts explosions or ISAM) 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 by relational query mechanisms.

Objects in a NeoAccess database are organized by class. This is analogous to the way that relational systems store records in tables. So, for example, all CCircle objects in a graphics application would be indexed together, as would all CSquares. But NeoAccess also knows how classes are related to one another. It knows, for example, that CCircle and CSquare have a common parent class, CShape. Knowing the genealogy of classes allows the query

mechanism to be much more powerful than a purely relational system could ever be. By performing a single database query, the screen update method of our mythical graphics application can locate all objects having a base class of CShape that are in a particular update region.

NeoAccess includes several static function members including `Find`, `FindEvery` and `FindByID` - for locating an object or group of persistent objects. These methods can search a specific class of objects, or a base class and all of its subclasses. Your application-specific subclasses of CNeoPersist can include additional methods that provide similar capabilities (`CShape::FindShapeByRegion`, for example).

As another example, suppose the personal productivity application that was mentioned

before includes the ability to list all those people that the user is scheduled to meet with today. The day object or the calendar component defines a method that locates the proper set of person objects. This method, lets call it `FindTodaysPeople`, locates all the appointment objects for today, identifies the people the appointments are with, and then locates and returns those person objects. This may sound complicated, but, in fact, it is quite easy to implement.

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 person 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.

When the permanent state of an object changes in memory, steps need to be taken to insure that this change is reflected in the on-disk state of the object. The method that changes the

value of a permanent data member should call the object's `setDirty` method to indicate that the object's state has changed and needs to be updated on disk. This change will be committed to disk when the database is later committed.

Removing an Object

Inevitably, your application will need to remove objects from a database. The database's `removeObject` method 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.

Deleting an Object

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

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, the purpose of which 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. So locating the object the next time will be fast. But don't worry, the cache is purged when application memory is low. So 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. (Though your application's mileage may vary.)

The NeoAccess Development Experience

Who's Who

A typical development team on a project using NeoAccess has probably assigned specific responsibilities to individuals or groups within the team. It depends on the project, but most teams of two or more people 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 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 enjoyable as possible. But it is particularly important that complexity and logistics be hidden from application developers. This not only allows them to be as

productive as possible, but also allows them to design and develop a front-end that is decoupled from the specifics of how objects are defined, organized and accessed in the back-end.

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 extendibility that is the most important advantage the object systems provide over procedural systems.

Preliminaries

Introduction

This section contains reference material for developing applications using NeoAccess. Rather than force you to wade through the complete interface definition, this section provides a set of topics that are generally of interest to developers. You can obtain more detailed information by referring to specific methods or classes in the object definition sections that follow this one. Many of these examples refer to the various sample applications which are included with NeoAccess.

Environment-Neutral Frameworks

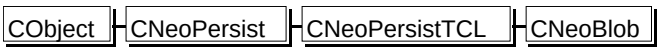
NeoAccess is an object framework that provides object persistence capabilities. This support is provided as a set of C++ classes that extend standard application frameworks on several

different platforms using a number of different development environments and compilers. The product was written to be environment-neutral so as to facilitate portability. NeoAccess portability is implemented through the use of environment-specific classes and by using compile-time symbols and typedefs.

CNeoBlob Inheritance Tree Using OWL

Consider the above diagram which shows a simplified inheritance tree for the class CNeoBlob when built using Borland's ObjectWindows Library (OWL) application framework. The root of all streamable classes in OWL is TStreamableBase, so naturally that is CNeoBlob's root class as well. Because blobs are persistent objects CNeoBlob also inherits from CNeoPersist. The immediate parent of CNeoBlob is CNeoPersistOWL. This class supplements CNeoPersist, which is framework-neutral, with persistence support that is unique

to OWL. For example, all OWL classes should support the `isA` and `isEqual` methods.



CNeoBlob Inheritance Tree Using TCL

Now consider the diagram immediately above which again depicts the inheritance tree of CNeoBlob, but this time under Symantec's THINK Class Library (TCL) application framework. The TCL also has a single root class, but it is called CObject. In this environment CNeoBlob's immediate parent is CNeoPersistTCL. The environment-specific support that this class provides is different than that provided by CNeoPersistOWL.

Isolating environmental dependencies in subclasses greatly improves portability. It also reduces the complexity and increases the clarity of environment-neutral classes.

Adding and Removing References to an Object

The sharing property inherited from CNeoPersist is useful whether an object is permanent or not. However, care must be taken to insure that references are properly added and removed. This will ensure that an object is deleted from memory only after all references have been removed.

References can be added to an object by using the `referTo` method of the object. A reference is implicitly added to an object when it is obtained from a database. When your application has finished referring to it, it should call `unrefer` to remove the reference.

NOTE

There is an easy way to remember which NeoAccess methods add references to objects and which don't. The static `FindByX` methods all add a reference, as do swizzler objects; all other methods should not. Another way to remember is that methods called primarily by application developers (again, the `FindByX` and swizzler methods) add a reference before returning an object, while methods used primarily by database developers (index-related methods and the like) do not add a reference.

Consider the sample routines below:

```
void Masseuse(CNeoDatabase *aDatabase)
{
 long    index;

 for (index = 0; index < 100; index++)
     MassageObject(aDatabase, index);
```

```
}

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

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

 /**
  ** 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 error that caused a `Failure` 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, nil, nil);

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

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

Unless the compiler supports C++ exceptions, 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.

Our third example, `MassageObject3`, removes the reference that is added by the database object, but adds a reference using `object`'s `autoReferTo` method.

```
void Masseuse2(CNeoDatabase *aDatabase)
{
 long    index;

 /**
  ** Remember the current state of the reference table.
  **/
 checkpoint = CNeoPersist::GetCheckpoint();
 NEOTRY {
     for (index = 0; index < 100; index++)
         MassageObject(aDatabase, index);
 }
 NEOCATCH {
     /**
      ** Restore the state of the reference table
      ** by removing all the references that have
      ** been added since the checkpoint occurred.
      **/
     CNeoPersist::ResetCheckpoint(checkpoint);
 }
 NEOENDTRY;
}

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

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

 /**
  ** Replace our object reference with one
  ** that is checkpointed.
  **/
 object->autoReferTo();
 object->unrefer();

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

 /**
  ** Send the object the autoUnrefer to remove the
  ** reference we obtained from autoReferTo.
  **/
 object->autoUnrefer();
}
```

The methods `autoReferTo` and `autoUnrefer` are just like `referTo` and `unrefer`, respectively.

The method `autoReferTo` adds a pointer to the object to a reference table. The reference table is a global array.

Each occupied entry in this array points to an object that has had a reference added to it by `autoReferTo`. If an object has had references added to it by `autoReferTo` more than once, then there will be multiple entries in the reference table that point to the object.

The method `autoUnrefer` differs from `unrefer` in that `autoUnrefer` removes the object from the reference table. The method `autoUnrefer` removes the reference from the LAST object in the table.

References to objects need to be removed from objects in the reverse order in which they were added. Care must also be taken to remove references added by `autoReferTo` with `autoUnrefer`, and references added by `referTo` have to be removed by `unrefer`.

NOTE

Don't forget that the reference table is a global array of finite length. The methods `autoReferTo` and `autoUnrefer` should not generally be used to add references to an array of objects. The number of entries in an array is often indeterminate and may quickly overflow the reference table.

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 the allows developers to signal and to recognize 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 "Adding and Removing References to an Object" immediately above for

more information.

An object can be marked busy and unbusy by using the `setBusy` and `setUnbusy` methods respectively.

```
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();
}
```

The CNeoPersist class definition also includes `autoBusy` and `autoUnbusy` methods that keep the busy state of an object consistent in the event of a failure. Objects for which these methods are called are tracked by a busy table, which is used the same way that the reference table is used for tracking object references.

```
void ChangeObject(CAppSpecific *aObject)
{

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

 /**
  ** 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->autoUnbusy();
}
```

The methods `GetCheckpoint` and `ResetCheckpoint` track and reset the state of the busy table and reference table.

NOTE

It is the application's responsibility to ensure that a `NEOCATCH` block be set up to capture failures and restore objects to a consistent state. The `ResetCheckpoint` method simply resets the busy state and reference count of those objects that have been set busy using `autoBusy` or had references added using `autoReferTo` since the checkpoint was taken.

Cross-Platform Development

NeoAccess supports the development of cross-platform applications. Cross-platform support includes the use of environment-neutral classes and their derivatives. Whereas other cross-

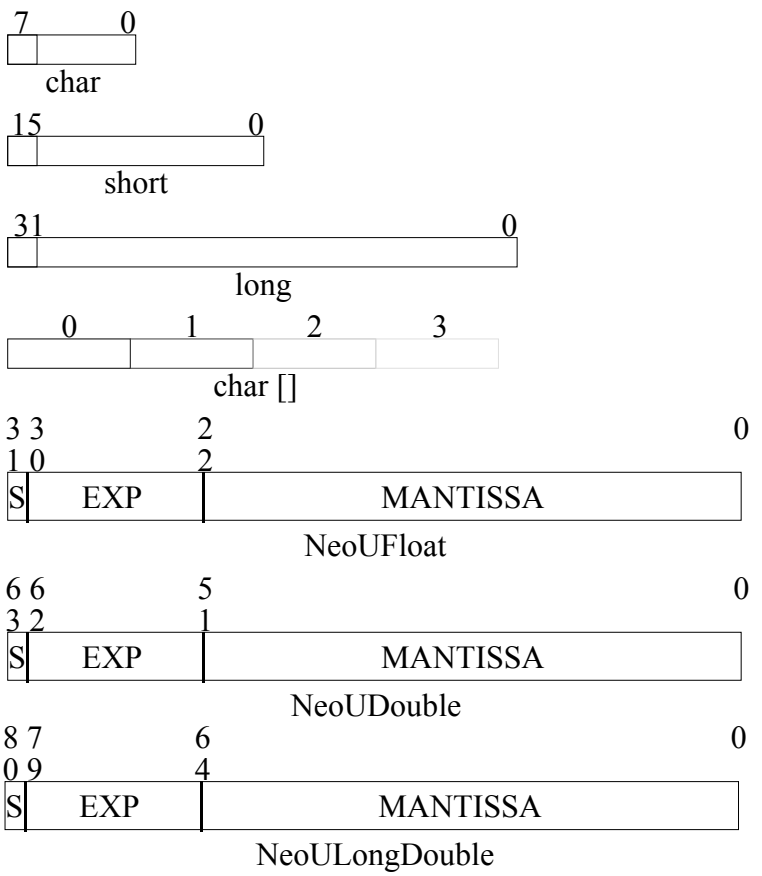
platform development environments rely primarily on preprocessor macros and conditional compilation to isolate compiler, development environment and machine dependencies, in NeoAccess these dependencies have been isolated into environment-specific classes. NeoAccess source code was organized to make portability as easy as possible. Most environment-specific aspects of NeoAccess are isolated in environment-specific header files so that these symbols can be redefined easily. Moving NeoAccess to still other platforms, Unix and OS/2 for example, has been greatly simplified as a result.

While NeoAccess has been reworked to improve portability, the overall complexity of the API has also improved. You can continue to write your application code in a machine-dependent fashion, or take advantage of the platform-independent features of NeoAccess to help you develop applications in multiple environments.

Binary Compatibility of NeoAccess Files

NeoAccess supports binary compatibility of databases on any platform. What this means is that users can create databases on any machine and then copy that file to another kind of machine and have the application on the second machine open and manipulate it without converting the database's internal format. This incredibly powerful feature helps developers that use NeoAccess build multi-platform applications that produce documents that can be used on any machine.

Binary compatibility is achieved by standardizing the binary format of data in NeoAccess databases. This standardization approach, which is often called **byte swapping**, has been used in network communication protocols for years. It allows networked machines to communicate in heterogeneous environments.



NeoAccess Standardized Data Formats

NeoAccess defines several macros which are used to convert data to and from machine-usable and NeoAccess standardized formats. If the machine-usable format in a particular environment is the same as the standardized format, then these macros are trivial and the symbol `qNeoByteSwap` is zero. One-byte data types do not have to be converted because the machine-usable format of signed and unsigned characters is the same as the

standardized format in all environments that NeoAccess currently supports. The macros for converting short, long and floating point formats are as follows:

```
NeoSwapShort(x)          // swap 2-byte integer in place
NeoSwapShortInto(s, d)   // swap 2-byte integer s into d
NeoSwapLong(x)           // swap 4-byte integer in place
NeoSwapLongInto(s, d)    // swap 4-byte integer s into d
NeoUFloat2Native(s, d)   // copy NeoUFloat s to NeoFloat d
NeoNative2UFloat(s, d)   // copy NeoFloat s to NeoUFloat d
NeoUDouble2Native(s, d)  // copy NeoUDouble s to NeoDouble d
NeoNative2UDouble(s, d)  // copy NeoDouble s to NeoUDouble d
NeoULDouble2Native(s, d) // copy NeoULongDouble s to NeoLongDouble d
NeoNative2ULDouble(s, d) // copy NeoLongDouble s to NeoULongDouble d
```

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.

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 dimensions 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 created OpenDoc or OLE containers.

The interface to CNeoPersist, the base persistence class of NeoAccess, includes a pair of object serialization methods, `readObject` and `writeObject`, which are used to serialize the persistence state of objects to and from NeoAccess streams. Subclasses of CNeoPersist override these methods so that their persistent data members are also preserved and restored appropriately. For the most part, `readObject` and `writeObject` methods can be written without regard for the type of stream being used. The advantage of this approach is that a single set of methods 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 methods. In order to avoid asking developers to override multiple sets of serialization methods, 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` methods.

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.

Framework-Specific Subclasses of CNeoStream

Some application frameworks include their own type of stream classes. Developers wishing to read and write objects to these streams usually need to override a set of methods in much the same way that users of NeoAccess streams override `readObject` and `writeObject`.

NeoAccess environment-specific support for these frameworks include an environment-specific subclass of CNeoStream which allows developers to use both NeoAccess streams classes as well as the framework-specific stream classes without having to implement multiple sets of serialization methods. Developers that use framework-specific stream classes through this interface can preserve and restore objects using NeoAccess standard `readObject` and `writeObject` methods.

Iterating over a List of Objects

Iterators can be used to iterate over a collection of objects. Some application frameworks in which NeoAccess is supported include their own iterator classes. The collection classes of still other frameworks include iteration methods that support similar capabilities. The type of collection that NeoAccess uses are called extended binary trees (also called btrees). So

NeoAccess’s iterator classes extend the set of iterator classes available to the developer to include keyed iterators that can be used to iterate over btrees. (See the CNeoIterator and CNeoIndexIterator sections for more information on NeoAccess’s iterator classes.)

Occasionally, an application needs to apply a function to all objects of a class 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. Another way would be to pass a function pointer to one of the `FindByX` methods. Yet another alternative would be to use the database's `doUntilObject` method. This final option is the one that 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` method, 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(CNeoNode *aNode, const short aIndex, 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` method. 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 parameters passed to this function and the value it returns will be discussed in more detail in the example that follows this.

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 CNeoPersist {
public:
                        …
 Boolean                isPriority(void);
                        …
protected:
                        /** Instance Variables **/
 long                   fPriority;
                        …
};

void *DisplayPriorityMsg(CNeoNode *aNode, const short aIndex,
                         const NeoLockType aLock, void *aParam)
{
 Boolean     done    = FALSE;
 CMessage *  msg;

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

 /**
  ** 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` method. 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 that the first two arguments are a pointer to a CNeoNode object and a `short` integer. The class CNeoNode is the abstract base class of all btree nodes. 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` method to obtain a pointer to it.

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

Notice that the object has an additional reference added to it while the object is referenced by `DisplayMsg`. This is because the `getObject` method of CNeoNode does not add a reference before returning the object. It is the caller's responsibility to do so. `DisplayPriorityMsg` uses the methods `autoReferTo` and `autoUnrefer` to add and remove the reference. This example assumes that the caller of `DisplayPriorityMsgs` has set up `NEOTRY` blocks and checkpoints to ensure the integrity of object reference counts.

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

Changing an Object

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 the next time changes are committed. The methods of objects that modify these data members should use the `setDirty` method to mark the object as having been modified.

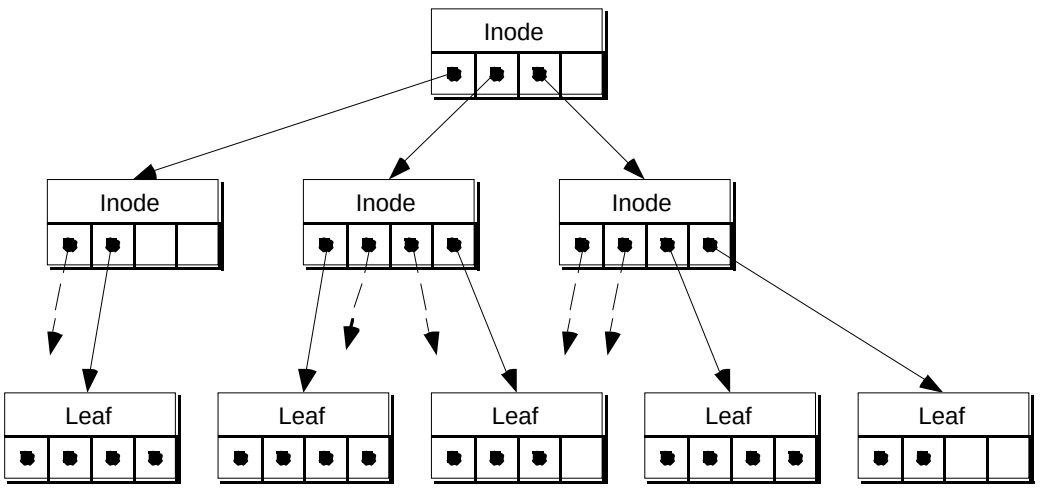
```
void CAppSpecific::setPermValue(const long aValue)
{
 fPerm = aValue;
```

```
 setDirty();
}
```

Btree Classes

One of the most powerful subcomponents of NeoAccess is the **extended binary trees** (or simply **btree**) that it uses to organize data. While the general structure of a typical NeoAccess database are discussed in greater detail elsewhere, developers should know that most the constructs such as classes and indices are in fact btrees. 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. The abstract base class on which all NeoAccess node classes are based is CNeoNode.

Btrees are a type of collection class. Arrays and linked lists are other collection classes that you might have worked with before.



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

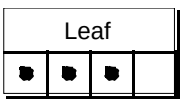
nodes are connected. Leaf nodes provide the tree's content.

There are two general types of btree classes. One whose entries refer directly to some other target object is called a **direct btree**. We'll call btrees whose entries do not refer to target objects **indirect btrees**. The target objects, which tree 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. 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 can 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 once. 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. The class CNeoNode 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 methods for manipulating the header and entries of a node. Subclasses override these methods to implement and manage the specific capabilities of a node derivative.

Different kinds of btrees may contain many different types of nodes. But most searching and tree traversal methods are unaware of other node types 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 method has a pointer to an object of an indeterminate class and that method would like to instantiate another node of that same class. It would use the `getAnother` method 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

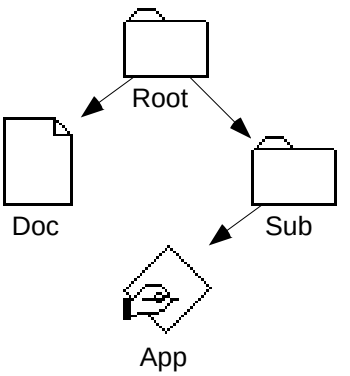
Indices organize objects in some sorted order. While other diagrams in NeoAccess documentation show the structure of a database depicted indices as a single block, they are in fact (not surprisingly) btrees. The reason they are btrees is that random searches need to be as fast as possible, and no construct can provide

faster random access to large collections of persistent objects than can btrees.

The order that objects are sorted in is determined by its index classes. All index classes are derived from CNeoNode. The NeoAccess Developer's Toolkit includes several index classes. But in order to sort objects in some application-specific fashion, by shoe size for example, you need to define your own index class.

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 how this feature might be exploited, consider one way in which the Macintosh Finder, the Windows File Manager or any other hierarchical file system might be implemented using NeoAccess.



Sample File System Containment Hierarchy

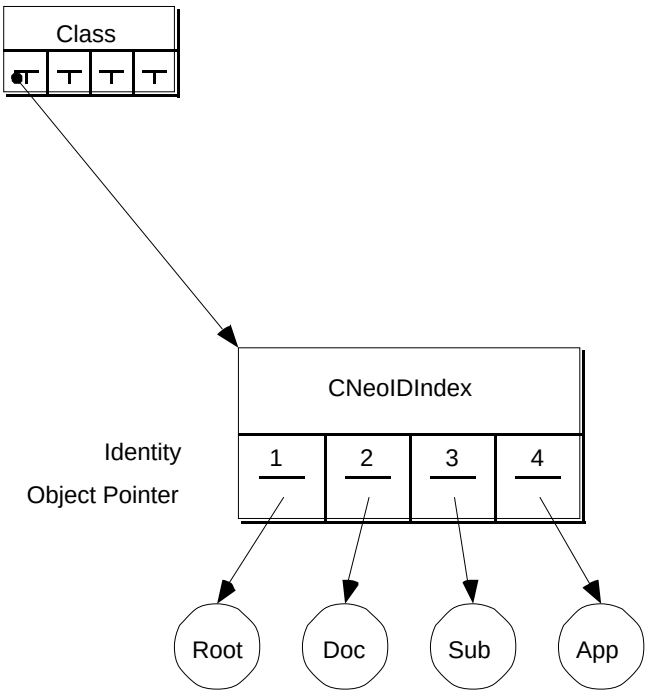
Suppose the file system contained a set of four files having a configuration as depicted in the figure above. The file system assigns a 4-byte value to uniquely identity each file. Each file also tracks the identity of its parent. That is to say, the class CFile has two indices.

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 by ID. Sorting objects by ID is in fact

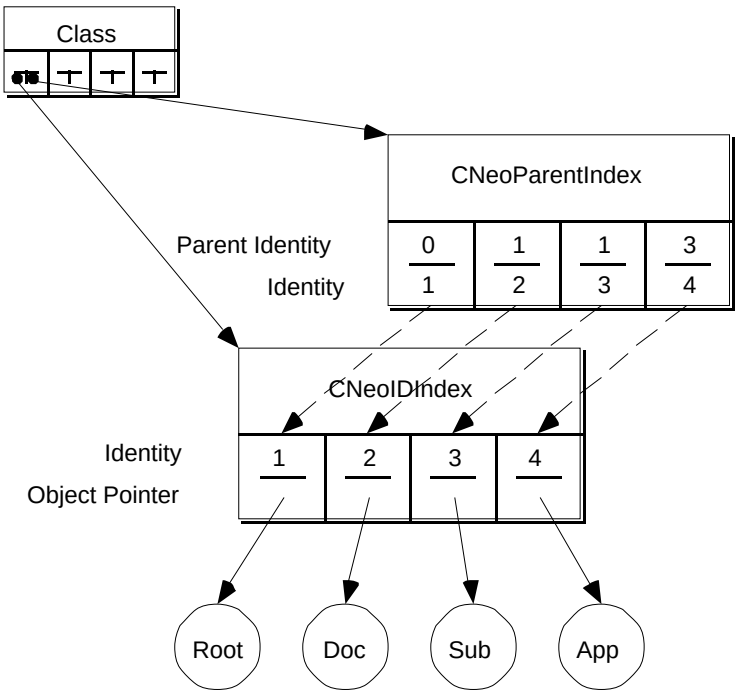
the default sorting order of objects in a NeoAccess database. The second index sorts file objects by parent ID. Any index beyond the primary index is called, not surprisingly, a **secondary index**. Secondary indices are always indirect btrees. (Direct and indirect btrees were discussed in greater detail in the immediately previous discussion titled “Btree Classes”.)

Classes are themselves btrees. (You’d be amazed to find out how 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 to the file objects directly.



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.

Developers interested in knowing more about NeoAccess index classes and how they are defined and used should refer to the discussions of the CNeoIDIndex, CNeoParentIndex, and CNeoParentNNameIndex classes in the class reference section of this document.

Consolidated Indices

It is sometime useful to sort all objects having a given base class in a single index. Consider the file system example once again. User usually want to view a folder's contents in alphabetical order without regard for the type of the file. Yet 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.

A **consolidated index** is one which refers to all objects having a common base class. This is in contrast to the usual practice of organizing all objects of each specific leaf class in a separate index. A consolidated index is created by configuring all metaclass objects of classes having the common base class to have a index root value equal to the class ID of the base class. See the discussion of the `setKey` method of the CNeoMetaClass class for more information on creating a consolidated index.

Creating Application-Specific Index Classes

Applications that wish to sort objects in an application-specific order can do so by creating their own index classes. This process is fairly straightforward. 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. A good example of a secondary index is CNeoParentIndex.

In additional to the methods all CNeoNode subclasses must override, all index classes need to override the following methods as well; `getObject`, `insertObject`, `removeObject` and `KeyManager`. Primary indices also need to override the `getChildIndex`, `remove` and `setEntryParent` methods.

KEY POINT

NeoAccess is among the fastest commercial database engines available. It uses a highly optimized binary search algorithm to perform random access searches. The only method of an application-specific index class that are called during these searches (other than the I/O methods) is `getObject`. This method may be called literally hundreds or even thousands of times a second during a search. For this reason special care should be take to ensure that it is implemented as efficiently as possible.

forgetChildren(const short aIndex)

This method is called when the node entry of the specified child is being deleted. It's purpose is to break the bond that exists between parent and child objects in memory. This routine uses the child object's `setParent` method to break the child's reference to the index, then it breaks its reference to the child.

getChildIndex

The `getChildIndex` method simply iterates over the node's entries until the given child object is found.

getObject

The `getObject` method returns the object referred to by the specified entry. The entries of a primary index, by definition, refer directly to target objects. While the entries of secondaries refer to objects indirectly. As such, the implementation of the `getObject` method varies based on whether the index is primary or secondary.

The entry of a primary index refers directly to the object if it is already in memory, in which case `getObject` simply returns a pointer to it (without adding a reference!). If the object is not in memory, then a new object must be allocated using the metaclass's `getOne` function. The newly allocated object's `readObject` method is called to read the object's state from a location supplied by the index entry.

While secondary indices don't refer to the target object directly, they do hold enough information to locate the object using the primary index. The `getObject` method for these indices creates a select key of a type supported by the primary index and calls the database's `findObject` method.

KEY POINT

The database's `findObject` method adds a reference to every object that matches the given selection criterion. However, the object returned by an index's `getObject` method should not have a reference added. The `getObject` method for secondary indices should therefore remove the reference added by `findObject` before passing the object to back to the caller. (A reference to an object is removed by using the object's `unrefer` method.)

insertObject

The process of adding an object to a database involves inserting an entry into each of the indices

for that given class of objects. NeoAccess determines the proper place in the index to add the entry by performing a binary search. Once that location has been determined the `insertObject` method is called to add the object in the proper place in the index node.

The implementation of `insertObject` simply fills in the fields of an index entry which is allocated on the stack and then calls the index's `insertEntry` method to actually insert the entry into the node.

If all of the entries in the node are already in use then NeoAccess may need to add extra inodes or leaf nodes to the index tree. This may result in a different node being at the root

of the tree. The return value of `insertEntry` refers to this new root if there is one. The index's `insertObject` method should return this value to its caller.

KeyManager

The metaclass of each index class must be set to refer to a **key manager** function. The general structure of this function is often called a **dispatch** routine. Dispatch routines usually serve multiple purposes. The first argument of a key manager function indicates which operation to perform. A variable number of additional arguments may also be passed depending on the operation requested. At a minimum all key manager functions must support the `kNeoCanSupport` and `kNeoGetKey` operations.

The `kNeoCanSupport` operation may be called with one or two additional argument. If the second argument is non-`nil`, then it is a pointer to a select key object. If this is the case, then the key manager should return a Boolean value indicating whether or not this index class can support a binary search using this type of select key. If the second argument is `nil`, then the third argument should be a `NeoSelectType` enumeration and the key manager function should return a Boolean indicating whether the index can support a binary search of that type.

The `kNeoGetKey` operation can be called with one additional argument. The second argument is a pointer to a persistent object. The key manager function should return a select key that can be used to uniquely locate the given object in the index.

remove

The `remove` method removes the index node and all objects that it refers to from the database and frees the space in the database that the object occupied. This is actually quite a handy method because everything that the node refers to also is removed. For example, if `remove` is called on a node that is the root of an index, then the entire index tree and all the objects that it refers to is also removed from the database.

KEY POINT

This method should not be called directly. It should not be used to remove an application-specific object from a database. Use `CNeoDatabase::removeObject` instead.

setEntryParent

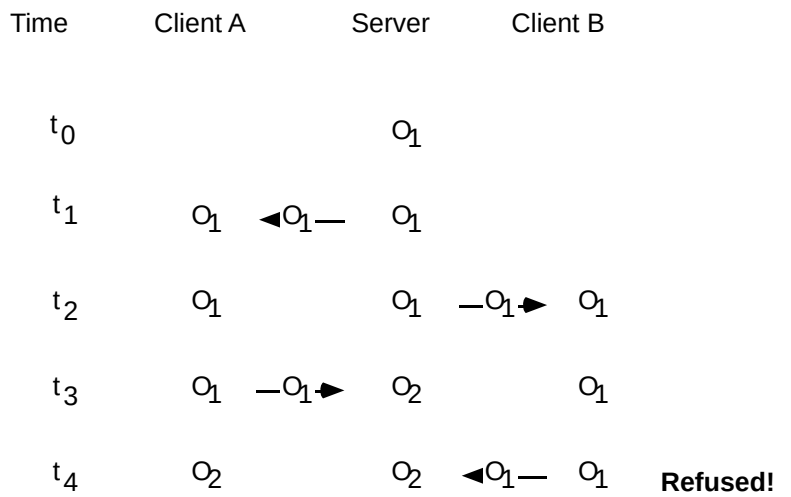
Primary index entries refer to objects, which refer back to the index. This bond between the index and the target object needs to be established when an object is brought into memory. Additionally, the bond may need to change should the parentage of an object change due to additions or deletions in the index tree. The `setParentEntry` method of primary indices simply sets the `fParent` data member of the target object to refer to the index node.

Object Versioning

NeoAccess includes constructs to facilitate the sharing of objects within an application. The object reference count and busy bit are two mechanisms that help 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` method 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 both 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*** is ***O1***. 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 ***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 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 that are 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 you should be prepared for, 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 that 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. However, your applications should 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 duplicate this code, NeoAccess also 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 or 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 it and

recreate it later.

NeoAccess makes it easy for an application to organize and access application-specific temporary data. So easy in fact, that you might want to consider using it to organize and cache objects that don't persist after the application quits. The advantage 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 in every way 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` method. 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 which have their `fTemporary` bit set will be deleted when the application calls the database's `removeTempObjects` method with a non-zero parameter value.

Object Caching

NeoAccess supports a very 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. (Though your application's mileage may vary.)

The object cache uses memory not otherwise being used by the application. This cache can become quite large and consume a significant portion of your application's memory. By default, the cache will use everything it can get its hands on. In environments where virtual memory is not available or where memory allocation for an application is otherwise

bounded, NeoAccess provides a mechanism for freeing objects in the cache when memory is needed. The cache can be purged by calling the `purge` method of a CNeoDatabase object.

```
void GetMemory(CNeoDatabase *aDatabase, const Size aNeeded)
{
 aDatabase->purge(&aNeeded);
}
```

The single argument to `purge` is a pointer to a long that indicates how much memory is needed. The database will attempt to free up at least enough memory to create a block of this size. Depending on how NeoAccess has been configured, `purge` may free more memory than is currently needed. This is done in order to reduce the number of times that low memory situations occur while trying not to reduce the usefulness of the object cache.

Some execution environments, such as pointer-based environments lacking robust virtual memory systems, require that the size of the object cache be bounded. The size of the cache can be bounded. NeoAccess will limit the size of the object cache to something close to the amount specified by the static variable `CNeoPersist::FCacheSize`. If an allocation would cause the cache to exceed this limit, then the `Purge` method of all open NeoAccess databases will be called to reduce the cache so that the allocation can occur.

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 when the operation is scheduled and it finally completes. The parameters passed to the scheduling function includes a pointer to an **i/o completion routine**, a function which will be 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 was 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 and 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 method 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` methods 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 when you license NeoAccess you are given complete source code. As the designer of NeoAccess, I believe that I have provided you with the best object-oriented database solution available. However if you don't agree, then you can change whatever you don't like. How's that for power!

kNeoBusyTableSize

The busy table is an array of pointers to objects that have been set busy by calls to the `autoBusy` method of CNeoPersist. The number of entries in this table is defined by the value of the `kNeoBusyTableSize` constant. You should make sure that your application sets this value high enough so that the table does not overflow.

kNeoMaxIndice

This constant determines the maximum number of indices that any class can have. The default value of this compile time symbol is 4. If you think that your application may include a class having more than this number of indices, then you should includes this value before shipping your application.

KEY POINT

Changing the value of `kNeoMaxIndice` may change the format of your database on disk. For this reason, the value of `kNeoMaxIndice` should be set to a safe maximum value before your application goes into production. Having said that, developers should keep in mind that increasing the value of `kNeoMaxIndice` by one will increase the memory and file space requirements of the application on average by about 200 bytes.

kNeoMaxClasses

This constant determines the size of the metaclass table. All application-specific class ids should be between 20 and kNeoMaxClasses. If your application requires more than 28

classes, then the value of kNeoMaxClasses should be changed accordingly. The default value of kNeoMaxClasses is 48.

kNeoClassEntries

This constant determines the maximum number of entries that a CNeoClass object can contain. Your application's database may contain more or fewer classes than this value. However, if there are more than `kNeoClassEntries`, then more than one CNeoClass object will be needed and accessing classes will be minimally slower.

kCNeoDatabaseCurrVersion

You should initially define the constant to be 1. Each time the format of your database changes, you will need to change this value and write a routine that converts objects with changed layouts to the new format.

kCNeoDatabaseQuantum

As objects are deleted, the file space they occupied is tracked so that it can be reused. Adjacent blocks of freed file space are combined into a single larger block. Each block is an entry in the database's free list. The value of the `kCNeoDatabaseQuantum` constant indicates the minimum size of a free block. Setting it too low may result in many small blocks of free space that are too small to be reused. Setting the value too high may result in inefficient use of file space.

kNeoPurgeQuantum

Permanent objects that have been read in from or added to a database but disposed of by the application are cached by the database object. This cache enhances the performance of search operations significantly. However, the cache also uses up precious memory. When memory becomes low, the `new_handler` or

`GrowZone` functions of the runtime environment can ask the database object to delete objects in the cache. Rather than free just enough memory to satisfy the current memory shortage, the database will free a chunk of memory based on the value of the constant `kNeoPurgeQuantum`. Setting this value too low may cause time-consuming purge operations to occur more frequently. Setting it too high may reduce the effectiveness of the object cache.

kNeoReferTableSize

The reference table is an array of pointers to objects that have been referred to by calls to the `autoReferTo` method of CNeoPersist. The number of entries in this table is defined by the value of the `kNeoReferTableSize` constant. You should make sure that your application sets this value high enough so that the table does not overflow.

kNeoSubclassEntries

This constant determines the maximum number of entries that a CNeoSubclass object can contain. The subclass list is traversed when locating objects of a base class and its subclasses. Setting this value high enough will minimize the tree depth of the subclass list. However, setting it too high may result in wasted file and memory space. In general, you should leave it as it is.

qNeo2P0FileFormat

With the introduction of streams in version 2.2 of NeoAccess, the average size of a NeoAccess database is reduced. This is possible because of optimizations made in the database format of some internal NeoAccess classes. However this feature can be disabled, thereby preserving the old database format used in earlier releases by defining the compile-time symbol `qNeo2P0FileFormat`.

qNeoLaundry

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 to simply mark objects dirty and then at commit time traverse the class list and index trees, writing any dirty objects found 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.

We recommend that NeoAccess-based applications be built with `qNeoLaundry` not defined, though there may be some situations where it may be more advantageous to use it.

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.

Debugging Tips

Great care has been taken in the design of NeoAccess to make the programming interface and documentation as clear as possible. However our experience working with developers are they built their first NeoAccess-based products has shown us which areas developers experience the most problems in getting their applications to function properly. The purpose of this section is to convey some information that other developers have experienced in the past.

Rule #1

Make sure that the compile time symbol `qNeoDebug` is defined throughout the development process.

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.

Define `verify` Methods for Your Persistent Classes

CNeoPersist has a virtual function, called `verify`, which is conditionally compiled in when `qNeoDebug` is defined. The purpose of this method is to verify the consistency of the object. It does this by making assertions, such as:

```
NeoAssert(fRefCnt > 0); // Otherwise it wouldn't be in memory
```

All subclasses of CNeoPersist should override `verify`. These overrides should make as many universal assertions as possible about the state of the object and other objects which it might refer to. In most cases, the implementation should end by returning the value of `NeoInherited::verify(aValue)`.

The meaning of `verify`'s only argument varies depending on the type of object it is being called for. For node classes, the argument is either `nil` or a pointer to the last entry in the immediately preceding leaf node in the btree. For fruit objects, the argument is either zero or the length of the database file in bytes. The `verify` method of these class should also return the same type of value that they take

KEY POINT

The `verify` method is a `const` function. It should NEVER allocate memory nor bring in objects from disk. Its purpose is to passively, though rigorously, verify the state of an object in memory.

Add Debugging Code

While NeoAccess has been instrumented to verify during debugging the state of objects in memory, there is no reason why developers can't augmented this process by adding additional debugging code where trouble is suspected.

The most prevalent form of debugging code found in NeoAccess is `NeoAssert`. This is a macro which generates an error if the given assertion fails. Like this:

```
NeoAssert(microsoft == kNeoFriendly);   // Usually fails.
```

`NeoAssert` is disabled when `qNeoDebug` is undefined.

Sometimes debugging code can't be shoehorned into an assertion. These checks should be conditionally compiled using the following construct.

```
#ifdef qNeoDebug
 if (newRoot)
     NeoNodeVerify(newRoot);
#endif
```

Garbage Data

Probably the most difficult type of problem to debug is when the developer believes that an object has been committed to disk but contains garbage after it is read back in. Most commonly this is caused by either the object NOT actually being written to disk or by it being written or read incorrectly.

One reason why an object may not be written to disk is that it's not dirty. Remember, if you change the state of an object's permanent data member, then you must call the object's `setDirty` method. Don't just set the object's `fDirty` data member either. That's not enough. You must call `setDirty`.

Another common cause of garbage data is errors in the `readObject` or `writeObject` methods of your application-specific classes. These methods should both begin by immediately calling `NeoInherited`. This should be followed by a call to an i/o method of the stream for each of the persistent data members of your class. Don't try to read or write the data members of the parent classes.

Also, verify that you are using the proper i/o method for the data member's type. `NeoID`s and `NeoMark`s are longs, for example.

The order in which data members are read in are the same in which they are written out.

```
void CWidget::readObject(CNeoStream *aStream, const NeoTag aTag)
{
 NeoInherited::readObject(aStream, aTag);

 fThingamabob = aStream->readLong();
 fWidgetCount = aStream->readShort();
}

void CWidget::writeObject(CNeoStream *aStream, const NeoTag aTag)
{
 NeoInherited::writeObject(aStream, aTag);

 aStream->writeLong(fThingamabob);
 aStream->writeShort(fWidgetCount);
}
```

If you find that an object contains garbage data after it is read back in, then you should set breakpoints at the top of the leaf class's `writeObject` and `readObject` methods. When the debugger stops in the `writeObject` method you should inspect and remember the values of the data members of the garbage data members. Also remember the value of `fMark`, the location of the object on disk. Single step in the debugger to see that each data member is written out as the proper type and in the proper order. Then continue execution until the debugger stops in the `readObject` method for that same object as it is read back in. Verify that the `fMark` value is the same as when it was written out. Finally, step through the `readObject` method of the class containing the garbage values to verify that data members are being read back in as the same type and order they were written out.

At a very low level, NeoAccess is a memory manager – the type of space that it manages is file space. Just as enough memory must be allocated to contain an object in memory, enough file space must be allocated for the object on disk. Make sure that the `getFileLength` method is overridden by every concrete persistent class. This method should return the maximum amount of file space which objects of that class will occupy on disk. If a data member is variable length, like a string, then `getFileLength` should assume the maximum length that it can be. If `getFileLength` returns a value which is too low to contain the persistent data members of an object then writing this object to disk may cause other objects to be overwritten and corrupted.

KEY POINT

The `getFileLength` method is a `const` function. The prototype for this function should be EXACTLY as follows:

```
virtual long        getFileLength(void) const;
```

KEY POINT

C++ DOES NOT SUPPORT THE CONCEPT OF A VARIABLE LENGTH CLASS. NEITHER DOES NEOACCESS. If one of your classes contains a variable length data member, like a string, but you would prefer not to have to reserve file space for the maximum length of that data member, then embed in your persistent class an ENeoBlob data member to store this variable length data discontiguously from the object proper. See the discussion of the ENeoBlob class in the reference section of this manual for more information.

Dangling or Insufficient References

The reference counting mechanism in NeoAccess minimizes application complexity. It allows objects to be easily shared by different components within an application. But in order to make this mechanism work for them, developers should pay special attention to insure that object references are properly added and removed.

A particular area of confusion with some developers is understanding when NeoAccess automatically adds references to objects that it returns. There is an easy way to remember which NeoAccess methods add references to objects and which don't.

KEY POINT

The static `FindByX` methods all add a reference, as do the swizzler objects; all other methods do NOT. Another way to remember is that methods called primarily by application developers (again, the `FindByX` methods) add a reference before returning an object, while methods used primarily by database developers (index-related methods and the like) do not add a reference. The methods of an iterator, such as `nextObject` and `currentObject`, do not add a reference to the objects that they return.

If you find your code referencing an object with an invalid reference count or one that looks like it has been freed, then there is a chance that you didn't add an object reference when you should have and as a consequence, the object was purged.

If you find your application is constantly running out of memory, then perhaps you've failed to remove references to objects. Another possibility is that objects are being marked busy but never reset once they are in a consistent state again.

Block Move Errors

The most common type of error after dangling pointers in any software is caused by incorrectly moving variable length data blocks from one location in memory to another. If you are experiencing memory corruption problems you should double check your pointer arithmetic for both the source and destination. Also, verify that you are in fact moving the correct amount of data. And if you are working with handles, then make sure you're doing the double indirection properly.

Function Overrides

C++ virtual functions allow a derived class to override behavior provided by its base classes. This is an incredibly powerful construct. It is the way in which polymorphic behavior is implemented in the C++ language.

When overriding a virtual function, developers need to make sure that the prototype of the override is exactly the same as the function it is overriding. In particular, the `const`'ness of a function and its arguments are used by the compilers to determine whether a function is an override or an entirely different function. For example, the following two methods are different functions as far as C++ is concerned:

```
virtual NeoID       getClassID(void) const;
virtual NeoID       getClassID(void);
```

Another issue to consider is whether the implementation of the override should call the inherited method which it overrode at some point, like this:

```
void CWidget::readObject(CNeoStream *aStream, const NeoTag aTag)
{
 NeoInherited::readObject(aStream, aTag);

 fWidgetLong = aStream->readLong();
 fWidgetCount = aStream->readShort();
}
```

Metaclasses and Class IDs

Some developer experience problems the first time they add an object of a particular class to a database. This can sometimes be caused by the fact that a metaclass object for that class was not created when the application started up. The constructor of your application class should create a metaclass object for each of your application-specific persistent classes.

Every application-specific persistent class, even application-specific index classes, should be assigned a unique class ID value between 20 and `kNeoMaxClasses`. You can change the value of `kNeoMaxClasses` if need be. The default value is 48.

Finally, verify that each of your concrete persistent classes override the `getClassID` method and that it returns the proper (and unique) class ID value.

The value of `gNeoDatabase`

NeoAccess supports the ability to have more than one database open at a time. Most database operations are calls to CNeoDatabase methods. These methods automatically set the `gNeoDatabase` global variable to refer to this database for the duration of the

method. However some methods which access the database, such as CNeoBlob's `getBlob` method, assume that `gNeoDatabase` set already set to refer to the database in which the blob is contained. Your application may operate incorrectly if gNeoDatabase is not properly set. The safest way to insure that `gNeoDatabase` is always set properly is to set it in the activation method of the document or window objects.

Heap Fragmentation in Pointer-Based Environments

Macintosh memory management is entering a potentially painful intermediate phase in its transition from a handle-based environment to one which is pointer-based.

All major application frameworks on the platforms that NeoAccess supports are now pointer-based. While framework developers are quickly taking advantage of this by using multiple inheritance and stack-resident and embedded objects, application developers need to pay much closer attention to the issue of heap fragmentation than was the case in handle-based environments.

The Macintosh Motorola 680X0 runtime architecture is in a particularly difficult phase in this transition. The difficulty lies in the fact that 680X0 runtime architecture still allocates fixed amounts of memory within which applications are expected to operate. Heap fragmentation in pointer-based applications can render 20-50% percent of this memory inaccessible.

NeoAccess includes a mechanism which limits the amount of application memory used by the NeoAccess object cache. The maximum size (roughly) of the cache can be configured at run time by changing the value of the static variable `CNeoPersist::FCacheSize`. The approximate current size of the cache is defined by the static variable `CNeoPersist::FCacheUsed`.

Macintosh developers might want to use temporary memory to allocate objects with a short life span. Temporary memory is memory "borrowed" from the system heap. There are some rather restrictive conventions associated with the use of temp memory. Consult the Macintosh technical documentation for complete details.

Believe it or not, as the overall mix of applications running at once becomes largely pointer-based, asking users to turn on virtual memory becomes a viable option. Some of the sluggishness that has come to be associated with Macintosh virtual memory in the past was due to "thrashing" caused when a heap consisting mostly of handles was compacted.

In general, application developers should be aware that during this transitionary period memory management is an issue that will require careful attention.

You Have Source Code!

Most database engines are black-box procedural libraries. When something goes wrong with these systems, the only thing a developer has to go on is what it says in the documentation. NeoAccess is different. It has an open architecture which was designed to be extended by developers. While NeoAccess's programming interface was designed to minimize visible complexity, if there is something about the system that you don't understand, though you'd like to, feel free to explore the internals of the engine itself.

Changes in NeoAccess 3.0

NeoLogic recognizes that developing sophisticated software can take time. Rome was not built in a day. This being the case, it is our policy to minimize changes to the NeoAccess programming interface between minor releases. This results in a smooth transition path that allows developers to begin a project using the latest NeoAccess release but easily upgrade to future releases as they become available. At the same time, a delicate balance must be struck to insure that NeoAccess can continue to evolve in light a technological advances in the field. Major NeoAccess updates may include changes from the previous interface which are necessary to support advanced features, frameworks and architectures.

The following is a summary of the changes to the programming interface introduced in NeoAccess 3.0. Developers moving from

previous NeoAccess releases should carefully review this list.

Class and Variable Name Changes

Existing customers have commented about the names of several NeoAccess classes and variable names. The name of the database class in previous releases was called CNeoFile instead of CNeoDatabase. The global variable which refers to the current database object used to be called `gNeoFile` instead of `gNeoDatabase`. And the names of some methods and data members also referred to the database as a file. For example, NeoAccess document classes included a `newFile` method which is now called `newDatabase` and the document's `fFile` data member which exists in some environments is now called `fDatabase`.

The class CNeoPartMgr used to be called CNeoPart.

The base class of all index classes used to be CNeoIndex. This abstract base class has been eliminated. All index classes now derive directly from CNeoNode.

The mechanism in which consolidated indices are supported in NeoAccess 3.0 differs from previous releases. This eliminated the need for the CNeoGenericIndex class.

Constants and Compile Time Symbols

The names of some constants and compile time symbols have been changed from earlier releases usually because they did not conform to NeoAccess naming conventions. Changes include the following:

New Name	Old Name
`kNeoNotFound`	NeoNotFound
`kNeoMemFull`	NeoMemFull
`kNeoNoError`	NeoNoErr, noError

The ability to bound the maximum size of the object cache used to be enabled only if the

compile time symbol `qNeoBoundedCache` was defined. This capability is always enabled and the need for the compile time symbol has been eliminated.

The calling sequence to functions called by NeoAccess for each object that matches a given selection criteria used to vary based on whether the compile time symbol

`qNeoFastDoUntilObject` was defined. The protocol to thes functions is now constant and the compile time symbol is not longer needed.

Calling Sequences to NeoAccess Methods

The `compare` method of CNeoPersist classes have been eliminated. As have the `compareEntry` methods of CNeoNode classes. The comparision functions used during database search operations are now handled by the `compare` functions of select objects. See the discussion of the CNeoSelect class and its derivatives for more information.

The calling sequence to persistent objects' `verify` methods has changed. See the "Debugging Tips" discussion earlier in this section for more details.

The `GetCurrentDatabase` and `SetCurrentDatabase` methods of

`CNeoDatabase` used to be called `GetCurrentFile` and `SetCurrentFile`, respectively.

The `addDefaultClasses` method of the database class has been eliminated.

The `getAnother` and `removeObject` methods of CNeoNode are no longer virtual.

The `getSelectType` method of CNeoSelect is no longer virtual.

The `addToList` `deleteFromList` methods of CNeoPartMgr take an object pointer instead of a part list entry pointer.

Most CNeoNode subclasses no longer need to override the `forgetChild` method.

The `getNextSibling` and `getPreviousSibling` methods of CNeoPersist have been eliminated. Most uses of these methods should be converted to use interator objects instead.

The prototype of persistent objects' `commit` method have changed. Their first argument used to be a database object instead of a CNeoContainerStream pointer.

The implementation of CNeoNode derivatives' `KeyManager` method has changed significantly. See the `KeyManager` discussion in the CNeoMetaClass reference section for more information.

Tutorial

Introduction

The structure and flow of control of applications built using NeoAccess is different in some ways from other applications. One major difference is that most other applications read in the entire contents of a file 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 see how NeoAccess is integrated into and used in an application. While the sample used in this discussion may not be available in your development environment, you should still read through this material as 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, part list and iterators. We'll examine in detail the PowerPlant version of Laughs. While some aspects of this discussion are PowerPlant-specific, the general principles and structure can be applied to development in other environments in which NeoAccess operates.

Laughs is a simple database application. The first time it runs it creates a database to which it adds some objects. Subsequent runs of Laughs 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. With the exception of some initialization that PowerPlant

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 = new CLaughsApp();
 app->Run();
 delete app;

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

The application class for Laughs is itself fairly plain. It is a subclass of the native NeoAccess application class. It includes a constructor and an override of the `Run` method 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.

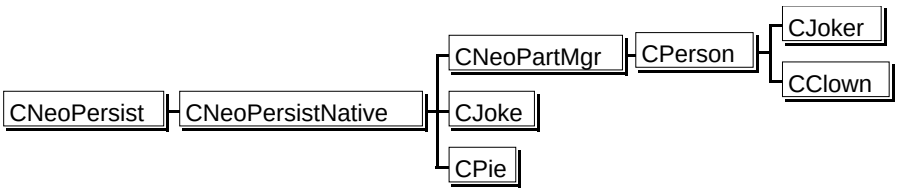
```
const OSType kLaughsSig      = 'Neo6';

class CLaughsApp : public CNeoAppNative {
public:
                     /** Instance Methods **/
                     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 five persistent classes: CNeoPersist, an optional class generically referred to as CNeoPersistNative, CPerson, CJoker and CClown. 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 Methods **/
 virtual            ~CNeoPersist(void) {}

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

 virtual long       getFileLength(void) const;

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

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

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

```
/** Concurrency Methods **/
```

```
 void                referTo(void);
 NeoRefCnt           unrefer(void);

 void                setBusy(void);
 void                setUnbusy(void);
};
```

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 methods, `getClassID` and `New`, provide a two-way mapping between a class ID and an object of that class. The instance method `getClassID` will return the class ID of a particular object. The class method `New` is called 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 method. These overrides will simply add the amount of file space needed to preserve the persistent attribute values of that class to the requirements of the parent class.

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

The most significant value 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 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` method. If the ID of an object isn't set by the application, NeoAccess will assign the object a unique ID when the object is added to the database.

Applications are often dynamic systems. The state of persistent objects change in response to user actions. NeoAccess provides a simple mechanism to manage change. Methods that modify persistent attribute values use the `setDirty` method to indicate that the object has changed and that that changed object needs to be committed to disk. NeoAccess keeps track of all dirty objects and commits changes all at once. This means that 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 methods 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 methods `FindEvery` and `FindByID`, for example, add a reference to every object they find. Application code uses the `referTo` and `unrefer` methods 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` method. NeoAccess will take care of deleting the object from memory once the last reference to it has been removed.

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` methods. 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.

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 used in. However, providing a seamless integration of CNeoPersist into some application frameworks might require adding additional overrides beyond the general support provided in CNeoPersist.

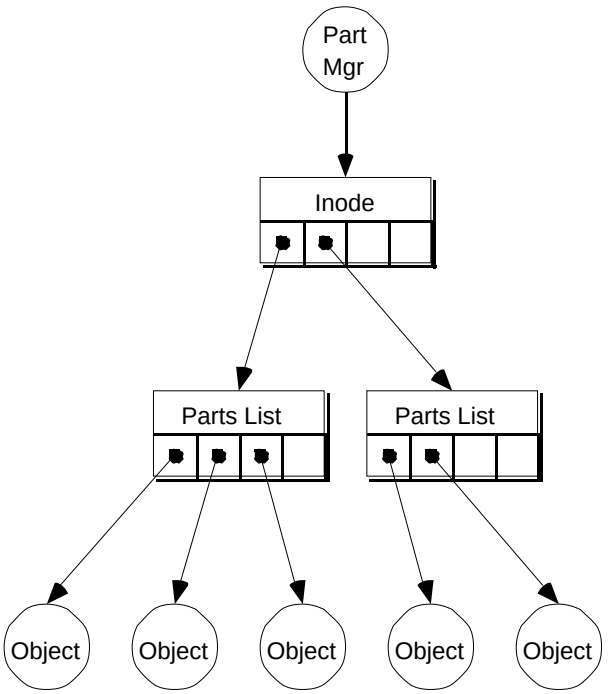
Borland's ObjectWindows Library (OWL) application framework, for example, asks that all concrete subclasses of TObject override the `isA` method. One way to provide this support would be to compare an object's class ID with the given argument value. A basic implementation could do this simply by calling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

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 to some applications. 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 Mgr 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 and are able to pronounce their name on demand. Names can be a maximum of 49 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. The definition of CPerson is as follows:

```
class CPerson : public CNeoPartMgr
{
public:
                CPerson(const CNeoString &aName = "\p");
 void           getName(CNeoString &aName) const {aName = fName;}
 void           printName(void) const;
 void           setName(const CNeoString &aName = "\p") {fName = aName;}
 virtual void   skill(void) const = 0;

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

protected:
 CNeoString     fName;
};

const NeoID     kPersonID       = 25;

const long kPersonFileLength = kNeoPersistFileLength + 50;
```

Note that `kPersonID` is the unique class ID for this class. Class ID values 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 value of `kPersonFileLength` is the amount of file space needed to preserve attribute values of this class and all of its parent classes. Subclasses of CPerson can use this value as a reference in determining the amount of space they need.

The implementation of the CPerson constructor is simplicity itself. It simply takes the given name of this object.

```
CPerson::CPerson(char *aName)
{
 fName = aName;
}
```

The method `printName` prints a line on `stdout` which states the person's name.

```
void CPerson::printName(void) const
{
 printf("Name is %s\n", fName);
}
```

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 methods of CPerson, `readObject` and `writeObject`, are used primarily by NeoAccess to serialize data members. Each class which contains data members the values of which must be preserved, must override these two methods to read in and write out these values. The `readObject` method 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` method 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 view of application developers.

```
void CPerson::readObject(CNeoStream *aStream, const NeoTag aTag)
{
 NeoInherited::readObject(aStream, aTag);

 aStream->readString(fName, sizeof(fName));
}

void CPerson::writeObject(CNeoStream *aStream, const NeoTag aTag)
{
 NeoInherited::writeObject(aStream, aTag);

 aStream->writeString(fName, sizeof(fName));
}
```

The implementations for `readObject` and `writeObject` methods of CPerson begin by giving the parent class an opportunity to read/write its data members. Once this is done it calls the stream’s `readString` and `writeString` methods to restore/preserve the name of this person.

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 value of `aTag` is sometimes useful in these situations. We’ll ignore it during this tutorial.

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 know are not very funny!) The definition of CJoker is as follows:

```
const NeoID kJokerID         = 26;

class CJoker : public CPerson
{
public:
                     CJoker(char *aName = "");

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

 void                skill(void) const;
 void                forgetJoke(const CJoke *aJoke);
 CJoke *             getJoke(const long aIndex) const;
 long                getJokeCount(void) const {return getListCount();}
 void                learnJoke(CJoke *aJoke);
};
```

The constructor passes its arguments on to its parent, CPerson. We saw earlier that CPerson is a type of part class. 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(char *aName)
 : CPerson(aName)
{
 setObjClassID(kJokeID);
}
```

Concrete classes should override the `New`, `getClassID` and `getFileLength` methods to create the proper type of objects and return the persistence particulars for the class, respectively. The implementations of these methods are as expected.

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

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

long CJoker::getFileLength(void) const
{
 return kPersonFileLength;
}
```

Teaching a joker a new joke is accomplished by using the method `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 can also be used to sort objects in some other order. Adding a joke to a joker's part list involves simply calling the joker's `addToList` method.

```
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` method simply calls the joker's `deleteFromList` method. Note though, that removing a joke from a joker's part list does not remove it from the database entirely. Jokes can be share (read: stolen) between jokers. Just because one joker decides to forget a joke does not mean that all other jokers must as well.

```
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` method for jokers tells a joke. The joke it tells is chosen at random from the vast repertoire, which is kept track of by the part list of the joker object. The method begins by determining how many jokes are in the joker's part list. If you look at the implementation of `getJokeCount` , it simply calls CNeoPart's `getListCount` method, 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.

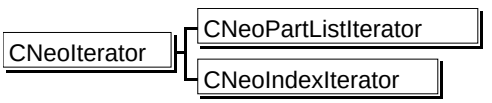
```
void CJoker::skill(void) const
{
 long    count   = getJokeCount();   // How many jokes do I know?
 CJoke * joke;

 if (count) {
     // Pick a joke at random.
     joke = getJoke((rand()&0x7FFFFFFF) % count);
     NeoAssert(joke);

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

     // Don't forget to remove our reference to the joke.
     joke->unrefer();
     joke = nil;
 }
 else
     printf("Tells Jokes, but has no jokes to tell.\n");
}
```

Now let's take a look at what's involved in retrieving a joke from a joker's part list. The joker's `getJoke` method 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. The CNeoIndexIterator class is used to traverse a NeoAccess index. (The database's `getIterator` method can be used to create and initialize an index iterator.) But the implementation of `getJoke` uses a CNeoPartListIterator to iterate over the joke part list. We use the base part class's `getIterator` method 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 method 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` method 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 aIndex) const
{
 CJoke *               joke;
 CNeoPartListIterator *iterator  = (CNeoIDListIterator *)getIterator();

 // Randomly pick a joke
 iterator->leap(aIndex);
 joke = (CJoke *)iterator->currentObject();

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

 // Don't need this any more.
 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 definition is as follows:

```
const NeoID  kJokeID         = 28;
const long   kMaxJokeLength  = 80;
```

```
class CJoke : public CNeoPersistNative
{
public:
                    CJoke(const char *aText = "");

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

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

 void                getJoke(char *aText) const
                        {strncpy(aText, fJoke, kMaxJokeLength -1);}
 void                printJoke(void) const;
 void                setJoke(const char *aText)
                        {strncpy(fJoke, aText, kMaxJokeLength -1);}

protected:
 char                fJoke[kMaxJokeLength];
};
```

CJoke is a concrete persistent class. As such, it is a derivative of CNeoPersistNative, and overrides the `New`, `getClassID` and `getFileLength` methods to create the proper type of objects and return the persistence particulars for the class, respectively. The implementations of these methods are once again no surprise.

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

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

long CJoke::getFileLength(void) const
{
 return kPersistFileLength;
}
```

As we saw in the implementation of CPerson, the `readObject` and `writeObject` methods 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)
{
 NeoInherited::readObject(aStream, aTag);

 aStream->readString(fJoke, sizeof(fJoke));
}
```

```
void CJoke::writeObject(CNeoStream *aStream, const NeoTag aTag)
{
 NeoInherited::writeObject(aStream, aTag);

 aStream->writeString(fJoke, sizeof(fJoke));
}
```

For completeness, we see that the `getJoke`, `printJoke` and `setJoke` methods are everything you would expect them to be.

```
void CJoke::getJoke(char *aText) const
{
 strncpy(aText, fJoke, kMaxJokeLength -1);
}

void CJoke::printJoke(void) const
{
 char    joke[64];

 strcpy(joke, fJoke);
 printf("%s\n", joke);
}

void CJoke::setJoke(const char *aText)
{
 strncpy(fJoke, aText, kMaxJokeLength -1);
}
```

CClown

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

```
const NeoID kClownID     = 27;

class CClown : public CPerson
{
public:
                CClown(char *aName = "", NeoID aID = 0);

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

 void           skill(void) const;
 void           setPieType(char *aPieType) {strcpy(aPieType, fPietype);}

protected:
 char           fPietype[80];
};

const long kClownFileLength  = kPersonFileLength + 80;
```

The constructor passes its arguments on to its parent, CPerson. It also designates Custard to be the default type of pie that clowns throw.

```
CClown::CClown(char *aName, NeoID aID)
 : CPerson(aName, aID)
{
 setPieType("Custard");
}
```

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

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

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

long CClown::getFileLength(void) const
{
 return kClownFileLength;
}
```

The `skill` method 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. Use the `setPieType` to change the type of pie a clown throws.

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:
                    CPie(const char *aFilling = "Custard");

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

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

 void                getFilling(char *aText) const
                        {strncpy(aText, fFilling, kMaxFillingName -1);}
 void                setFilling(const char *aText)
                        {strncpy(fFilling, aText, kMaxFillingName -1);}

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

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

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

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

long CPie::getFileLength(void) const
{
 return kPersistFileLength;
}
```

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

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

 aStream->readString(fPieType, sizeof(fPieType));
}
```

```
void CPie::writeObject(CNeoStream *aStream, const NeoTag aTag)
{
 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 CLaughsApp.

The Constructor

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

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

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

 // Add the application-specific metaclasses to the metaclass table.
 meta = new CNeoMetaClass(kNameIndexID, kNeoNullClassID, "\pCNameIndex",
                          CNameIndex::New, CNameIndex::KeyManager);

 meta = new CNeoMetaClass(kNeoIDListID, kNeoNullClassID, "\pCNeoIDList",
                          CNeoIDList::New);

 meta = new CNeoMetaClass(kPersonID, kNeoPersistID, "\pCPerson",
                          CPerson::New);
 meta->setKey(kNeoSecondaryIndex, kNameIndexID);

 meta = new CNeoMetaClass(kJokerID, kPersonID, "\pCJoker", CJoker::New);
 meta->setKey(kNeoSecondaryIndex, kNameIndexID, kPersonID);

 meta = new CNeoMetaClass(kJokeID, kNeoPersistID, "\pCJoke",
                    CJoke::New);

 meta = new CNeoMetaClass(kClownID, kPersonID, "\pCClown", CClown::New);
 meta->setKey(kNeoSecondaryIndex, kNameIndexID, kPersonID);

 meta = new CNeoMetaClass(kPieID, kNeoPersistID, "\pCPie", CPie::New);

 // Inject a little more randomness into what is about to occur.
 srand((int)time(nil));
}
```

The constructor of the Laughs application begins by announcing that the fun is about to begin. This is followed by the allocation of three metaclass objects.

A metaclass is a class that describes other classes. 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` method for

the class. 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 secondary index key is designated for the joker and clown classes. This is done by calling their metaclass's `setkey` method. The first argument to `setKey` indicates that this is a secondary index. The second argument is the class ID of the index class, which is kNameIndexID in both cases.

If a third argument is given to `setKey`, it must be the class ID of some base class. For example, the third argument of the `setKey` call on the CJoker and CClown classes is `kPersonID`. This is an example of a consolidated index. A **consolidated index** is one which refers to all objects having a common base class. All concrete person objects, be they jokers or clowns, are all sorted 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 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.

The last line of the constructor simply increases the randomness of the results we get when retrieving jokes and pies from the database.

Creating a Document

Laughs' `createDocument` method is called to create a new document. 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` method very much like this one.

The method begins by setting up a `NEOTRY` block to capture and recover from any failures which might occur as the document is allocated or initialized. The `NEOCATCH` handler for this `NEOTRY` 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 constructor indicate whether the document needs to support printing, whether the database being opened is a new database and whether it is a remote database. Laughs is simple enough that it doesn't support printing. For now we can assume that it is a new database being opened (as opposed to a pre-existing one). The last argument is reserved for future use and should be FALSE.

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 that the user worries about losing it to a system crash uses the Save As… menu item. 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.

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

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

 NEOTRY {
      /*
       * Create your document.
       *
       * The arguments indicate whether the document is...
       *  printable (FALSE),
       *  a new database (TRUE),
       *  remote (FALSE).
       */
      document = new CLaughsDoc(FALSE, TRUE, FALSE);

      /*
       * Call the document's newDatabase method.
       */
      document->newDatabase();
 }
 NEOCATCH {
      /*
       * 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) {
          delete document;
          document = nil;
      }
 }
 NEOENDTRY;

 return document;
}
```

The Laughs Document Class

It is the charter of the document class of an application to consolidate the various information sources that the document 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 applications also manage Publish & Subscribe editions as well. Microsoft's OLE 2 (Object Linking and Embedding) and Apple's OpenDoc will push the evolution of document classes further still.

Some frameworks, PowerPlant included, 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);

 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 less friendly environments.

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

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

Creating a Document

Earlier in this tutorial we saw that the document's `newDatabase` method is called by the application's `createDocument` method immediately after creating the document. This method begins by setting up a `NEOTRY` block to capture any failures that might occur. (This is included in the sample simply to illustrate the importance of setting up `NEOTRY` blocks.)

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 method, `Specify`, which takes a name and a Macintosh directory ID. The name of the Laughs database file is always "Laughter". A volume reference number and directory ID of zero refers to the directory containing the Laughs application.

If a file exists in the specified location, as determined by the `existsOnDisk` method, then the database's permissions are set to read only, otherwise they are set to read/write and a file of the proper type is created on disk.

The parent class's `newDatabase` method is given an opportunity to do its thing before the file is actually opened. Having opened the file with the proper permissions, the `createObjects` method is called to add objects to the database. Otherwise, `printOut` is called to print its contents.

```
void CLaughsDoc::newDatabase(void)
{
 NeoPerms    permissions;
 FSSpec      fileSpec;

 // Most applications don't hardcode the location of its file on
 // disk. Laughs is so simple that it does.
 fileSpec.vRefNum = 0;
 fileSpec.parID = 0;
 NeoStringCopy("\pLaughter", fileSpec.name);
 fFile->Specify(&fileSpec);

 fNewDatabase = !fFile->existsOnDisk();
 if (fNewDatabase) {
     fFile->create();
     permissions = NeoReadWritePerm;
 }
 else
     permissions = NeoReadPerm;

 NeoInherited::newDatabase();

 fFile->open(permissions);                    // Open the database.

 if (fNewDatabase)
     createObjects();
 else
     printOut();

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

 printf("\n\nDone!\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 set this value when a document is activated. Developers should make sure that it is set properly in their applications.

Adding Objects to the Database

OK, let's get some work done. The first time Laughs is run it ends up creating a new database named "Laughter" in the same folder as the application. It then calls `createObjects` to add two CJoker objects and a CClown object to the database. Each joke it taught a joke or two and each clown is given an arsenal of pies to dole out.

```
/***
*    createObjects
*
*    Add three objects to the database, two jokers and a clown.
*
***/
void CLaughsDoc::createObjects(void)
{
 CJoke *         joke1;
 CJoke *         joke2;
 CPie *          pie;
 CJoker *        joker;
 CClown *        clown;
 CNeoDatabase *  database        = getDatabase();
 CNeoString      string;
 char            name[64];

 // Tell them what we're about to do.
 database->getName(string);
 NeoBlockMove(&string[1], name, string[0]);
 name[string[0]] = 0;
 printf("Storing 2 Jokers & a Clown in \"%s\".\n", name);

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

 // Add it to the database.
 // Note: An object ID is assigned automatically by addObject.
 database->addObject(joke1);

 // Is this a joke???
 joke2 = new CJoke("My dogs got no nose?");

 // Add it to the database.
 database->addObject(joke2);

 // Create a joker object.
 joker = new CJoker("\pJack");

 // 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;

// Create a clown.
```

```
clown = new CClown("\pFred");

// 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();
pie = nil;

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

// Create another joker.
joker = new CJoker("\pHarry");

// 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;
}
```

NeoAccess persistent objects are created just as all C++ objects are, by using the `new` operator. Each person object is initialized with a name. Having done that, the object is added to the database using the database's `addObject` method. Note that adding joker and clown 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?

Once an object is added to the database, the pointer to it is no longer needed. In order to keep the reference count of the object consistent, `createObjects` uses the `unrefer` method. And then, just for completeness, it sets the pointer to `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` method marks each object dirty so that its state is saved to disk when the changes are committed. These changes are committed by the caller of `createObjects` by using the

database's `commit` method.

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 Laughs finds a pre-existing database it opens the database and calls the application's `printOut` method. This method searches the database locating each object by using an index iterator to traverse the list of people in the database by name.

```
/***
*    printOut
*
*    Find in turn each of the three objects in the database, two jokers
*    and a clown.
*
***/
void CLaughsDoc::printOut(void)
{
 CPerson *           person;
 CNeoNameSelect      key("\p");
 CNeoDatabase *      database        = getDatabase();
 CNeoIndexIterator   iterator(database, kPersonID, &key, TRUE, TRUE);
 CNeoString          string;
 char                name[64];

 // Tell them what we're about to do.
 database->getName(string);
 NeoBlockMove(&string[1], name, string[0]);
 name[string[0]] = 0;
 printf("Restoring %ld Jokers & %ld Clowns from \"%s\".\n",
     database->getObjectCount(kJokerID, FALSE),
     database->getObjectCount(kClownID, FALSE), name);

 person = (CPerson *)iterator.currentObject();
 while(person) {
     person->printName();
     person->skill();
     printf("\n");
     person = (CPerson *)iterator.nextObject();
 }
}
```

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 study has been unencumbered by user interface issues and many of the other logistics which often complicate applications. But in some ways Laughs is too simple. It doesn't use the normal constructs for opening a pre-existing database. It doesn't use iterators, multiple or application-specific indices, blobs, part lists or any of the many other features which makes NeoAccess so useful to developers.

Photographer's Assistant

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 study has been unencumbered by user interface issues and many of the other logistics which often complicate applications. But in some ways Laughs is too simple. It doesn't use the normal constructs for opening a pre-existing database, for example.

Interested readers should also read the last section of this document, titled Photographer's Assistant. It discusses the NeoDemo sample application which is included in the TCL-specific portion of the NeoAccess Developer's Toolkit.

CNeoApp

Heritage

The Heritage and Ancestry of CNeoApp

NeoAccess is a persistence framework that has been designed to extend standard application frameworks. As such, there are several NeoAccess classes which must interface with the application framework. CNeoApp is one of these classes. It is an environment-neutral class. The immediate base class of CNeoApp varies depending on the application framework within which NeoAccess is being built. The environment-specific header file for the framework will define the typedef `CNeoAppbase` to be a synonym for this base class.

Some application frameworks include an application class from which the project-specific class inherits. Some methods of this

base class may need to be overridden in order to provide more natural and complete support for CNeoApp subclasses. These overrides should be implemented as an environment-specific subclass of CNeoApp. The environment-specific header file for the framework will define the typedef `CNeoAppNative` to be a synonym for this subclass.

Introduction

NeoAccess occasionally needs to interface with environment-specific areas of the application framework. The architecture of most standard application frameworks usually delegates primary responsibility for process state manipulation, the scheduling and dispatching of idle tasks and the handling of low memory situations to the application object. The abstract base class CNeoApp is designed to be a subclass of the application framework's application class. CNeoApp includes a number

of pure virtual functions that are further overridden by environment-specific subclasses of CNeoApp. These virtual functions provide an environment-neutral interface with which to access these functional areas.

Using CNeoApp

There are situations that arise in NeoAccess where some tasks need to be deferred until some later point in time. These deferred tasks are called chores. CNeoApp provides support for tracking chores and seeing that they are done during idle time. Your application

can make use of this facility by subclassing the base class CNeoChore. Chores are scheduled, dispatched and dequeued by calling `addChore`, `doChores` and `removeChore`, respectively. (Environment-specific subclasses of CNeoApp call `doChores` at idle time.) The methods `getAppName` and `getAppVersion` return the application's name and version strings, respectively. The application's `purgeCache` method should be called in low memory situations to free memory used by the NeoAccess object cache.

CNeoBlob

Heritage

The Heritage and Ancestry of CNeoBlob

Introduction

Developers strive to make everything in today's applications an object. But the fact remains that not everything can be. In recognition of this fact, NeoAccess provides an class called ENeoBlob. Blobs are persistent objects that are used to store and retrieve non-object entities. ENeoBlob can be embedded as a data member in subclasses of CNeoPersist to provide blob management capabilities.

CNeoBlob is a "convenience class" which delegates responsibility for managing blobs to a data member of type ENeoBlob. This class is included in the standard NeoAccess release for backward compatibility reasons as well as

because is an excellent example of how a CNeoPersist subclass might add blob management capabilities by using ENeoBlob.

Using CNeoBlob

CNeoBlob can be used to “objectify” free-form variable-length data. Applications can use the capabilities of CNeoBlob by defining subclasses of CNeoBlob.

A significant feature of the CNeoBlob class is that the blob object and blob data are separate on disk and in memory. So the blob object can be brought into memory without having to have the potentially much larger blob data portion in memory. The object and data portions can be allocated in non-contiguous locations in the database as well.

There may be situations when the free-form variable-length component of a blob is dirty or in an inconsistent state though the “object” part is not. This class includes mechanisms for

separately marking the object's and data's busy and dirty states, as well as the ability to load and purge the data portion separate from the object.

ENeoBlob

Heritage

The Heritage and Ancestry of ENeoBlob

Introduction

Developers strive to make everything in today's applications an object. But the fact remains that not everything can be. In recognition of this fact, NeoAccess provides an class called ENeoBlob. Blobs are persistent objects that are used to store and retrieve non-object entities. ENeoBlob can be embedded as a data member in subclasses of CNeoPersist to provide blob management capabilities.

Interested developers might want to refer to the CNeoBlob documentation and source for insights as to how ENeoBlob might be embedded into a persistent object.

Using ENeoBlob

ENeoBlob can be used to “objectify” free-form variable-length data. Applications can use the capabilities of ENeoBlob by adding ENeoBlob data members to persistent classes.

A significant feature of the ENeoBlob class is that the blob object and blob data are separate on disk and in memory. So the blob object can be brought into memory without having to have the potentially much larger blob data portion in memory. The object and data portions can also be allocated in non-contiguous locations in the database as well.

There may be situations when the free-form variable-length component of a blob is dirty or in an inconsistent state though the “object” part is not. This class includes mechanisms for separately marking the object’s and data’s busy and dirty states, as well as the ability to load and purge the data portion separate from the

object.

Subclassing ENeoBlob

It may be useful subclass ENeoBlob to create domain-specific blob classes. By far the mostly sited use of blobs is for managing variable length strings. A EString class could be used to manage all persistent strings in an application. Another useful ENeoBlob class would be an image class. Some developers have even created their own persistent variable length array class by subclasses ENeoBlob.

CNeoDatabase

Heritage

The Heritage and Ancestry of CNeoDatabase

CNeoDatabase is an environment-neutral class. The immediate base class of CNeoDatabase varies depending on the application framework within which NeoAccess is being built. The environment-specific header file for the framework will define the typedef `CNeoDatabaseBase` to be a synonym for this base class.

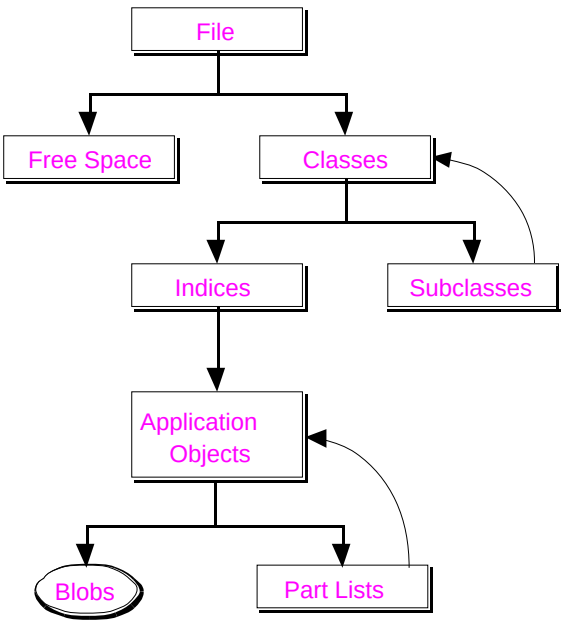
Some application frameworks include abstract base classes from which most application-specific classes inherit properties. Some methods of this base class may need to be overridden in order to provide more natural and complete support for CNeoDatabase subclasses. These overrides should be implemented in an environment-specific subclass of

CNeoDatabase. The environment-specific header file for the framework will define the typedef `CNeoDatabaseNative` to be a synonym for this subclass.

Introduction

The Structure of a NeoAccess Database

CNeoDatabase usually stores objects in a file's data fork or other type of storage container (like an OpenDoc or OLE container). From the application developer's perspective, a typical NeoAccess database contains only application-specific objects. In reality, a database contains many other objects as well. These other objects are used internally by NeoAccess to support the rich set of features that the system provides to developers. The database developer should be aware of the containment hierarchy used by NeoAccess to organize objects in a database.



The Containment Hierarchy of a NeoAccess Database

The Database Header

NeoAccess reserves a certain amount of space at the beginning of a database for a header. The header is read into memory by NeoAccess when the database is opened and a default header is initialized when a new database is created.

The Free List

Deep in the bowels of the system, NeoAccess provides a mechanism for managing the

efficient use of file space. This mechanism works in much the same way that an operating system’s low-level memory manager manages main memory.

The simplest way to allocate file space is to extend the database's length. But as objects are added and later removed from the database, opportunities to shrink the database, move objects or reallocate freed space begin to present themselves. NeoAccess takes advantage of this potential by maintaining a list of free space in a database. Not surprisingly, this is called the free list.

As an application or database developer, there will never be a need to know any details as to how NeoAccess manages file space and the free list. Though NeoAccess automatically takes care of allocating and freeing space for objects as they are added and removed from a database, the database object includes methods that can be used to allocate and free file space directly,

should the need arise.

The Class List

Objects in a NeoAccess database are typically organized primarily by leaf class (the most specific class). NeoAccess uses a class list to keep track of the various classes of objects that have been added to the database. An entry is added to this list the first time an object of a particular class is added to the database. For more information on adding a class, see the topic titled “Making an Object Permanent” later in the CNeoDatabase discussion. The information contained in a class helps NeoAccess determine:

- the class’s 4-byte identity,

- how many objects of that class are contained in the database,
- its ancestry and heritage (its super and sub classes),
- whether or not all objects of a class are temporary, and
- the order in which objects of the class are sorted and accessed.

As is the case with the free list, there is rarely any reason for developers to be aware of how NeoAccess manages classes in a database. The database object contains methods that developers can use to access and manipulate their state.

Subclasses

NeoAccess knows the inheritance hierarchy of persistent classes. For example, a drawing program based on NeoAccess knows that CSquare is a subclass of CRectangle and that CRectangle and CCircle are subclasses of

CShape. NeoAccess uses this information to allow the developer to locate objects based on any base class. So the drawing program can locate all CShape objects that are located in a document's update region with a single call to NeoAccess without regard for whether the shape is a circle or a square.

Each class in a database can have a list of subclasses. A class's subclass list refers to other classes that are based on the parent class. Database developers needn't be concerned with the details of how subclasses are maintained and manipulated.

Indices

While application-specific objects are organized primarily by class, they may be sorted within a class in any number of different ways. Database developers define how a class of objects is to be indexed. Every persistent class has a primary index which defines the primary sorting order

of objects of that leaf class in the database. The default primary index organizes objects by identity. Each class can also have zero or more secondary indices. You may want to refer to the “Index Classes” discussion in the Preliminaries section for more information.

For example, a genealogy application might organize individuals primarily by name, and secondarily by the father’s name and by the mother’s name. This application would be able to quickly locate all individuals based on name or parentage. If individuals were only indexed by name, then answering the question, “Who are the children of Nancy Cotter?” would involve reading in every person object in the database. Indices are what make database systems faster and more flexible than other storage mechanisms.

But application developers can use the same API to locate objects without regard for whether the search can be performed quickly using an

index or whether it involves a linear search of the objects in the class. The “only” tradeoff is performance.

Application Objects

The purpose of using NeoAccess in object-oriented applications is to facilitate the storage and retrieval of application-specific objects, thereby improving the overall architecture and reducing the complexity of applications.

All application-specific persistent classes have a base class of CNeoPersist. Classes that refer to free-form variable-length data may have a data member of type ENeoBlob or may themselves be a subclass of CNeoBlob. Objects that maintain references to collections of other persistent objects should be based on CNeoPartMgr. Btree classes, which include both primary and secondary indices as well as part lists, should be based on CNeoNode. Part lists should either have an architecture similar

to CNeoIDList. See the sections of this

manual that discuss these particular base classes for detailed information on how to create these application-specific subclasses.

Part Lists

A CAD application may allow the user to deal with a collection of parts as a single component. In an even larger context, this component may itself be a member of a sub-assembly, and so on.

A drawing application may include the ability to group shapes into a single construct that the user can manipulate as a single object.

While most database applications require the ability to locate objects by means of relational queries using abstract selection criteria, applications such as those just mentioned may also require support for performing referential queries. The use of this other type of construct is often called a **parts explosion**. The collection of parts referred to by a part manager is, not

surprisingly, called a **parts list**. Using NeoAccess, application-specific grouping objects should have a ENeoPartMgr data member and parts list collection classes should have a base class of, or have an architecture similar to, CNeoIDList.

Subclassing CNeoDatabase

The developers of NeoAccess see two general reasons to subclass CNeoDatabase:

- to provide environment-specific features, and
- to add additional features beyond the scope of the base implementation.

Environment-specific subclasses are defined in order to more naturally support the features of a given development environment. These subclasses should override the `close`, `create`, `flush`, `getName`, `getPathName`, `isOpen`, `open`, `setFileMark` and

`setLength` methods.

A good example of a subclass that provides a parallel feature set with the same programming interface might be a multi-user database implementation. A subclass of CNeoDatabase would manage inter-application concurrency issues in this scenario. It is hard to predict which methods these types of subclasses may need to override, but strong candidates include `addClass`, `addObject`, `close`, `create`, `findObject`, `getClassCount`, `getUnqiueID`, `markClassTemporary`, `open`, `purge`, `removeObject` and `commit`. Complete consideration of the implementations of these methods are beyond the scope of this discussion because of the wide variety of subclasses possible.

Using CNeoDatabase

A CNeoDatabase object is created just like any other C++ object, by using the `new` operator. As

CNeoDatabase is an environment-neutral class, creating a database object usually involves instantiating an environment-specific subclass of CNeoDatabase. To access objects contained in a database, the database must be open. However, before it can be opened a path name must be specified. This path is where the database resides in the file system. Most of the specification methods of CNeoDatabase are environment-specific. Refer to the methods topic of this discussion and of the environment-specific subclasses of

CNeoDatabase for more information on how to specify a database in a particular development environment.

Deep in the bowels of the system, NeoAccess provides a mechanism for managing the efficient use of file space. This mechanism works in much the same way that an operating system's low-level memory manager manages main memory. Though NeoAccess automatically takes care of allocating and freeing file space for objects as they are added and removed from a database, the database object includes methods, `getSpace` and `freeSpace`, that can be used to allocate and free file space directly should the need arise.

Application objects in a NeoAccess database are typically organized primarily by leaf class (the most specific class). The first time an object of a particular class is added to the database, the object's class and indices are

implicitly added as well. The database object includes a method, `addClass`, that developers can use to explicitly add classes before any objects of that class are added.

While application-specific objects are organized primarily by class in a NeoAccess database, they may be sorted within a class in any number of different ways. For example, a genealogy application might organize individuals primarily by name, and secondarily by the father's name and by the mother's name. This application would be able to quickly locate all individuals based on name or parentage. If individuals were only indexed by name, then answering the question "Who are the children of Nancy Cotter?" would involve reading in every person object in the database. Indices are what make database system faster and more flexible than other storage mechanisms.

Database developers define how a class of objects is to be indexed. But application

developers can locate objects in a database based on selection criteria with the same ease whether or not the objects are indexed that way or not. The ‘only’ tradeoff is performance.

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. This process is referred to as committing the database or committing the changes. The database's `commit` method is used to commit changes.

A database object needs to be closed, by using the database's `close` method, before the application terminates. If any objects have been added or changed, then the database needs to be updated before it is closed.

Creating and Opening a New Database

Creating and initializing an environment-specific derivative of CNeoDatabase is straightforward, as is specifying a database path and opening it. Consider the routine `createFile`. In practice, the creation of a native database object is usually done in the constructor of the environment-specific document class. It is shown here for completeness.

```
CNeoDatabase *createFile(Str63 aName, short aVolNum, long aDirID)
{
 CNeoDatabaseMA *    database;

 /**
  ** Allocate a new database object.
  **/
 database = new CNeoDatabaseNative(kApplicationSig, kAppFileType);

 /**
  ** Create a new database, and open it normally.
  **/
 database->SpecifyHFS(aName, aVolNum, aDirID);
 database->create();
 database->open(fsRdWrPerm);

 return database;
}
```

The first two calls create the database object and initialize it. The arguments passed to the constructor indicate the creator and type of the database. The creator should be the application signature. The type is generally the document file type of the application. The argument accepted by the constructor in your development environment may be different. Refer to the discussion of that environment's database class for more information.

There are several ways to specify a database's path. The method `SpecifyHFS` is used here. However, we have also seen `SFSpecify` used in the description of CNeoDoc above. Several other database path specification methods are also available. They are for the most part interchangeable, though the set available differs depending on which development environment you're using. Use the one that best suits the needs of your application.

Once the database name has been specified, use the `create` method to create the database.

Once the database has been created it can be opened. The argument to `open` specifies whether the database is to be opened read only or for update. Obviously, a new database needs to be opened for update.

Opening a Pre-Existing Database

Opening a pre-existing CNeoDatabase is similar to opening a new one. Consider the routine `openFile` below. In practice, the creation of a native database object is usually done in the constructor of the environment-specific document class. It is shown here for completeness.

```
CNeoDatabase *openFile(const FSSpec *aFileSpec)
{
 CNeoDatabaseNative *    database;

 /**
  ** Allocate a new database object and initialize it.
  **/
 database = new CNeoDatabaseNative(kApplicationSig, kAppFileType);

 /**
  ** Send the database a SpecifyFSSpec() message to set
  ** up the name, volume, and directory.
  **/
 database->SpecifyFSSpec(aFileSpec);

 /**
  ** Send the database an open() message to open it.
  **/
 database->open(NeoReadWritePerm);

 return database;
}
```

The first two calls create a database object native to the development environment and initialize it. The arguments passed to the `CNeoDatabaseNative` constructor indicate the creator and type of the database.

There are several ways to specify a database's path. The method `SpecifyFSSpec` is used here. However, we have also seen `SFSpecify` used in the description of CNeoDoc above. Several other database path specification methods are also available. They are for the most part interchangeable, though the set available differs depending on which development environment you're using. Use the one that best suits the needs of your application.

Finally, once the database location is specified it can be opened. The argument to `open` specifies whether the database is to be opened read only or for update.

Committing a CNeoDatabase

When the permanent state of a database's objects in memory changes, these changes need to be committed to the database. This commit process is called database synchronization and is performed by the database object's `commit` method. Synchronizing a NeoAccess database is fast because only those objects that have been marked dirty need to be updated on disk.

```
 Boolean compress    = FALSE;

 /**
  ** Synchronize the database's contents.
  **/
 database->commit(compress);
```

The single argument to this method indicates whether the commit process should attempt to reduce the size of the database on disk. If the value of this argument is `TRUE`, then the

system will try to relocate each object in the database to a location closer to the beginning. If objects at the end of the database can be relocated, then the overall size of the database can be reduced.

Closing a CNeoDatabase

The database object needs to be closed and deleted before the application terminates. If any objects have been added or changed, then the database needs to be updated before it is closed. The routine `closeFile` is called after it has been determined whether the database is dirty and, if so, whether the user has requested that the database's contents be updated.

```
void closeFile(const Boolean aCommit)
{
 /**
  ** Commit changes to the database's contents upon request.
  **/
 if (aCommit)
     database->commit(FALSE);

 /**
  ** Send the database a close() message to close it.
  **/
 database->close();

 /**
  ** Delete the database object now that we're done with it.
  **/
 delete database;
 database = nil;
}
```

The argument `aCommit` indicates whether the changes to the database should be committed before the database is closed. Once the database is closed, the database object can be deleted.

Making an Object Permanent

Persistent objects become permanent by giving them an identity and adding them to a database. The database's `addObject` method is used to perform this task.

```
 CNeoPersist *   object;
 CNeoDatabase *  database;

 database->addObject(object);
```

If the object's identity is zero, then an id unique to the database is assigned automatically when the object is added to the database.

Locating an Object

The ability to locate objects based on specific selection criterion is probably the single most important function of a database system. The interface to this mechanism needs to balance simplicity with power. NeoAccess provides an extensible interface that offers both.

The CNeoPersist class includes `Find`, `FindEvery` and `FindByID` methods for locating an object or group of objects. These methods can search a specific class of objects, or even search a base class and all of its subclasses. Subclasses of CNeoPersist can include additional methods that provide similar capabilities tailored to the specific needs of those subclasses.

The easiest and most common method for locating objects is `FindByID`. The following example illustrates several possible uses:

```
void *CountObj(CNeoPersist *aObject, void *aParam)
{
 (*(short *)aParam)++;

 return nil;
}

Examples(void)
{
 short           count       = 0;
 NeoID           classID     = kAppSpecificID;
 NeoID           objectID;
 CNeoNameSelect  key("\pTobias");
 CNeoArray *     array;
 CAppSpecific *  object;
 CNeoDatabase *  database;

 /**
  ** Locate a single object that is a member of the classID
  ** class and which has the indicated object ID.
  **/
 object = CNeoPersist::FindByID(database, classID, objectID, FALSE);

 /**
  ** Locate a single object that has a base class of classID
  ** and which has the indicated object ID.
  **/
 object = CNeoPersist::FindByID(database, classID, objectID, TRUE);

 array = new CNeoArray;
 array->IArray(sizeof(CNeoPersist *));
 /**
  ** Locate all those objects that are a member of the classID
  ** class and which has the indicated object ID.
  **/
 CNeoPersist::FindByID(database, classID, objectID, FALSE, nil, array);

 /**
```

```
 ** Count the number of objects that have a base class of classID
 ** and which has the indicated object ID.
 **/
CNeoPersist::FindByID(database, classID, objectID, TRUE, CountObj,
```

```
                        &count);

 /**
  ** Locate all those objects that have a base class of classID
  ** and which has the indicated name value.
  **/
 CNeoPersist::Find(database, classID, &key, FALSE, nil, array);
}
```

The first call to `FindByID` locates in `database` a single object of class CAppSpecific having an object ID of `objectID`. The fourth argument indicates that subclasses of CAppSpecific should not be searched. The fifth argument can be a pointer to a function that is called for each object found. When this argument is `nil`, as it is in this first example, then no function is called. The final argument is also `nil`, indicating that only the first object found should be returned. If multiple objects having the given id exist in this class, then it's not possible to predict which of them will be returned.

The second call to `FindByID` is similar to the first except that if the object is not found in the base class, CAppSpecific, then each of the subclasses of CAppSpecific are searched until a matching object is found.

The third call to `FindByID` is an example of searching for multiple objects from a single class. Notice that the return value from this call is ignored. This is because the second from last argument is `nil` and the last argument refers to an array object into which pointers to all objects having the given object id are added.

The fourth example of `FindByID` simply counts the number of objects of the given base class that have the specified object id. The function `CountObj` is called for every matching object.

The final example uses the method `Find` to locate those objects of the specified class having the name "Tobias". The constructor for `key` initializes the selection to look for the name Tobias. This key is passed to `Find` to locate the objects.

This final example is actually quite interesting because it shows the true power of NeoAccess's searching capabilities. The fact is that there isn't a single line of code in all of NeoAccess that knows anything about how to locate CAppSpecific objects based on a name field. And the `Find` method of CNeoPersist doesn't know anything about selection criteria based on a textual value. The searching mechanism of NeoAccess uses selection criteria objects having a base class of CNeoSelect and persistent btree classes having a base class of CNeoNode to search lists of objects sorted in orders supported by the index class based on selection criteria supported by the selection class. CNeoNode can be subclassed to organize objects in some other application-specific fashion. CNeoSelect can be subclassed to create other types of selection criteria.

Removing an Object

Objects come and objects go. Eventually your application may need to remove an object from a database. The database's `removeObject` method takes care of all the details.

```
CNeoPersist *   object;
CNeoDatabase *  database;

/**
 ** Send the database the removeObject message
 ** to remove the indicated object.
 **/
database->removeObject(object);
```

This method removes the object from the database that it has been in and frees up the file space that the object used to occupy.

NOTE

As is the case with all changes to a database, an object is permanently removed from a database only after the database has been committed.

An object that has been removed from a database is still a persistent object. It needs to be disposed of when the application no longer refers to it, though it can also be made permanent again at some later point.

Fast File Space Allocation

Some applications need to be able to allocate file space as quickly as possible. The `setFastAllocation` method allows an application to set the allocation mode of the database to provide database space allocation at standard speeds or extremely fast speeds. Fast allocation is achieved by ignoring the current state of the free list and simply extending the length of the database to accommodate the allocation request.

Concurrency in a Multi-Threaded Environment

Some execution environments include cooperative threads of execution in a single process. NeoAccess includes an optional feature which allows multiple threads to be in the database at any one time. This concurrency is managed through the use of semaphores which manage concurrent access to shared objects and constructs. Database objects are protected using a multiple-reader/single-writer semaphore. Each method 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 lock reference 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` methods are used to obtain and free database lock references.

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. (Though your application's mileage may vary.)

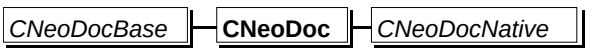
The object cache uses memory not otherwise being used by the application. This cache can become quite large and consume a significant portion of your application's memory. By default, the cache will use everything it can get its hands on. In environments where virtual memory is not available or where the memory allocation for an application is otherwise bounded, NeoAccess provides a mechanism for freeing objects in the cache when memory is needed. NeoAccess also includes a mechanism for limiting the amount of memory used by the object cache.

The cache can be purged by calling the `purge` method of a CNeoDatabase object. The single argument to `purge` is a pointer to a long 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 number of times that low memory situations occur while trying not to reduce the usefulness of the object cache.

Some execution environments, such as pointer-based environments lacking robust virtual memory systems, require that the size of the object cache be bounded. The size of the cache can be bounded. NeoAccess will limit the size of the object cache to something close to the amount specified by the static variable `CNeoPersist::FCacheSize`. If an allocation would cause the cache to exceed this limit, then the `purge` method of all open NeoAccess databases will be called to reduce the cache so that the allocation can occur.

CNeoDoc

Heritage



The Heritage and Ancestry of CNeoDoc

NeoAccess is a persistence framework that has been designed to extend standard application frameworks. As such, there are several NeoAccess classes which must interface with the application framework. CNeoDoc is one of these classes. It is an environment-neutral class. The immediate base class of CNeoDoc varies depending on the application framework within which NeoAccess is being built. The environment-specific header file for the framework will define the typedef `CNeoDocBase` to be a synonym for this base class.

Some application frameworks include a document class from which application-specific document classes are derived. Some methods of

this base class may need to be overridden in order to provide more natural and complete support for CNeoDoc subclasses. These overrides should be implemented in an environment-specific subclass of CNeoDoc. The environment-specific header file for the framework will define the typedef `CNeoDocNative` to be a synonym for this subclass.

Introduction

The abstract base class CNeoDoc is designed to be a subclass of the native application framework's document class. CNeoDoc includes a few pure virtual functions that are further overridden by environment-specific subclasses of CNeoDoc. These virtual functions provide an environment-neutral interface for accessing the list of currently open documents.

NOTE

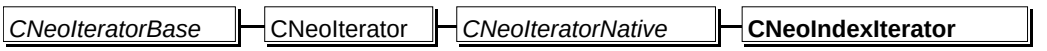
Some application frameworks don’t require the use of documents. The CNeoDoc class is not used in these situations. However, in these environments the benefits provided by CNeoDoc need to be done manually instead.

Using CNeoDoc

Whenever a new document is created it is assigned a unique four-byte identity and threaded into a list of currently open documents. The document is removed from the list when it is deleted. The purpose of this list is to allow documents to be accessed by document ID. The static member function of CNeoDoc, `FindByID`, is used to locate a document using a document ID. `ResetDocListHead` is called when the current database setting is changed, usually when one document window is deactivated and another activated. The `PurgeCache` method is called when a certain amount of memory needs to be obtained by purging the object caches of the databases associated with open documents. The `getDatabase` and `setDatabase` methods are used for locating and setting the database object associated with a document.

CNeoIndexIterator

Heritage



The Heritage and Ancestry of CNeoIndexIterator

Introduction

There are several iterator classes that come standard with NeoAccess. (See the discussion of CNeoIterator for more information on NeoAccess iterator classes.) These classes greatly simplify the manipulation of btrees by application developers. They all have a common base class of CNeoIterator which supports a set of operations that are similar to those provided by most iterators. These operations 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 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.

CNeoIndexIterator is used to iterate over all matching objects of a base class and optionally all its subclasses.

Using CNeoIndexIterator

CNeoIterator and its subclasses are keyed iterators that provide for the iteration over a list of entries in a NeoAccess btree. These btrees can be index trees, parts lists, or any other type of btrees derived from nodes based on CNeoNode and its subclasses. These iterators support operations that are common to most iterator classes. Initialize a newly created index iterator by using `CNeoIndexIterator`. To move to the next entry in the list use `nextObject`. To move to the previous entry in the list use `previousObject`. To get the

current object call `currentObject`.

CNeoIterator

Heritage

The Heritage and Ancestry of CNeoIterator

CNeoIterator is an environment-neutral class. The immediate base class of CNeoIterator varies depending on the application framework within which NeoAccess is being built. The environment-specific header file for the framework will define the typedef `CNeoIteratorBase` to be a synonym for this base class.

Some application frameworks include abstract base classes from which all iterator classes inherit properties. Some methods of this base class may need to be overridden in order to provide more natural and complete support for CNeoIterator subclasses. These overrides should be implemented in an environment-

specific subclass of CNeoIterator. The environment-specific header file for the framework will define the typedef `CNeoIteratorNative` to be a synonym for this subclass.

Introduction

The abstract base class CNeoNode and its various derivatives can all be used to construct various types of collection classes which are generally called extended binary trees. Arrays and linked lists are other examples of collection classes which most developers are probably familiar with. There are in fact ten’s or maybe even a 100 or more different kinds of collection classes and derivatives that are used in systems today. Some are as basic as linear arrays, while others are as complex as heuristic neural nets. The diversity of collection classes is due to the different requirements that each application has in terms of performance and resource

utilization. Good developers, like all other good craftsmen, use the right tool the for the job.

But, as is often the case, along with diversity comes the potential for confusion and excessive complexity. In keeping with the principle of encapsulation, object developers should be able to manipulate the contents of various types of collections using a single common interface without regard for how the contents are organized internally. This is the motivation behind iterator classes.

NeoAccess uses btrees as its primary collection class because they have unique properties that make them truly ideal for database systems. NeoAccess's iterator classes greatly simplify the manipulation of btrees by application developers. The iterator classes are

environment-neutral so they act as a natural subclass of the native application framework's iterator class, if any. Because CNeoIterator is an environment-neutral class it may, if applicable, be a subclass of one of the application framework's iterator classes.

There are several iterator classes that come standard with NeoAccess. They all have a common base class of CNeoIterator. These iterator classes support operations that are similar to those provided by most iterators. These operations 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.

CNeoIterators can also be used to iterate over a subset of items that match a NeoAccess select key and to apply a function to each item in the

list. (See the description of CNeoSelect for more information on select keys.) NeoAccess iterators are often called **keyed iterators** because of this unique capability to iterate over a subset of a collection based on an abstract select key.

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. (See the CNeoIndexIterator and CNeoIDListIterator class for more information on the special capabilities of these subclasses.)

Using CNeoIterator

CNeoIterator and its subclasses are keyed iterators that provide for the iteration over a list of entries in a NeoAccess btree. These btrees can be index trees, parts lists, or any other type of btrees derived from nodes based on CNeoNode and its subclasses. Iterators support

operations that are common to most iterator classes. Initialize a newly created iterator by using the CNeoIterator constructor. To move to the next or previous object in the list use `nextObject` or `previousObject`, respectively. To get the current object call `currentObject`. Use more to test whether there are additional objects in the list beyond the current one. To apply a function to all objects from the current one forward call `doUntil`. This class also includes several access methods to determine the direction of the iterator (forward or backward), to obtain a pointer to the select key associated with the iterator, or to get a pointer to the current leaf node. NeoAccess iterators are environment-neutral so the native subclass of CNeoIterator may support additional iterator operations native to those environments. Subclasses may override some of these methods or extend this repertoire of operations and more naturally

extend the capabilities of the native application framework.

Subclassing CNeoIterator

Concrete subclasses of CNeoIterator may need to override some of its methods. These methods include `advance`, `currentObject`, `more`, `nextObject`, `previousObject` and `reset`.

CNeoMetaClass

Heritage

The Heritage and Ancestry of CNeoMetaClass

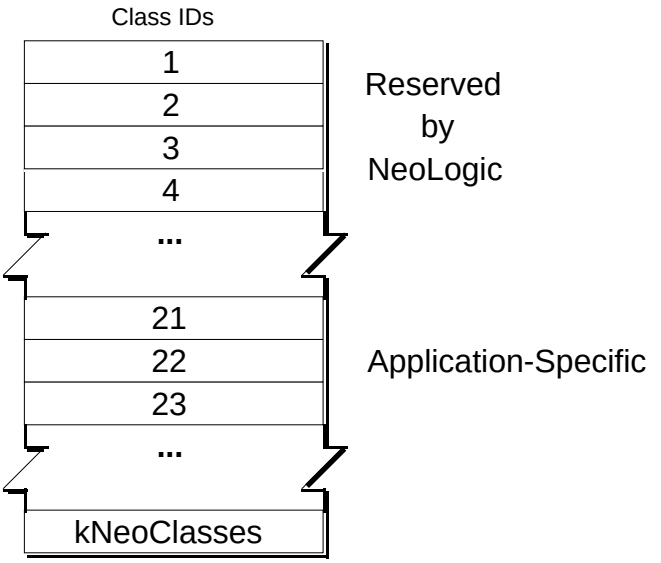
Introduction

A metaclass is an object-oriented concept. Metaclass objects define or describe other classes. NeoAccess refers to metaclass objects to obtain information about persistent classes. In particular, a metaclass provides the following information:

- the class's class ID,
- a list of class IDs that refer to the class's superclass's,
- the name of the class,
- the number of index keys the class maintains in the database,
- the index type of each of the class's

indices,

- a pointer to a function that creates and initializes an instance of the class,
- a pointer to a function that manages selection objects for index classes.



The Layout of the Metaclass Table

Adding to the Metaclass Table

NeoAccess relies on an array of metaclass objects to obtain information about persistent classes. NeoAccess initializes part of this array automatically when the application starts up. Your application class must complete this initialization process by adding application-specific metaclass objects to the array.

Consider the following application class constructor:

```
const long   kAppSpecificID  = 21;

CMyApp::CMyApp(void)
```

```
 : CNeoAppNative()
{
 CNeoMetaClass * meta;

 /**
  ** Initialize the metaclass table with application-specific classes.
  **/
 // Add CNeoAppSpecific class to metaclass table
 meta = new CNeoMetaClass(kAppSpecificID, kNeoOffspringID,
                          kAppSpecificName, CAppSpecific::New);
 meta->setKey(kNeoPartIndex, kNeoSecondaryIndex);
 …
}
```

The CMyApp constructor adds information about its application-specific classes to the metaclass table. CNeoMetaClass objects are created, as all other objects are, by using the `new` operator.

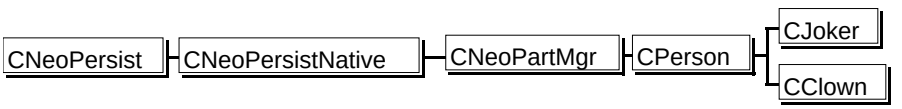
The arguments to the CNeoMetaClass constructor refer to the class ID of the class, the class ID of the class's superclass, a pointer to the `New` function of that class and the name of the class. As a convenience, this initialization method also adds the metaclass to the metaclass table.

By default, the primary index of application-specific persistent classes is CNeoIDIndex. The constructor initializes metaclass to use this index class. To specify a different primary index or a specify additional secondary indices, the `setKey` method of the metaclass is used. In the sample code shown above, CNeoAppSpecific objects are sorted primary in ascending order by ID and secondarily by parent ID.

It is sometime useful to sort all objects having a given base class in a single index. A **consolidated index** is one which refers to all objects having a common base class. This is in contrast to the usual practice of organizing all objects of each specific leaf class in a separate index. A consolidated index is created by configuring all metaclass objects of classes having the common base class to have a index root value equal to the class ID of the base class. This is done by passing the class ID of the common base class as the third argument to the `setKey` method for each of the concrete classes having that base class. See the discussion of the Laughs' application constructor in the Tutorial section for an example of how a consolidated index might be used.

KEY POINT

The metaclass object for all superclasses must have already been added to the table before adding a metaclass for a persistent class.



Sample Inheritance Tree for Persistent Classes

Consider, for example, the simple class tree shown above. The metaclass for CNeoPersist is created automatically by the CNeoApp constructor. Metaclass objects for CNeoPart must be added next, followed by CPerson, and finally CJoker and CClown.

Multiple Inheritance

Some development environments allow classes to be derived from more than one parent class. Metaclasses provide a mechanism for defining the inheritance tree as it relates to persistence. By convention this tree often mimics the C++ inheritance tree, though there are no constraints within NeoAccess that requires it to. Developers are free to define the superclass of a persistent class without regard to the actual C++ ancestry. Multiple superclasses can also be assigned to a persistent class whether or not the C++ class has more than one parent. The primary superclass of a persistent class is given to the CNeoMetaClass constructor. Additional superclasses can be added by using the `setSuperclass` method for the metaclass of a class. The maximum number of superclasses that any class can have is defined by the compile-time constant `kNeoMaxSuper`. The value of this constant can be configured by changing this value.

The `getOne` Function

A metaclass object maintains pointers to two functions for manipulating instances of the class: `getOne` and `KeyManager`. The value of `getOne` needs to be set by all concrete persistent classes (that is, classes for which instances may be instantiated or added to the database). The `getOne` function pointer should be set to a function that allocates, initializes and returns a pointer to an object of that class. For example, the method

`CAppSpecific::New` below creates and initializes an instance of the CAppSpecific class.

```
CNeoPersist *CAppSpecific::New(void)
{
 /**
  ** Allocate and initialize and return a new instance.
  **/
 return new CAppSpecific();
}
```

The KeyManager Function

The other function pointer, `KeyManager`, needs to be set only for index classes. A key manager routine supports a set of operations on selection criteria objects used by an index class. Only two operations are currently used by NeoAccess. They are `kNeoCanSupport` and `kNeoGetKey`.

The `kNeoCanSupport` operation is called with three arguments. The first is of course the operation selector. The second argument is a pointer to a select key object. When called with the `kNeoCanSupport` operation, the `KeyManager` method should return a Boolean value indicating whether or not this index class can support a binary search using this type of select key. If the second argument is nil, then every object will match so `KeyManager` should return `TRUE`. If the index can not support a given select key, then `KeyManager` should call `NeoInherited`, being careful to make sure the third argument passed to `KeyManager` is passed along with the initial two. This third argument is a long integer having a value of either zero or 1. If it is 1 and the key is a complex selection criterion, then `CNeoNode::KeyManager` may optimize the criterion by rearranging the terms of key.

Consider the `KeyManager` method of the CNeoIDIndex class shown below.

```
void *CNeoIDIndex::KeyManager(const NeoKeyOp aOp, ...)
{
 va_list         argptr;                 // pointer to argument list.
 NeoSelectType   selectType;
 long            optimize;
 void *          value   = nil;
 CNeoSelect *    key;
 CNeoPersist *   object;

 va_start(argptr, aOp);
 switch (aOp) {
 case kNeoCanSupport:
     key = va_arg(argptr, CNeoSelect *);
     selectType = (key ? key->getSelectType() : pNeoID);
     if (selectType == pNeoID)
         value = (void *)(unsigned char)TRUE;
     else {
         optimize = va_arg(argptr, long);
         value = NeoInherited::KeyManager(kNeoCanSupport, key,
             optimize);
     }
     break;
```

```
 case kNeoGetKey:
     object = va_arg(argptr, CNeoPersist *);
     value = (void *)new CNeoIDSelect(object->fID);
     break;
 }
 va_end(argptr);

 return value;
}
```

The first argument of this method is the operation type. Its value indicates the operation this routine is being called for. Each additional argument is obtained by using the `varargs` support of the native development environment.

The `kNeoCanSupport` operation takes two additional arguments; a pointer to a select key and a long `Boolean` value. The implementation of this operation should return `TRUE` if nodes of this type can support a binary search using the given key. The value of the last argument indicates whether the inherited `KeyManager` routine should try to optimize the terms of the given key.

The `kNeoGetKey` operation can be called with one additional argument. The second argument is a pointer to a persistent object. The `KeyManager` function should return a select key that can be used to uniquely locate the given object in the index.

Metaclasses for Index Classes

The steps involved in creating metaclass objects for index classes are the same as for other classes except that the last argument to the CNeoMetaClass constructor should refer to the `KeyManager` routine for that index class. Consider the code below:

```
// Add CNeoIDIndex class to metaclass table
metaclass = new CNeoMetaClass(kNeoIDIndexID, kNeoNullClassID,
                        kNeoIDIndexName, CNeoIDIndex::New,
                        CNeoIDIndex::KeyManager);
```

Notice that the parent class of CNeoIDIndex is set to `kNeoNullClassID`. This indicates to NeoAccess that this is a support class that is not itself indexed, which index classes never are.

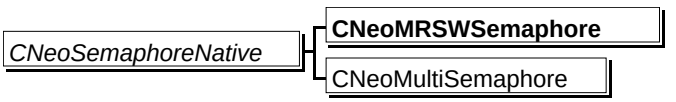
Using CNeoMetaClass

NeoAccess relies on an array of metaclass objects to obtain information about persistent classes. NeoAccess initializes part of this array when a database is opened. Your application must complete this initialization process by

adding application-specific metaclass objects to the array.

CNeoMRSWSemaphore

Heritage



The Heritage and Ancestry of CNeoMRSWSemaphore

Introduction

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. (See the discussions of the CNeoThread and CNeoThreadNative classes in the reference section of this document for more information on multi-threaded execution environments.)

S**emaphores** are used to restrict entry into a critical section of code. They control concurrent access to shared resources in multi-threaded

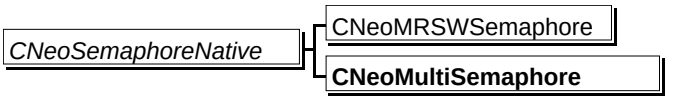
runtime environments. NeoAccess includes includes a set of environment-specific semaphore class which are the abstract base class for a set of special-purpose semaphore classes. CNeoSemaphoreMac is the environment-specific base semaphore class used in all Macintosh-based development environments except PowerPlant.

NeoAccess's cooperative multi-threading support is enabled when built with the compile time symbol `qNeoThreads` defined. When operating in this type of environment, database objects are protected using a multiple-reader/single-writer semaphore of type CNeoMRSWSemaphore. Each method 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 semaphore's `lock` and `unlock` methods are used to obtain and free these lock references.

CNeoMultiSemaphore

Heritage



The Heritage and Ancestry of CNeoMultiSemaphore

Introduction

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. (See the discussions of the CNeoThread and CNeoThreadNative classes in the reference section of this document for more information on multi-threaded execution environments.)

NeoAccess's cooperative multi-threading support is enabled when built with the compile time symbol `qNeoThreads` defined. When

operating in this type of concurrent environment, the individual entries of persistent node objects need to be protected to prevent the situation where more than one thread tries to load the object referred to by a node entry. Concurrency of access to node entries is managed using a CNeoMultiSemaphore semaphore. Each node object has a single CNeoMultiSemaphore semaphore which is used to manage access to all the entries of the node.

CNeoPartListIterator

Heritage

The Heritage and Ancestry of CNeoPartListIterator

Introduction

There are several iterator classes that come standard with NeoAccess. (See the discussion of the CNeoIterator class for more information on NeoAccess iterator classes.) These classes greatly simplify the manipulation of btrees by application developers. They all have a common base class is CNeoIterator which supports a set of operations that are similar to those provided by most iterators. These operations include the ability to traverse the collection forwards and backwards, 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

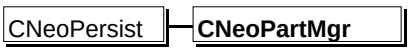
beginning of the list again. NeoAccess iterators are often called **keyed iterators** because of their unique capability to iterate over a subset of a collection based on an abstract select key.

Using CNeoPartListIterator

CNeoPartListIterator is the base class used to iterate over parts lists. It supports operations which are common to most iterator classes. Initialize a newly created part list iterator by using `CNeoPartListIterator`. To move to the next entry in the list use `nextObject`. To move to the previous entry in the list use `previousObject`. To get the current object call `currentObject`.

CNeoPartMgr

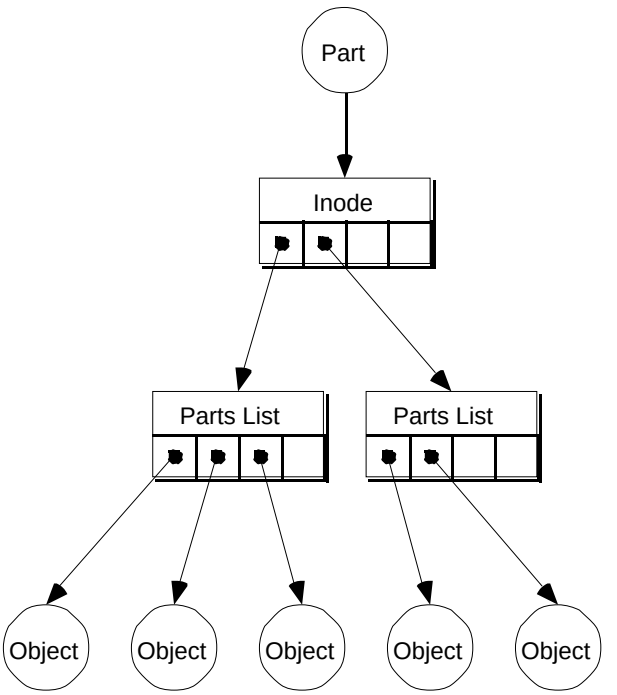
Heritage



The Heritage and Ancestry of CNeoPartMgr

Introduction

Relational systems organize data in tables on which associative lookups can be performed. While NeoAccess supports a powerful form of associative lookup, probably the biggest advantage it has over relational systems is the speed and ease with which it is able to organize and access objects in lists.



A Part with a Direct Parts List

The use of this other type of access is often called a **parts explosion**. NeoAccess includes a class, ENeoPartMgr, which can be used to group objects into a collection called

a **parts list**. Accessing objects in lists is sometimes called **referential access**. Application-specific grouping objects should have a have a data member of class ENeoPartMgr to which part list management responsibilities are delegated.

CNeoPartMgr is a “convenience class” which delegates responsibility for managing part lists to a data member of type ENeoPartMgr. This class is included in the standard NeoAccess release for backward compatibility reasons as well as because is an excellent example of how a CNeoPersist subclass might add part list management capabilities by using ENeoPartMgr.

Using CNeoPartMgr

Initialize a newly created part by using the CNeoPartMgr constructor. Use `addToList` to add an entry to the parts list and `deleteFromList` to delete an entry. Use

deleteList to remove the entire list from the database and delete the part's reference to it in memory. Use `doUntilPart` to apply a function to every entry in the list. Call `getListCount` to obtain a count of the number of entries in a parts list.

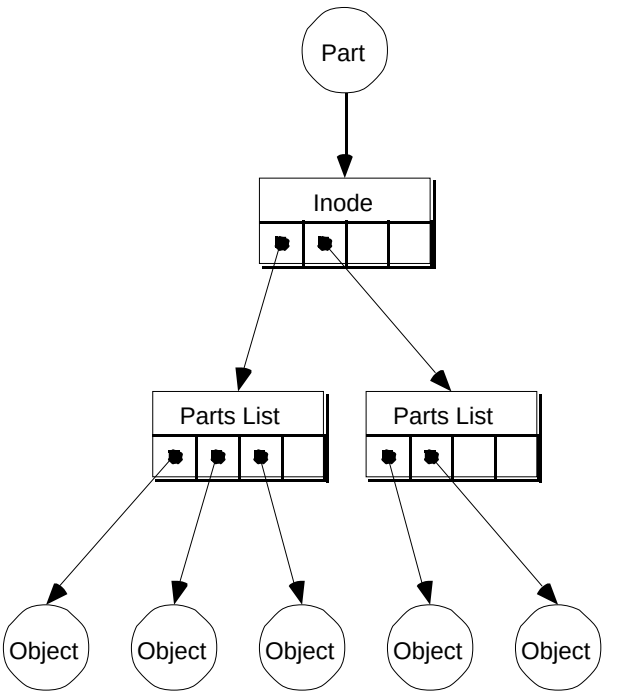
ENeoPartMgr

Heritage

The Heritage and Ancestry of ENeoPartMgr

Introduction

Relational systems organize data in tables on which associative lookups can be performed. While NeoAccess supports a powerful form of associative lookup, probably the biggest advantage it has over relational systems is the speed and ease with which it is able to organize and access objects in lists.



A Part with a Direct Parts List

The use of this other type of access is often called a **parts explosion**. NeoAccess includes a class, ENeoPartMgr, which can be used to group objects into a collection called a **parts list**. Accessing objects in lists is sometimes called **referential access**. Application-specific grouping objects should have a have a data member of class ENeoPartMgr to which part list management responsibilities are delegated.

Subclassing CNeoPartMgr

The class ENeoPartMgr provides a grouping property to classes that include ENeoPartMgr data members. This class has been designed to minimize the need to subclass it in order to support application-specific part list types. For example, it makes no assumptions about the class of its associated parts list or the base class of its fruit objects. Developers who feel the need to subclass ENeoPartMgr for this purpose should look instead into whether the parts list class is in fact the class that needs to be modified.

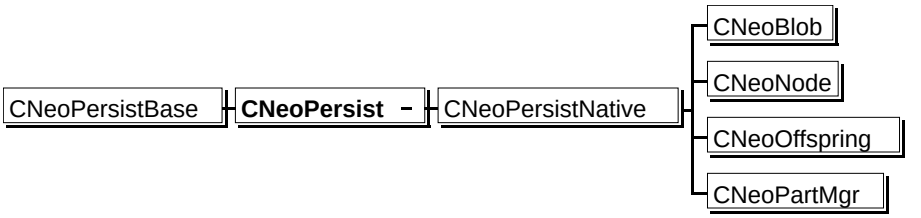
Using ENeoPartMgr

Initialize a newly created part by using the ENeoPartMgr constructor. Use `addToList` to add an entry to the parts list and `deleteFromList` to delete an entry. Use deleteList to remove the entire list from the

database and delete the part's reference to it in memory. Use `doUntilPart` to apply a function to every entry in the list. Call `getListCount` to obtain a count of the number of entries in a parts list.

CNeoPersist

Heritage



The Heritage and Ancestry of CNeoPersist

CNeoPersist is an environment-neutral class. The immediate base class of CNeoPersist varies depending on the application framework within which NeoAccess is being built. The environment-specific header file for the framework will define the typedef `CNeoPersistBase` to be a synonym for this base class.

Some application frameworks include abstract base classes from which most application-specific classes inherit properties. Some methods of this base class may need to be overridden in order to provide more natural and

complete support for CNeoPersist subclasses in these environments. These overrides should be implemented in an environment-specific subclass of CNeoPersist. The environment-specific header file for the framework will define the typedef `CNeoPersistNative` to be a synonym for this subclass.

Introduction

CNeoPersist is an abstract base class that provides its subclasses with the ability to preserve their permanent state in a CNeoDatabase database and to share objects between application components. Object integrity in this shared environment is supported through the use of shared and exclusive locking methods. By default, permanent objects are organized in a database according to the object class and sorted within each class based on object identity, though virtually any organization of objects can also be supported.

Objects in a database can be located quickly and easily using binary search algorithms.

Using CNeoPersist

Object persistence and integrity is provided by the abstract class CNeoPersist. A newly created persistent object is created through the use of the `new` operator. The object can be

made permanent by using the database's `addObject` method. Making an object permanent involves associating the object with a database and assigning it an identity. An identity is simply a value that, in conjunction with the class of the object, can be used find the object in the database. In order to unambiguously refer to it, an object should have a unique identity within its class within its database. However, ambiguity may be desirable in some situations, so finding all objects of a given base class having a given identity is also supported.

Typically, permanent objects are sorted within their class in ascending id order. This allows the use of binary searches to locate objects given an id value. However custom indexing strategies are also supported.

The database object’s `removeObject` method breaks the association between database and

object. The process of removing an object from a database involves deleting references to it in the indices and freeing the file space that had been allocated for it in the database.

The `unrefer` method supports the object's reference count mechanism. This method decrements the reference count. If the count goes to zero and the object is not locked, then the object is purgeable. Purgeable objects can be freed during purging. Purging, which is usually invoked when memory is low, involves traversing the database's data structures freeing purgeable objects encountered until a large enough block of memory can be allocated.

Subclassing CNeoPersist

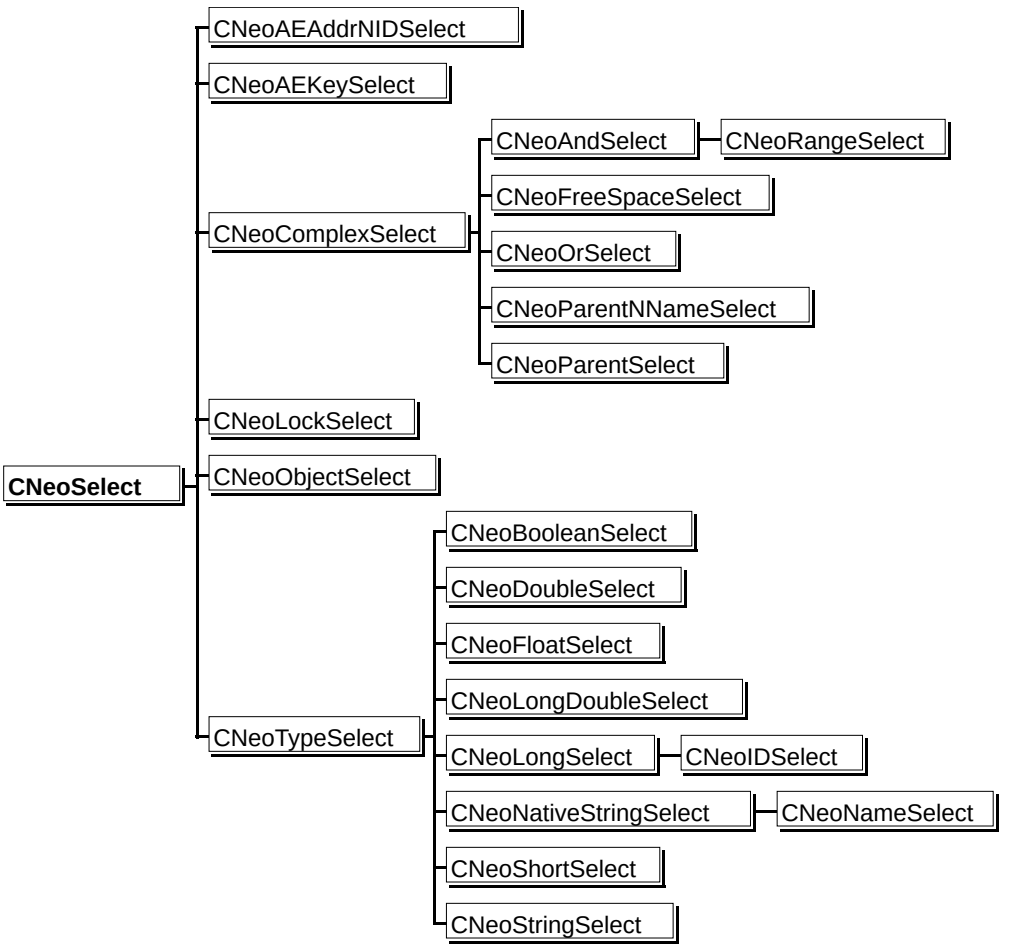
Concrete subclasses of CNeoPersist need to override some methods inherited from CNeoPersist in order to take full advantage of the inherited capabilities. The table given below defines which methods should be overridden

and when.

Override	Only if…
`add`	instances refer directly to other persistent objects or data
`commit`	instances refer directly to other persistent objects
`getClassID`	is a concrete class
`getFileLength`	is a concrete class
`getValue`	instances have data members accessible via tags
`New`	is a concrete class
`purge`	refers directly to other persistent objects or data
`readObject`	instances have data members
`remove`	instances refer directly to other persistent objects or data
`verify`	there are assertions to be made about instance state
`writeObject`	instances have persistent data members

CNeoSelect

Heritage



The Heritage and Ancestry of CNeoSelect

Introduction

By this point you probably already recognize the power and versatility provided by

NeoAccess’s flexible index classes. Indices provide developers with great flexibility in

organizing objects. But the real power that a database system should deliver is a versatile accessing mechanism.

The search mechanisms provided by NeoAccess use a very 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.

Sounds kind of whiz-bang but you're a little fuzzy on the concept, right? Let's walk through a couple of examples to see what this beast really is and how it's used. Consider the CNeoPersist method `FindByID`. The implementation of this method is actually quite simple:

```
void *CNeoPersist::FindByID(CNeoDatabase *aDatabase,
    const NeoID aClassID, const NeoID aID, const Boolean aDeeply,
    NeoTestFunc1 aFunc, void *aParam, const NeoLockType aLock)
{
    CNeoDatabase *  database    = aDatabase ? aDatabase : gNeoDatabase;
    CNeoIDSelect    key(aID);

    return database->findObject(aClassID, &key, aDeeply, aFunc, aParam,
```

```
        aLock);
}
```

CNeoIDSelect is a subclass of CNeoSelect. It is a type of **selection criterion** (also called **select key**) that is used to locate objects based on their identity. In the code shown above, an id select key named `key` is allocated on the stack and initialized with the value of `aID`.

The second argument to CNeoDatabase's `findObject` method is an abstract selection criterion. A calendar application may include a CMonth class that has a method `FindByDate`. The implementation of this method would be very similar to the implementation of `FindByID`. Instead of declaring a CNeoIDSelect key, it would declare one of class CDateSelect. The database object's `findObject` method is equally capable of satisfying both requests.

The next mystery is how CNeoDatabase satisfies a selection request using an abstract selection key. Adding a class entry to a database involves making use of most of the information maintained by the metaclass for that persistent class. The process of locating objects in the database begins by using the class id parameter of `CNeoDatabase::findObject` to locate the proper class entry. A class entry knows about all the indices of that class. The `kNeoCanSupport` operation of the `KeyManager` function of each index class (beginning with the primary index) is queried to determine whether that index can be used to perform a binary search using the given select key.

If the selection criterion is not supported by one of the indices, then the object list is traversed serially using that index. As the search processes, the `compare` method of the selection key is called for objects in the list. For every object that matches the select key, the `aFunc` parameter of `CNeoDatabase::findObject` is called until it returns a non-`nil` value. If `aFunc` is `nil` but `aParam` is not, then `aParam` must refer to an array into which every matching object is found. If both `aFunc` and `aParam` are `nil`, then a pointer to the first matching object is returned by `CNeoDatabase::findObject`.

If the selection criterion is not supported by any of the indices, then the object list is traversed serially using the primary index. The `compare` method of the select key is called for each object in the list. For every entry or object for which `kNeoExact` is returned, the `aFunc` parameter of `CNeoDatabase::findObject` is called until it returns a non-`nil` value. As is the case when doing binary searches, if `aFunc` is `nil` but `aParam` is not, then `aParam` must refer to an array into which every matching object is found. If both `aFunc` and `aParam` are both `nil`, then a pointer to the first matching object is returned by `CNeoDatabase::findObject`.

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.

So supporting the ability to locate objects using a specific kind of selection criterion you simply instanciate some subclass of CNeoSelect which is to be the criterion and pass it to the database's `findObject`. This can be made even simpler by defining a "convenience" method much like `FindByID`.

Using CNeoSelect

If a selection criterion is supported by an index, then an object list can be traversed serially using that index. As a search processes, the `compare` method of the select key is called for objects in the list. If a selection criterion is not supported by any of a class's indices, then the object list of that class is traversed serially using its primary index.

Application developers typically don't manipulate select keys directly. Database queries are usually performed by calling "convenience" methods like `FindByID`, `FindByParent` and application-specific

methods like these. Select objects can be allocated on the stack as local variables. Key objects obtained by using the `kNeoGetKey` operation of an index class's `KeyManager` method should be properly disposed of.

Subclassing CNeoSelect

Subclasses of CNeoSelect override the `compare` method to perform comparisons based on the type of selections that they support. Some subclasses that support memory management optimizations might also override `operator new` and `operator delete`.

CNeoStream

Heritage

The Heritage and Ancestry of CNeoStream

Introduction

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.

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, an

imaginary number 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 directions in which stream derivations can occur. One direction 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 build a stream class, CNeoFileStream, for reading from and writing to a file.

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.

Using CNeoStream

The greatest exposure to streams that developers are likely to have is in the implementations of the `readObject` and `writeObject` methods of persistent classes. The substance of these methods consist of calls to the stream's i/o methods. While in most

cases `readObject` and `writeObject` can be implemented by simply using the stream's `readChunk` and `writeChunk` methods, we recommend that developers use the methods which aptly represent the type of data being transferred. Use `readLong` and `writeLong`, for example, when reading or writing a long integer.

The `openList` and `closeList` methods are sometimes used to establish the bounds of a list of items. While the most commonly referred to type of stream, CNeoFileStream, does not require the use of these methods when reading from and writing to disk, other types of streams which may be used in the future may require them. Developers are encouraged to use these methods in their `readObject` and `writeObject` implementations in order to insure future compatibility.

The `getStreamType` method can be used to

determine the type of a stream. The `getStreamType` method of every concrete CNeoStream subclass returns a value unique to that class of stream.

Some stream classes support the concept of a creator and type. (This type differs from the stream type returned by `getStreamType`.) Developers can get and set these values by using the `getCreator`, `getType`, `setCreator` and `setType` methods.

Subclassing CNeoStream

CNeoStream is an abstract base class that has been designed for easy subclassing. There are only three pure virtual functions which a subclass must provide in order to be concrete. Those methods are `getStreamType`, `readChunk` and `writeChunk`. Every concrete stream subclass should override the `getStreamType` method to return a unique value which identifies that stream class. All

other i/o methods have a default implementations which ultimately call `readChunk` or `writeChunk` to perform the actual i/o. Stream classes that use the NeoTag values passed to i/o methods and those which can not distill i/o operations down to simple `readChunk` and `writeChunk` calls should override the complete set of i/o methods.

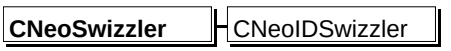
Stream classes that use the length and mark properties of the stream will need to override the methods `getLength`, `setLength`, `getMark` and `setMark`.

Some CNeoStream subclasses use data caching algorithms to optimize the performance of their i/o methods. These subclasses should also override `flush` to flush this data from the cache.

The table given below defines which methods should be overridden and when.

CNeoSwizzler

Heritage



The Heritage and Ancestry of CNeoSwizzler

Introduction

One of the primary design goals in the development of NeoAccess has been to keep complexity in the developer interface at the minimum. NeoAccess includes a class called CNeoSwizzler that takes this goal to the limit. Swizzlers are smart pointers. They're actually objects that look and act like pointers. But they're more powerful than standard C pointers in that they can be used to refer to permanent objects that might not even be in memory.

Using CNeoSwizzler

CNeoSwizzler is an abstract base class upon which concrete derivatives are based. For the

most part, swizzlers can be used like any other C pointer. They can be assigned to, compared with and indirected through. But there are additional methods that are used to manipulate the special properties that swizzlers provide. The `getClassID` method can be used to obtain the class ID of the object that the swizzler refers to. The `getKey` method can be used to obtain a select key used to locate the object in the database. The `getObject` method will return a pointer to the object. And `purge` will delete the swizzler's reference to the object in memory.

But the methods of a swizzler that are most often used are its operator overloads. The 'pointing to' operator ('->') can be used to indirect through to the object that the swizzler refers to. The 'not' operator ('!') is overridden to determine whether or not the swizzler refers to a valid object.

CNeoThread

Heritage

The Heritage and Ancestry of CNeoThread

Introduction

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 when the operation is scheduled and it finally completes. The parameters passed to the scheduling function includes a pointer to an **i/o completion routine**, a function which will be 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 was 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 and 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 method 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` methods 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.

MacApp 3.1 Support

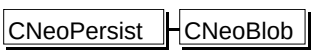
Introduction

NeoAccess is an object framework that provides object persistence capabilities. This support is provided as a set of C++ classes that naturally extend standard application frameworks on several different platforms using a number of different development environments. NeoAccess portability is implemented through the use of environment-specific classes and by using compile-time symbols and typedefs.

CNeoBlob Inheritance Tree Using OWL

Consider the above diagram which shows a simplified inheritance tree for the class CNeoBlob when built using Borland's ObjectWindows Library (OWL) application framework. The root of all streamable classes in OWL is TStreamableBase, so naturally that is

CNeoBlob's root class as well. Because blobs are persistent objects, CNeoBlob also inherits from CNeoPersist. The immediate parent of CNeoBlob is CNeoPersistOWL. This class supplements CNeoPersist, which is framework-neutral, with persistence support that is unique to OWL. For example, all OWL classes should support the `isA` and `isEqual` methods.



CNeoBlob Inheritance Tree Using PowerPlant

Now consider the diagram immediately above which again depicts the inheritance tree of CNeoBlob, but this time under MetroWerks' application framework, PowerPlant. PowerPlant has a 'mixin' architecture which eliminates the need for a single root class from which all other classes are derived. There is no additional support necessary in order to mix object persistence into a class. In this environment CNeoBlob's immediate parent is CNeoPersist.

The environment-specific support for PowerPlant-based applications is different than that provided for OWL applications.

The type CNeoPersistNative is defined in all environments to refer to the environment-specific subclass from which all persistent objects are based.

Isolating environmental dependencies in subclasses greatly improves portability. It also reduces the complexity and increases the clarity of environment-neutral classes. This allows a larger set of development efforts to access the power of NeoAccess regardless of the application framework being used. It also means that developers using NeoAccess can expect NeoAccess classes to include the same set of features that other classes native to the development environment support.

MacApp-Specific Symbols and Classes

The standard release of NeoAccess includes support for Apple's MacApp application framework. This support is provided through the use of environment-specific classes and compile-time symbols and typedefs. The include file NeoMacApp.h contains most of the MacApp-specific symbols typedefs. MacApp-specific subclasses include CNeoAppMA, CNeoDocMA, CNeoFileHandlerMA, CNeoDatabaseMA, CNeoIteratorMA and CNeoPersistMA.

CNeoAppMA

Heritage

The Heritage and Ancestry of CNeoAppMA

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, MacApp being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoApp is an environment-neutral class. The immediate base class of CNeoApp varies depending on the application framework. The base class of CNeoApp when built using MacApp is TApplication. The environment-specific header file for the framework will define the typedef

`CNeoAppbase` to be a synonym for this base class.

The MacApp-specific header file defines the typedef `CNeoAppNative` to be a synonym for CNeoAppMA. Customer-specific application classes should have a base class of `CNeoAppNative`.

Introduction

NeoAccess occasionally needs to interface with environment-specific areas of the application framework. The most standard application frameworks usually delegate primary responsibility for process state manipulation, the scheduling and dispatching of idle time and the handling of low memory situations to the application object. The abstract base class CNeoApp is designed to be a subclass of the application framework's application class. CNeoApp includes a number of pure virtual functions that are further overridden by

environment-specific subclasses of CNeoApp. These virtual functions provide an environment-neutral interface with which to access these functional areas.

Using CNeoAppMA

There are situations that arise in NeoAccess where some tasks need to be deferred until some later point in time. These deferred tasks are called chores. All environment-specific derivatives of CNeoApp include functions for tracking chores and seeing that they are performed during idle time. Chores are scheduled, dispatched and dequeued by calling

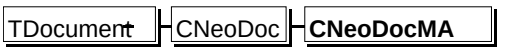
`addChore`, `doChores` and `removeChore`, respectively. The application's `purgeCache` method should be called in low memory situations to free memory used by the NeoAccess object cache.

All environment-specific application classes should include three static functions, `HideWindow`, `MoveWindow` and `ShowWindow`, which are used to hide, move and show application windows in that environment.

CNeoAppMA also overrides the `DoIdle` and `DoMakeFile` methods to execute chores at idle time and to instantiate and initialize a MacApp-specific derivative of CNeoDatabase.

CNeoDocMA

Heritage



The Heritage and Ancestry of CNeoDocMA

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, MacApp being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoDoc is an environment-neutral document class. The immediate base class of CNeoDoc varies depending on the application framework. The base class of CNeoDoc when built using MacApp is TDocument. The environment-specific header file for the framework will define the typedef

`CNeoDocBase` to be a synonym for this base class.

The MacApp-specific header file defines the typedef `CNeoDocNative` to be a synonym for CNeoDocMA. Application-specific document classes should have a base class of `CNeoDocNative`.

Introduction

The abstract base class CNeoDoc is designed to be a subclass of the native application framework's document class (which under MacApp is TDocument). CNeoDoc includes pure virtual functions that are further overridden by CNeoDocMA. These virtual functions provide an environment-neutral interface for accessing the list of currently open documents.

Using CNeoDocMA

MacApp document objects refer to their associated file objects through a mechanism

called a file handler. File handler objects abstract away the specifics of how files are manipulated. The MacApp support within NeoAccess includes a special type of file handler for manipulating NeoAccess databases. The `DoMakeFileHandler` method of CNeoDocMA creates and initializes a CNeoFileHandler object.

Many operations which are quite difficult to implement using the standard MacApp document class are trivial using NeoAccess. The `DoNeedDiskSpace`, `DoRead` and `DoWrite` methods are implemented in a few simple lines of code. `DoNeedDiskSpace` simply calls the database object's `getLength` method to determine the size of the database. NeoAccess-based applications bring objects into memory as they are needed rather than all at once. As such, the `DoRead` simply calls `NeoInherited` to read the application's print record. CNeoDocMA's `DoWrite` method calls the database object's `commit` method to write out to disk all permanent objects that are dirty.

Whenever a new document is created it is assigned a unique four-byte identity and threaded into a list of currently open documents. The document is removed from the list when it is deleted. The purpose of this list is to allow the

set of open documents to be accessed via various selection criteria. The static member function `FindByFSSpec` is used to locate a document using a file specification.

CNeoDatabaseMA

Heritage

The Heritage and Ancestry of CNeoDatabaseMA

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, MacApp being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoDatabase is an environment-neutral database class. The immediate base class of CNeoDatabase varies depending on the application framework. The base class of CNeoDatabase when built using MacApp is TFile. The environment-specific

header file for the framework will define the typedef `CNeoDatabaseBase` to be a synonym for this base class.

The MacApp-specific header file defines the typedef `CNeoDatabaseNative` to be a synonym for CNeoDatabaseMA. Application-specific database classes should have a base class of `CNeoDatabaseNative`.

Introduction

See the discussion of the class CNeoDatabase for information on what NeoAccess database objects are and how they are used and subclassed.

CNeoFileHandler

Heritage

The Heritage and Ancestry of CNeoFileHandler

Introduction

Each file–based document in MacApp is associated with an instance of TFileHandler, which manages disk file access for the document. The TFileHandler object sends messages to instances of TFile to do the actual work of file I/O. However, the way in which NeoAccess databases are accessed differs greatly from the way they are traditionally accessed in MacApp-based applications. When opening a file, MacApp applications in the past would open a file, read the entire contents into memory (inhale) and then close the file. When saving changes to the file, applications would rewrite the entire contents of the file back out to

disk (exhale).

NeoAccess-based applications should use the file handler subclass CNeoFileHandler. This derivative file handler, together with CNeoAppMA and CNeoDocMA, will provide much of the support necessary to effectively use NeoAccess databases in a MacApp environment.

CNeoIteratorMA

Heritage

The Heritage and Ancestry of CNeoIteratorMA

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, MacApp being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoIterator is an environment-neutral iterator class. The immediate base class of CNeoIterator varies depending on the application framework. The base class of CNeoIterator when built using MacApp is CIterator. The environment-specific header file

for the framework will define the typedef `CNeoIteratorBase` to be a synonym for this base class.

The MacApp-specific header file defines the typedef `CNeoIteratorNative` to be a synonym for CNeoIteratorMA. Application-specific iterator classes should have a base class of `CNeoIteratorNative`.

Introduction

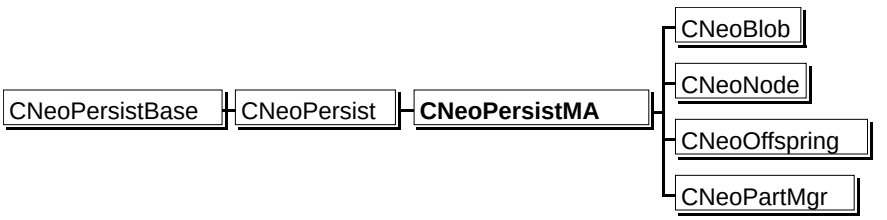
See the discussion of the CNeoIterator class for information on NeoAccess iterators.

Using CNeoIteratorMA

NeoAccess's iterator classes are based on CIterator when built under MacApp. Subclasses of CIterator are expected to override the `More` and `Reset` pure virtual functions of CIterator. The CNeoIteratorMA implementation of `More` simply calls `more`, and `Reset` simply calls `reset`.

CNeoPersistMA

Heritage



The Heritage and Ancestry of CNeoPersistMA

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, MacApp being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoPersist is an environment-neutral persistence class. The immediate base class of CNeoPersist varies depending on the application framework. Its base class when built

using MacApp is TObject. The environment-specific header file for the framework will define the typedef `CNeoPersistBase` to be a synonym for this base class.

The MacApp-specific header file defines the typedef `CNeoPersistNative` to be a synonym for CNeoPersistMA. Application-specific persistence classes should have a base class of `CNeoPersistNative`.

Introduction

See the discussion of the CNeoPersist class for information on NeoAccess persistence and how it is provided by CNeoPersist.

Using CNeoPersistMA

The only properties that NeoAccess's persistent classes inherit from MacApp's TObject base class the ability to clone. Use the `Clone` method to clone a persistent object and all the objects that it refers to. Use `ShallowClone`

to simply clone the persistent object itself.

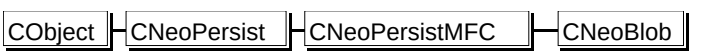
Subclassing CNeoPersistMA

All subclasses of CNeoPersistMA should also override the `Clone` and `ShallowClone` methods if they include data members that need special attention during the cloning process.

MFC 2.5 Support

Introduction

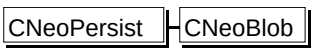
NeoAccess is an object framework that provides object persistence capabilities. This support is provided as a set of C++ classes that naturally extend standard application frameworks on several different platforms using a number of different development environments. NeoAccess portability is implemented through the use of environment-specific classes and by using compile-time symbols and typedefs.



CNeoBlob Inheritance Tree Using MFC

Consider the above diagram which shows a simplified inheritance tree for the class CNeoBlob when built using Microsoft Foundation Classes (MFC) application framework. The root of all classes in MFC is CObject, so naturally that is CNeoBlob's root class as well. Because blobs are persistent

objects, CNeoBlob also inherits from CNeoPersist. The immediate parent of CNeoBlob is CNeoPersistMFC. This class supplements CNeoPersist, which is framework-neutral, with persistence support that is unique to MFC. For example, all persistent MFC classes should support the `Serialize` and `Dump` methods.



CNeoBlob Inheritance Tree Using PowerPlant

Now consider the diagram immediately above which again depicts the inheritance tree of CNeoBlob, but this time under Metrowerks application framework, PowerPlant. PowerPlant has a ‘mixin’ architecture which eliminates the need for a single root class from which all other classes are derived. There is no additional support necessary in order to mix object persistence into a class. In this environment CNeoBlob’s immediate parent is CNeoPersist. The environment-specific support for

PowerPlant-based applications is different than that provided for MFC applications. The type CNeoPersistNative is defined in all environments to refer to the environment-specific subclass from which all persistent objects are based. Isolating environmental dependencies in subclasses greatly improves portability. It also reduces the complexity and increases the clarity of environment-neutral classes. This allows a larger set of development efforts to access the power of NeoAccess regardless of the application framework being used. It also means that developers using NeoAccess can expect NeoAccess classes to include the same set of features that other classes native to the development environment support.

Changes From Previous Versions

As of NeoAccess version 3.0, there is no need for MFC-specific subclass of CNeoDatabase. Developers use the CNeoDatabase class directly in their applications. However, for compatibility with earlier versions and to enable easier cross-platform development, the compile-time symbols `CNeoDatabaseMFC` and `CNeoDatabaseNative` are defined as synonyms of CNeoDatabase. (Correspondingly, `CNeoDatabaseMFCH` and `CNeoDatabaseNativeH` are synonymous with CNeoDatabaseH.)

MFC Serialization Support

The compile time symbol `qNeoSerialMFC` is defined whether CNeoPersistMFC and its subclasses include support for MFC's serialization mechanism. This mechanism is enabled when `qNeoSerialMFC` is defined.

For more information about MFC's object serialization support, refer to the MFC CObject and CArchive classes.

MFC-Specific Symbols and Classes

NeoAccess includes support for development using the Microsoft Foundation Classes application framework. This support is provided through the use of environment-specific classes and compile-time symbols and typedefs. The include files NeoIBMPC.h, NWin.h, NVC.h, and NMFC.h contain IBM-PC -specific, Windows-specific, MS Visual C++ specific, and MFC-specific symbols and typedefs, respectively. MFC-specific subclasses include CNeoAppMFC, CNeoDocMFC, CNeoPersistMFC and CNeoStreamMFC.

MFC-Specific Debugging & Exception Handling

MFC provides many debugging helper macros

(like ASSERT, TRACE and ASSERT_VALID). All of these macros can be used with NeoAccess and with any classes derived from CNeoPersistMFC. MFC also provides debug versions of the new and delete operators. These overrides can be used to detect memory leaks in your application. NeoAccess supports the use of those debugging constructs. Standard exception handling in MFC is implemented through the use of the TRY, CATCH, and END_CATCH macros. You can still use TRY/CATCH blocks to perform exception handling in NeoAccess; however, NeoAccess also provides an alternative set of exception-handling blocks such as NEOTRY/NEOCATCH/NEOENDTRY, and NEOTRYTO/NEOCLEANUP/NEOENDTRYTO. (See the "Exception Handling" discussion in the Preliminaries section of this document for more information on NeoAccess's support for exceptions.)

CNeoAppMFC

Heritage

The Heritage and Ancestry of CNeoAppMFC

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, Microsoft Foundation Classes, or MFC, being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoApp is an environment-neutral class. The immediate base class of CNeoApp varies depending on the application framework. The base class of CNeoApp when built using MFC is CWinApp. The environment-specific header file for the

framework will define the typedef `CNeoAppBase` to be a synonym for this base class.

The MFC-specific header file defines the typedef `CNeoAppNative` to be a synonym for CNeoAppMFC. Customer-specific application classes should have a base class of `CNeoAppNative`.

Introduction

NeoAccess occasionally needs to interface with environment-specific areas of the application framework. The architecture of most standard application frameworks usually delegates primary responsibility for process state manipulation, the scheduling and dispatching of idle time and the handling of low memory situations to the application object. The abstract base class CNeoApp is designed to be a subclass of the application framework's application class. CNeoApp includes a number

of pure virtual functions that are further overridden by environment-specific subclasses of CNeoApp. These virtual functions provide an environment-neutral interface with which to access these functional areas.

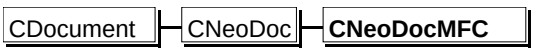
Using CNeoAppMFC

There are situations that arise in NeoAccess where some tasks need to be deferred until some later point in time. These deferred tasks are called chores. All environment-specific derivatives of CNeoApp include functions for tracking chores and seeing that they are

performed during idle time. Chores are scheduled, dispatched and dequeued by calling `addChore`, `doChores` and `removeChore`, respectively. The application's `purgeCache` method should be called in low memory situations to free memory used by the NeoAccess object cache. CNeoAppMFC also overrides the `OnIdle` method to ensure that chores are performed at idle time.

CNeoDocMFC

Heritage



The Heritage and Ancestry of CNeoDocMFC

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, MFC being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment. The base class CNeoDoc is an environment-neutral document class. The immediate base class of CNeoDoc varies depending on the application framework. The base class of CNeoDoc when built using MFC is CDocument. The MFC-specific header file defines the typedef `CNeoDocNative` to be a

synonym for CNeoDocMFC. Application-specific document classes should have a base class of `CNeoDocNative`.

Introduction

CNeoDoc includes pure virtual functions that are further overridden by CNeoDocMFC. These virtual functions provide an environment-neutral interface for accessing the NeoAccess database object associated with the document. CNeoDocMFC also provides overrides for several virtual methods of CDocument.

Using CNeoDocMFC

CNeoDocMFC is the environment-specific NeoAccess document class for applications built using the MFC framework. The methods that it overrides are necessary in order to provide native document support for NeoAccess-based applications built using MFC.

The methods `getDatabase` and

`setDatabase` are used to refer to the database object of the document. They are included in order to provide NeoAccess classes with a environment-neutral interface for referring to the database of a document.

CNeoPersistMFC

Heritage

The Heritage and Ancestry of CNeoPersistMFC

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, Microsoft Foundation Classes, or MFC, being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoPersist is an environment-neutral persistence class. The immediate base

class of CNeoPersist varies depending on the application framework. Its base class when built using MFC is the root class, CObject. The environment-specific header file for the framework will define the typedef `CNeoPersistbase` to be a synonym for this base class. The MFC-specific header file defines the typedef `CNeoPersistNative` to be a synonym for CNeoPersistMFC. Application-specific persistence classes should have a base class of `CNeoPersistNative`.

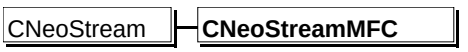
Introduction

The only methods that CNeoPersistMFC overrides are the virtual methods of CObject. See the discussion of the CNeoPersist class for information on NeoAccess persistence and how it is provided by CNeoPersist. Developers can use Serialize method for MFC-style serialization of the object. Other methods of CNeoPersistMFC are used to provide full

support for MFC debugging mechanisms. (See the Diagnostics section in the MFC Class Library User's Guide for more information about the debugging features of MFC.)

CNeoStreamMFC

Heritage



The Heritage and Ancestry of CNeoStreamMFC

Introduction

Most application frameworks, the MFC included, provide a streams mechanism to serialize the state of objects and other data. Classes of objects that can be serialized to a stream usually need to override a set of methods, much like CNeoPersist's `readObject` and `writeObject` methods, to preserve and restore the values of persistent data members.

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.

NeoAccess's MFC-specific support includes an environment-specific stream class which eliminates the need for subclasses of CNeoPersistMFC to override the `Serialize` method. The stream, CNeoStreamMFC, maps operations that would normally call an object's `Serialize` method to instead call its `readObject` and `writeObject` methods.

Using CNeoStreamMFC

There is really nothing that an application developer needs to do in order to take advantage of NeoAccess's stream mapping support in the MFC. A call to a persistent object's `Serialize` method will invoke the CNeoPersistMFC implementations. The implementation of these methods will cause the MFC-based stream (e.g. CArchive) to be mapped into a CNeoStreamMFC. After that the persistent object's `readObject` or `writeObject` method is called with a

CNeoStreamMFC as the first argument.

Please note, however, that NeoAccess itself is not using CNeoStreamMFC internally and developers need not use it to store data in the standard NeoAccess database file. This class stream class is provided to facilitate inputting/outputting of CNeoPersistMFC's subclasses to the streams included with MFC (i.e. subclasses of CArchive).

OWL 2.0 Support

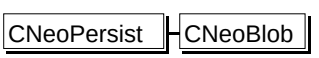
Introduction

NeoAccess is an object framework that provides object persistence and database capabilities. This support is provided as a set of C++ classes that naturally extend standard application frameworks on several different platforms using a number of different development environments. NeoAccess portability is implemented through the use of environment-specific classes and by using compile-time symbols and typedefs.

CNeoBlob Inheritance Tree Using OWL

Consider the above diagram which shows a simplified inheritance tree for the class CNeoBlob when built using Borland's ObjectWindows Library (OWL) application

framework. The root of all streamable classes in OWL is TStreamableBase, so naturally that is CNeoBlob's root class as well. Because blobs are persistent objects, CNeoBlob also inherits from CNeoPersist. The immediate parent of CNeoBlob is CNeoPersistOWL. This class supplements CNeoPersist, which is framework-neutral, with persistence support that is unique to OWL. For example, all OWL streamable classes should support the `Streamer::Write` and `Streamer::Read` methods. (Please see section 'ObjectWindows-Specific Symbols and Classes' below for additional information about the NeoAccess inheritance tree under OWL and about the symbol `qNeoOWLPersist.`)



CNeoBlob Inheritance Tree Using PowerPlant

Now consider the diagram immediately above

which again depicts the inheritance tree of CNeoBlob, but this time under Metrowerks' PowerPlant application framework. PowerPlant has a 'mixin' architecture which eliminates the need for a single root class from which all other classes are derived. There is no additional support necessary in order to mix object persistence into a class. In this environment CNeoBlob's immediate parent is CNeoPersist. The environment-specific support for PowerPlant-based applications is different than that provided for OWL applications.

The type CNeoPersistNative is defined in all environments to refer to the environment-specific subclass from which all persistent objects are based.

Isolating environmental dependencies in subclasses greatly improves portability. It also reduces the complexity and increases the clarity of environment-neutral classes. This allows a larger set of development efforts to access the power of NeoAccess regardless of the application framework being used. It also means that developers using NeoAccess can expect NeoAccess classes to include the same set of features that other classes native to the development environment support.

ObjectWindows-Specific Symbols and Classes

NeoAccess includes support for version 2.0 of Borland's ObjectWindows application framework. This support is provided through the use of environment-specific classes and compile-time symbols and typedefs. The include files NeoIBMPC.h, NWin.h, NBorland.h, and NObjWin.h contain IBM-PC

-specific, Windows-specific, Borland C++ specific, and OWL-specific symbols and typedefs, respectively. ObjectWindows-specific subclasses include CNeoAppOWL, CNeoDocOWL, CNeoPersistOWL and CNeoStreamOWL.

Object Windows 2.0 moved away from the "single root" approach used in earlier OWL releases. However, streamable classes still needs to be derived from TStreamableBase in order to support OWL-style streaming/persistence. By default, OWL-style persistence is disabled by NeoAccess. As such, by default CNeoPersist has no superclass. If you would prefer to retain OWL streaming capabilities for CNeoPersistOWL then you should define the compile time symbol `qNeoOWLPersist` at the top of the file `\NEO\INCLUDES\IBMPC\OWL\NOBJWIN.H`. In this case CNeoPersist inherits from TStreamableBase and CNeoPersistOWL

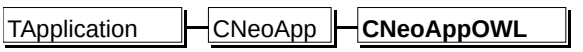
implements the `Streamer::Write` and `Streamer::Read` methods.

Changes From Previous Versions

As of NeoAccess version 3.0, there is no need for OWL-specific subclass of CNeoDatabase. Developers use the CNeoDatabase class directly in their applications. However, for compatibility with earlier versions and to enable easier cross-platform development, the compile-time symbols `CNeoDatabaseOWL` and `CNeoDatabaseNative` are defined as synonyms of CNeoDatabase. (Correspondingly, `CNeoDatabaseOWLH` and `CNeoDatabaseNativeH` are synonymous with CNeoDatabaseH.)

CNeoAppOWL

Heritage



The Heritage and Ancestry of CNeoAppOWL

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, Borland's ObjectWindows, or OWL, being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoApp is an environment-neutral class. The immediate base class of CNeoApp varies depending on the application framework. The base class of CNeoApp when built using OWL is TApplication. The environment-specific header file for the

framework will define the typedef `CNeoAppBase` to be a synonym for this base class.

The OWL-specific header file defines the typedef `CNeoAppNative` to be a synonym for CNeoAppOWL. Your application classes should have a base class of `CNeoAppNative`.

Introduction

NeoAccess occasionally needs to interface with environment-specific areas of the application framework. The architecture of most standard application frameworks usually delegates primary responsibility for process state manipulation, the scheduling and dispatching of idle time and the handling of low memory situations to the application object. The abstract base class CNeoApp is designed to be a subclass of the application framework's application class. CNeoApp includes a number of pure virtual functions that are further

overridden by environment-specific subclasses of CNeoApp. These virtual functions provide an environment-neutral interface with which to access these functional areas.

Using CNeoAppOWL

There are situations that arise in NeoAccess where some tasks need to be deferred until some later point in time. These deferred tasks are called **chores**. All environment-specific derivatives of CNeoApp include functions for tracking chores and seeing that they are

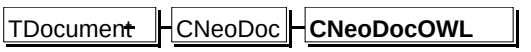
performed during idle time. Chores are scheduled, dispatched and dequeued by calling `addChore`, `doChores` and `removeChore`, respectively.

The application's `purgeCache` method should be called in low memory situations to free memory used by the NeoAccess object cache.

CNeoAppOWL also overrides the `IdleAction` method ensure that chores are performed at idle time.

CNeoDocOWL

Heritage



The Heritage and Ancestry of CNeoDocOWL

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, PowerPlant being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoDoc is an environment-neutral document class. OWL 2.0 supports a powerful document/view model and CNeoDoc is derived from OWL's standard TDocument class.

The OWL-specific header file defines the

typedef `CNeoDocNative` to be a synonym for CNeoDocOWL. Application-specific document classes should have a base class of `CNeoDocNative`.

Introduction

CNeoDoc includes pure virtual functions that are further overridden by CNeoDocOWL. These virtual functions provide an environment-neutral interface for accessing the NeoAccess database object associated with the document. CNeoDocOWL also overrides some virtual methods of TDocument to provide NeoAccess specific functionality in response to document/view events.

Using CNeoDocOWL

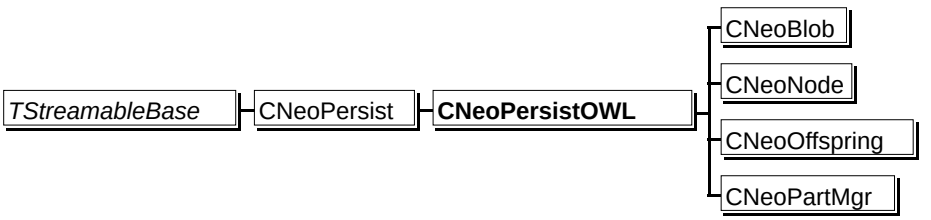
CNeoDocOWL is the environment-specific NeoAccess document class for applications built using the OWL framework. The methods that it overrides are necessary in order to

provide native document support for NeoAccess-based applications built using ObjectWindows.

The methods `getDatabase` and `setDatabase` are used to refer to the database object of the document. They are included in order to provide NeoAccess classes with a environment-neutral interface for referring to the database of a document.

CNeoPersistOWL

Heritage



The Heritage and Ancestry of CNeoPersistOWL

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, Borland's ObjectWindows, or OWL, being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoPersist is an environment-neutral persistence class. The immediate base

class of CNeoPersist varies depending on the application framework. ObjectWindows 2.0 moved away from the "single root" approach. However, one still needs to derive his classes from TStreamableBase in order to support OWL-style streaming/persistence for his classes. By default, NeoAccess does not support OWL-style persistence; thus, by default CNeoPersist does not have any superclass. However, if you would prefer to retain OWL streaming capabilities for CNeoPersistOWL then you should define `qNeoOWLPersist` symbol at the top of the file `\NEO\INCLUDES\IBMPC\OWL\NOBJWIN.H`. In this case CNeoPersist inherits from TStreamableBase and CNeoPersist OWL implements the `Streamer::Write` and `Streamer::Read` methods which allows OWL-style persistence support.

The environment-specific header file for the framework will define the typedef

`CNeoPersistBase` to be a synonym for the base class, if any.

The OWL-specific header file defines the typedef `CNeoPersistNative` to be a synonym for CNeoPersistOWL. Application-specific persistence classes should have a base class of `CNeoPersistNative`.

Introduction

See the discussion of the CNeoPersist class for information on NeoAccess persistence and how it is provided by CNeoPersist.

Using CNeoPersistOWL

The only methods that CNeoPersistOWL overrides are the streaming methods which are necessary to support OWL-style persistence.

NOTE

If the compile time symbol `qNeoOWLPersist` is not defined, then CNeoPersistOWL is defined to be the same as CNeoPersist and the symbol `CNeoPersistBase` is not defined.

CNeoStreamOWL

Heritage

CNeoStream | **CNeoStreamOWL**

The Heritage and Ancestry of CNeoStreamOWL

Introduction

Most application frameworks, the OWL included, provide a streams mechanism to serialize the state of objects and other data. Classes of objects that can be serialized to a stream usually need to override a set of methods, much like CNeoPersist's `readObject` and `writeObject` methods, to preserve and restore the values of persistent data members.

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.

NeoAccess's OWL-specific support includes an environment-specific stream class which eliminates the need for subclasses of CNeoPersistOWL to override the `Streamer::Write` and `Streamer::Read` methods. The stream, CNeoStreamOWL, maps operations that would normally call an object's `Streamer::Write` and `Streamer::Read` methods to instead call its `readObject` and `writeObject` methods.

Using CNeoStreamOWL

There is really nothing that an application developer needs to do in order to take advantage of NeoAccess's stream mapping support in the OWL. A call to a persistent object's `Streamer::Write` and `Streamer::Read` methods will invoke the CNeoPersistOWL implementations. The implementation of these methods will cause the OWL-based stream (e.g.

opstream or ipstream) to be mapped into a CNeoStreamOWL. After that the persistent object's `readObject` or `writeObject` method is called with a CNeoStreamOWL as the first argument.

Please note, however, that NeoAccess itself is not using CNeoStreamOWL internally and developers need not use it to store data in the standard NeoAccess database file. This class

stream class is provided to facilitate inputting/outputting of CNeoPersistOWL's subclasses to the streams included with OWL.

PowerPlant 1.0 Support

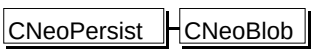
Introduction

NeoAccess is an object framework that provides object persistence capabilities. This support is provided as a set of C++ classes that naturally extend standard application frameworks on several different platforms using a number of different development environments. NeoAccess portability is implemented through the use of environment-specific classes and by using compile-time symbols and typedefs.

CNeoBlob Inheritance Tree Using OWL

Consider the above diagram which shows a simplified inheritance tree for the class CNeoBlob when built using Borland's ObjectWindows Library (OWL) application framework. The root of all streamable classes in OWL is TStreamable, so naturally that is CNeoBlob's root class as well. Because blobs

are persistent objects, CNeoBlob also inherits from CNeoPersist. The immediate parent of CNeoBlob is CNeoPersistOWL. This class supplements CNeoPersist, which is framework-neutral, with persistence support that is unique to OWL. For example, all OWL classes should support the `isA` and `isEqual` methods.



CNeoBlob Inheritance Tree Using PowerPlant

Now consider the diagram immediately above which again depicts the inheritance tree of CNeoBlob, but this time under MetroWerks application framework, PowerPlant. PowerPlant has a ‘mixin’ architecture which eliminates the need for a single root class from which all other classes are derived. There is no additional support necessary in order to mix object persistence into a class. In this environment CNeoBlob’s immediate parent is CNeoPersist. The environment-specific support for PowerPlant-based applications is different than

that provided for OWL applications.

The type CNeoPersistNative is defined in all environments to refer to the environment-specific subclass from which all persistent objects are based.

Isolating environmental dependencies in subclasses greatly improves portability. It also reduces the complexity and increases the clarity of environment-neutral classes. This allows a larger set of development efforts to access the power of NeoAccess regardless of the application framework being used. It also means that developers using NeoAccess can expect NeoAccess classes to include the same set of features that other classes native to the development environment support.

PowerPlant-Specific Symbols and Classes

The standard release of NeoAccess includes support for MetroWerks PowerPlant application framework. This support is provided through the use of environment-specific classes and compile-time symbols and typedefs. The include file NeoPowerPlant.h contains most of the PowerPlant-specific symbols typedefs. PowerPlant-specific subclasses include CNeoAppPP, CNeoDocPP and CNeoDatabasePP.

CNeoAppPP

Heritage

The Heritage and Ancestry of CNeoAppPP

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, PowerPlant being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoApp is an environment-neutral class. The immediate base class of CNeoApp varies depending on the application framework. The base class of CNeoApp when built using the PowerPlant is LDocApplication. The environment-specific header file for the framework will define the typedef

`CNeoAppbase` to be a synonym for this base class.

The PowerPlant-specific header file defines the typedef `CNeoAppNative` to be a synonym for CNeoAppPP. Customer-specific application classes should have a base class of `CNeoAppNative`.

Introduction

NeoPersist occasionally needs to interface with environment-specific areas of the application framework. The architecture of most standard application frameworks usually delegates primary responsibility for process state manipulation and the scheduling and dispatching of idle time. The abstract base class CNeoApp is designed to be a subclass of the application framework's application class. CNeoApp includes a number of pure virtual functions that are further overridden by its environment-specific subclasses. These virtual

functions provide an environment-neutral interface with which to access these functional areas.

Using CNeoAppPP

There are situations that arise in NeoAccess where some tasks need to be deferred until some later point in time. These deferred tasks are called chores. All environment-specific derivatives of CNeoApp include functions for tracking chores and seeing that they are

performed during idle time or sooner. Chores are scheduled, dispatched and dequeued by calling `addChore`, `doChores` and `removeChore`, respectively.

All environment-specific application classes should include three static functions, `HideWindow`, `MoveWindow` and `ShowWindow`, which are used to hide, move and show application windows in that environment.

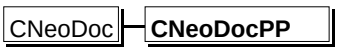
The `ChooseDocument` method presents the user with a standard dialog box with which to choose a database to open. If successful, it calls `OpenDocument` which verifies that the document hasn't already been opened.

CNeoAppPP overrides the `MakeNewDocument`, `OpenDocument`, `StartUp` and `UseIdleTime` methods which are inherited from the two PowerPlant base classes, LDocApplication and LBroadcaster, to

instantiate and open PowerPlant-specific derivatives of CNeoDoc.

CNeoDocPP

Heritage



The Heritage and Ancestry of CNeoDocPP

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, PowerPlant being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoDoc is an environment-neutral document class. The base class of CNeoDoc when built using PowerPlant is LSingleDoc. The PowerPlant-specific header file, NeoPowerPlant.h, defines the typedef `CNeoDocNative` to be a synonym for CNeoDocPP. Application-specific document

classes should have a base class of `CNeoDocNative`.

The PowerPlant-specific header file defines the typedef `CNeoDocNative` to be a synonym for CNeoDocPP. Application-specific document classes should have a base class of `CNeoDocNative`.

Introduction

CNeoDoc includes pure virtual functions that are further overridden by CNeoDocPP. These virtual functions provide an environment-neutral interface for accessing the list of currently open documents.

Using CNeoDocPP

CNeoDocPP is the environment-specific NeoAccess document class for applications built using the PowerPlant framework. The methods that it overrides are necessary in order to provide native document support for

NeoAccess-based applications built using PowerPlant.

Whenever a new document is created it is assigned a unique four-byte identity and threaded into a list of currently open documents. The document is removed from the list when it is deleted. The purpose of this list is to allow documents to be accessed by their order in the list, by document ID or by their file specification. The static member functions

`FindByFSSpec`, `FindByWindow` and `FindTop` are used to refer to a document by file specification, window object or order, respectively. NeoAccess's base document class, CNeoDoc, provides a `FindByID` method which allows documents to be referred by identity.

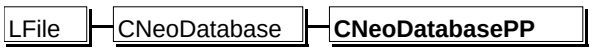
`DoSave` and `DoAESave` are overridden so that any changes to objects in the database are committed properly. The GetDescriptor method returns a Pascal string containing the title of the document. `ListenToMessage` handles low memory situations by purging the database's object cache accordingly. `openFile` is called to open an existing database on disk. `newDatabase` is called when a new empty document is created.

The methods `getDatabase` and `setDatabase` are used to refer to the database object of the document. The `isDirty`

and `setDirty` methods are used to test and set the modification state of the document, respectively. Both sets of methods are included in order to provide NeoAccess classes with a environment-neutral interface for referring to properties of a document.

CNeoDatabasePP

Heritage



The Heritage and Ancestry of CNeoDatabasePP

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, the PowerPlant being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoDatabase is an environment-neutral database class. The immediate base class of CNeoDatabase varies depending on the application framework. The base class of CNeoDatabase when built using PowerPlant is LFile. The environment-specific header file for the framework will define the

typedef `CNeoDatabaseBase` to be a synonym for this base class.

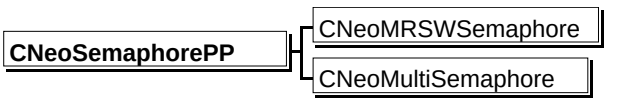
The PowerPlant-specific header file defines the typedef `CNeoDatabaseNative` to be a synonym for CNeoDatabasePP. Application-specific database classes should have a base class of `CNeoDatabaseNative`.

Introduction

See the discussion of the class CNeoDatabase for information on what NeoAccess database objects are and how they are used and subclassed.

CNeoSemaphorePP

Heritage



The Heritage and Ancestry of CNeoSemaphorePP

Introduction

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. NeoAccess's cooperative multi-threading support is enabled when built with the compile time symbol `qNeoThreads` defined.

Semaphores are used to restrict entry into a critical section of code. They control concurrent access to shared resources in multi-threaded runtime environments. NeoAccess includes

includes a set of environment-specific semaphore class which are the abstract base class for a set of special-purpose semaphore classes. CNeoSemaphorePP is the environment-specific base semaphore class used in the PowerPlant development environment.

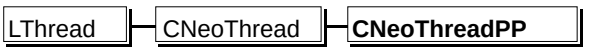
When operating in a multi-threaded environment, database objects are protected using a multiple-reader/single-writer semaphore having a leaf class of CNeoMRSWSemaphore. Each method that enters the database must first obtain a reference lock of a type appropriate to the kind of database operation being 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` methods are used to obtain and free

database lock references.

Concurrent access to individual entries in CNeoNode subclasses is controlled in multi-threaded runtime environments by a single CNeoMultiSemaphore object which is a data member of the node. A single CNeoMultiSemaphore object can control access to up to 32 individual node entries.

CNeoThreadPP

Heritage



The Heritage and Ancestry of CNeoThreadPP

Introduction

See the discussion of the CNeoThread class for information on what threads are and the level of support provided for threads in NeoAccess.

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, the PowerPlant being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoThread is an environment-neutral lightweight thread class. The immediate

base class of CNeoThread varies depending on the development environment. The base class of CNeoThread when built using the PowerPlant application framework is LThread. The environment-specific header file for the framework will define the typedef `CNeoThreadBase` to be a synonym for this base class.

The PowerPlant-specific header file defines the typedef `CNeoThreadNative` to be a synonym for CNeoThreadPP. Application-specific database classes should have a base class of `CNeoThreadNative`.

TCL 2.0 Support

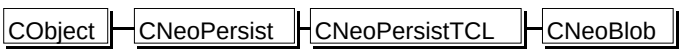
Introduction

NeoAccess is an object framework that provides object persistence capabilities. This support is provided as a set of C++ classes that naturally extend standard application frameworks on several different platforms using a number of different development environments. NeoAccess portability is implemented through the use of environment-specific classes and by using compile-time symbols and typedefs.

CNeoBlob Inheritance Tree Using OWL

Consider the above diagram which shows a simplified inheritance tree for the class CNeoBlob when built using Borland's ObjectWindows Library (OWL) application framework. The root of all classes in OWL is TObject, so naturally that is CNeoBlob's root class as well. Because blobs are persistent

objects, CNeoBlob also inherits from CNeoPersist. The immediate parent of CNeoBlob is CNeoPersistOWL. This class supplements CNeoPersist, which is framework-neutral, with persistence support that is unique to OWL. For example, all OWL classes should support the `isA` and `isEqual` methods.



CNeoBlob Inheritance Tree Using the TCL

Now consider the diagram immediately above which again depicts the inheritance tree of CNeoBlob, but this time under Symantec's application framework, the TCL. The root class from which all other classes are derived in the TCL is CObject. In this environment CNeoBlob's immediate parent is CNeoPersistTCL. The environment-specific support for TCL-based applications is different than that provided for OWL applications.

The type CNeoPersistNative is defined in all

environments to refer to the environment-specific subclass from which all persistent objects are based.

Isolating environmental dependencies in subclasses greatly improves portability. It also reduces the complexity and increases the clarity of environment-neutral classes. This allows a larger set of development efforts to access the power of NeoAccess regardless of the application framework being used. It also means that developers using NeoAccess can expect NeoAccess classes to include the same set of features that other classes native to the development environment support.

TCL-Specific Symbols and Classes

The standard release of NeoAccess includes support for Symantec's THINK Class Library application framework. This support is provided through the use of environment-specific classes and compile-time symbols and typedefs. The include file NeoTCL.h contains most of the TCL-specific symbols typedefs. TCL-specific subclasses include CNeoAppTCL, CNeoDocTCL, CNeoDatabaseTCL, CNeoPersistTCL and CNeoStreamTCL.

CNeoAppTCL

Heritage

The Heritage and Ancestry of CNeoAppTCL

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, TCL being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoApp is an environment-neutral class. The immediate base class of CNeoApp varies depending on the application framework. The base class of CNeoApp when built using the TCL is CApplication. The environment-specific header file for the framework will define the typedef

`CNeoAppbase` to be a synonym for this base class.

The TCL-specific header file defines the typedef `CNeoAppNative` to be a synonym for CNeoAppTCL. Customer-specific application classes should have a base class of `CNeoAppNative`.

Introduction

NeoAccess occasionally needs to interface with environment-specific areas of the application framework. The architecture of most standard application frameworks usually delegates primary responsibility for process state manipulation, the scheduling and dispatching of idle time and the handling of low memory situations to the application object. The abstract base class CNeoApp is designed to be a subclass of the application framework's application class. CNeoApp includes a number of pure virtual functions that are further

overridden by its environment-specific subclasses. These virtual functions provide an environment-neutral interface with which to access these functional areas.

Using CNeoAppTCL

There are situations that arise in NeoAccess where some tasks need to be deferred until some later point in time. These deferred tasks are called chores. All environment-specific derivatives of CNeoApp include functions for tracking chores and seeing that they are

performed during idle time or sooner. Chores are scheduled, dispatched and dequeued by calling `addChore`, `doChores` and `removeChore`, respectively.

The application's `purgeCache` method should be called in low memory situations to free memory used by the NeoAccess object cache.

All environment-specific application classes should include three static functions, `HideWindow`, `MoveWindow` and `ShowWindow`, which are used to hide, move and show application windows in that environment.

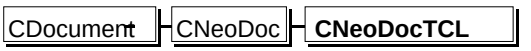
CNeoAppTCL also overrides the `createDocument`, `CreateDocument` and `OpenDocument` methods to instantiate and open TCL-specific derivatives of CNeoDoc.

The `MemoryShortage` method handles low-memory situations and the `Quit` method is

called when the application is quitting.

CNeoDocTCL

Heritage



The Heritage and Ancestry of CNeoDocTCL

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, the TCL being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoDoc is an environment-neutral document class. The immediate base class of CNeoDoc varies depending on the application framework. The base class of CNeoDoc when built using the TCL is CDocument. The environment-specific header file for the framework will define the typedef

`CNeoDocBase` to be a synonym for this base class.

The TCL-specific header file defines the typedef `CNeoDocNative` to be a synonym for CNeoDocTCL. Application-specific document classes should have a base class of `CNeoDocNative`.

Introduction

The abstract base class CNeoDoc is designed to be a subclass of the native application framework's document class (which under the TCL is CDocument). CNeoDoc includes pure virtual functions that are further overridden by CNeoDocTCL. These virtual functions provide an environment-neutral interface for accessing the list of currently open documents.

Using CNeoDocTCL

CNeoDocTCL is the environment-specific NeoAccess document class for applications

built using the THINK Class Library. The methods that it overrides are necessary in order to provide native document support for NeoAccess-based applications built using the TCL.

Whenever a new document is created it is assigned a unique four-byte identity and threaded into a list of currently open documents. The document is removed from the list when it is deleted. The purpose of this list is to allow documents to be accessed by their

order in the list, by document ID or by their file specification. The static member functions `FindByFSSpec` and `FindTop` are used to refer to a document by file specification and order, respectively. NeoAccess's base document class, CNeoDoc, provides a `FindByID` method which allows documents to be referred by identity.

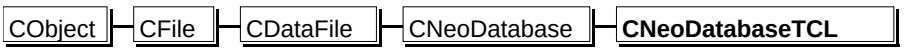
Most of the methods that are a part of CNeoDocTCL are overrides of functions inherited from CDocument, the base TCL document class. The `Activate` method is overridden to ensure that the value of the NeoAccess global variable called `gNeoDatabase` always refers to the database of the current document. `DoSave` and `DoSaveAs` are overridden so that any changes to objects in the database are committed properly. `ProviderChanged` handles low memory situations by purging the database's

object cache accordingly. `OpenFile` is called to open an existing database on disk. `NewFile` is called when a new empty document is created.

The methods `getDatabase` and `setDatabase` are used to refer to the database object of the document. They are included in order to provide NeoAccess classes with a environment-neutral interface for referring to the database of a document.

CNeoDatabaseTCL

Heritage



The Heritage and Ancestry of CNeoDatabaseTCL

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, the TCL being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoDatabase is an environment-neutral database class. The immediate base class of CNeoDatabase varies depending on the application framework. The base class of CNeoDatabase when built using the TCL is CDataFile. The environment-specific

header file for the framework will define the typedef `CNeoDatabaseBase` to be a synonym for this base class.

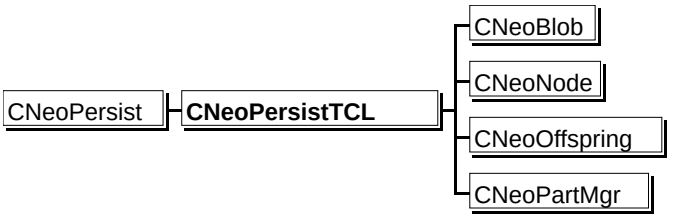
The TCL-specific header file defines the typedef `CNeoDatabaseNative` to be a synonym for CNeoDatabaseTCL. Application-specific database classes should have a base class of `CNeoDatabaseNative`.

Introduction

See the discussion of the class CNeoDatabase for information on what NeoAccess database objects are and how they are used and subclassed.

CNeoPersistTCL

Heritage



The Heritage and Ancestry of CNeoPersistTCL

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, the TCL being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoPersist is an environment-neutral persistence class. The newest releases of the TCL may use a mixin structure which eliminates the need for a single base class from

which all other classes are derived. As such, CNeoPersist doesn't have a base class when built in the TCL environment.

The TCL-specific header file defines the typedef `CNeoPersistNative` to be a synonym for CNeoPersistTCL. Application-specific persistence classes should have a base class of `CNeoPersistNative`.

Introduction

See the discussion of the CNeoPersist class for information on NeoAccess persistence and how it is provided by CNeoPersist.

Using CNeoPersistTCL

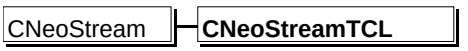
The only methods that CNeoPersistTCL overrides are `PutTo` and `GetFrom` which provide support for the "Object I/O" streams mechanism. The CNeoPersistTCL overrides of these methods encloses Object I/O streams into a NeoAccess stream which maps `PutTo` and

`GetFrom` calls to `writeObject` and `readObject` calls, respectively.

This allows CNeoPersistTCL subclasses to be serialized using either type of stream by just overriding `readObject` and `writeObject`.

CNeoStreamTCL

Heritage



The Heritage and Ancestry of CNeoStreamTCL

Introduction

Most application frameworks, the TCL included, provide a streams mechanism to serialize the state of objects and other data. Classes of objects that can be serialized to a stream usually need to override a set of methods, much like CNeoPersist's `readObject` and `writeObject` methods, to restore and preserve the values of persistent data members.

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.

NeoAccess's TCL-specific support includes an environment-specific stream class which eliminate the need for subclasses of CNeoPersist to override the `GetFrom` and `PutTo` serialization methods. The stream, CNeoStreamTCL, maps operations that would normally call an object's `GetFrom` method to instead call its `readObject` method. Operations that would call `PutTo` now call `writeObject`.

Using CNeoStreamTCL

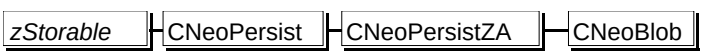
There is really nothing that an application developer needs to do in order to take advantage of NeoAccess's stream mapping support in the TCL. A call to a persistent object's `PutTo` or `GetFrom` methods will invoke the CNeoPersistTCL implementations. The implementation of these methods will cause the TCL-based stream to be mapped into a CNeoStreamTCL. The persistent object's

`readObject` or `writeObject` method will be called with a CNeoStreamTCL as the first argument.

zApp 2.1 Support

Introduction

NeoAccess is an object framework that provides object persistence capabilities. This support is provided as a set of C++ classes that naturally extend standard application frameworks on several different platforms using a number of different development environments. NeoAccess portability is implemented through the use of environment-specific classes and by using compile-time symbols and typedefs.



CNeoBlob Inheritance Tree Using zApp

Consider the above diagram which shows a simplified inheritance tree for the class CNeoBlob when built using Inmark's zApp application framework. The root of all streamable classes in zApp is zStorable, so naturally that is CNeoBlob's root class as well. Because blobs are persistent objects CNeoBlob

also inherits from CNeoPersist. The immediate parent of CNeoBlob though is CNeoPersistZA. This class supplements CNeoPersist, which is framework-neutral, with persistence support that is unique to zApp. For example, all persistent zApp classes can support the `io` method.

CNeoBlob Inheritance Tree Using OWL

Now consider the diagram immediately above which again depicts the inheritance tree of CNeoBlob, but this time under Borland's ObjectWindows (OWL) framework. OWL has a single root class for all streamable objects, which is called TStreamableBase. In this environment CNeoBlob's immediate parent is CNeoPersistOWL. The environment-specific support that is provided by this class is different than that provided by CNeoPersistZA.

Isolating environmental dependencies in

subclasses greatly improves portability. It also reduces the complexity and increases the clarity of environment-neutral classes. This allows a larger set of development efforts to access the power of NeoAccess regardless of the application framework being used. It also means that developers using NeoAccess can expect NeoAccess classes to include the same set of features that other classes native to the development environment support.

Changes from Previous Versions

As of NeoAccess version 3.0, there is no need for a zApp-specific subclass of CNeoDatabase. Developers use the CNeoDatabase class directly in their applications. However, for compatibility with earlier versions and to enable easier cross-platform development, the compile-time symbols `CNeoDatabaseZA` and `CNeoDatabaseNative` are defined as synonyms of CNeoDatabase. (Correspondingly, `CNeoDatabaseZAH` and `CNeoDatabaseNativeH` are synonymous with CNeoDatabaseH.)

Version 3.0 introduces significant expansion of CNeoDocZA interface in order to provide extended support for writing MDI applications under zApp. However, use of MDI components of zApp increases total size of the application. Thus, we require zApp DOS developers to use some DOS extender to make more than 640K of memory available to NeoAccess based applications. (See section Using NeoAccess with zApp for Dos for more information about changes in zApp support under DOS.)

ZApp-Specific Symbols and Classes

The standard release of NeoAccess includes support for Inmark's zApp application framework. This support is provided through the use of environment-specific classes and compile-time symbols and typedefs. The include files NeoIBMPC.h, NWin.h (or NDos.h for DOS developers), NVC.h (or NBorland.h for Borland C++ users), and NZapp.h contain IBM-PC -specific, operating system-specific, compiler specific, and zApp-specific symbols and typedefs, respectively. zApp-specific

subclasses include CNeoDocZA, CNeoPersistZA, CNeoStreamZA and a special chore class. These chores can be configured to be executed once and deleted or executed repeatedly at idle time. The environment-specific subclass for the CNeoApp is not provided as zApp does not provide derivable base classes for this purpose.

Using NeoAccess with zApp for DOS

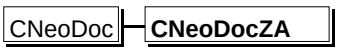
Starting from NeoAccess release 3.0 NeoAccess includes an extensive support for MDI applications with zApp. However, use of MDI components of zApp increases total size of the application. Thus, we require zApp for DOS developers to use some DOS extender to make more than 640K of memory available to NeoAccess based applications. The makefiles provided with NeoAccess assume that you are using Phar Lap 286 DOS extender. If you intend to use some other DOS extender then you have

to modify the default makefiles. Also, if you intend to use Phar Lap with Borland C++ version 4, please make sure that you have release 3.04 or later of Phar Lap product.

If you are using DOS text or DOS graphics versions of zApp you should replace the `TEMPLATE.MAK` file that is provided by Inmark with those included with NeoAccess. DOS text developers should use `\NEO\DEMO\ZAPP\DOS\TEMPLATE.DT`. DOS graphics developers should use `\NEO\DEMO\ZAPP\DOS\TEMPLATE.DG`.

CNeoDocZA

Heritage



The Heritage and Ancestry of CNeoDocZA

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, zApp being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing of these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoDoc is an environment-neutral document class. zApp does not currently include any document classes. Thus, CNeoDoc inherits from zEvH to provide zApp native event-handling support.

The zApp-specific header file defines the

typedef `CNeoDocNative` to be a synonym for CNeoDocZA. Application-specific document classes should have a base class of `CNeoDocNative`.

Introduction

CNeoDoc includes a couple pure virtual functions that are further overridden by CNeoDocZA. These virtual functions provide an environment-neutral interface for accessing the NeoAccess database object associated with a document.

Using CNeoDocZA

CNeoDocZA is the environment-specific NeoAccess document class for applications built using the zApp framework. The methods that it overrides are necessary in order to provide native document support for NeoAccess-based applications built using zApp.

Whenever a new document is created it is

assigned a unique four-byte identity and threaded into a list of currently open documents. The document is removed from the list when it is deleted. The purpose of this list is to allow documents to be accessed by their order in the list, by document ID or by their file specification. NeoAccess's base document class, CNeoDoc, provides a `FindByID` method which allows documents to be referred by identity.

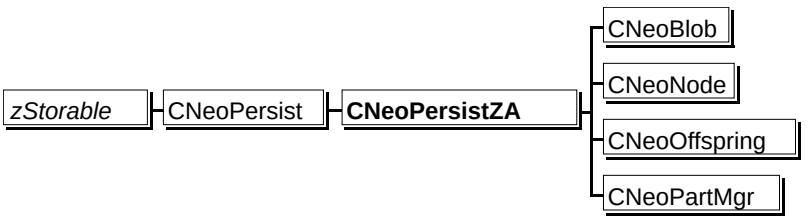
The methods `getDatabase` and `setDatabase` are used to refer to the database object of the document. They are included in order to provide NeoAccess classes with a environment-neutral interface for referring to the database of a document.

As zApp does not include any support for Document/View model, CNeoDocZA has to provide all the necessary event-handling methods. Developers should ensure that this methods are called whenever any document event (e.g., FileOpen command) occurs. (See sample applications Laughs and Neo for examples of how this can be done.) The `onNew`, `onOpen`, `onSave`, `onSaveAs`, and `onClose` methods should be called to handle FileNew, FileOpen, FileSave, FileSaveAs, and FileClose commands/events respectively. CNeoDocZA also has a pure virtual method `getWindow` which should be overridden by

application specific documents to return a pointer to zMDIFrameWin associated with this document.

CNeoPersistZA

Heritage



The Heritage and Ancestry of CNeoPersistZA

NeoAccess is a persistence framework that has been designed to extend standard application frameworks, the zApp being one of them. This cross-development support is implemented in part through the use of environment-neutral base classes. Environment-specific classes are defined by subclassing these neutral classes to provide a richer feature set that more fully integrates NeoAccess into that environment.

The base class CNeoPersist is an environment-neutral persistence class. The immediate base class of CNeoPersist varies depending on the application framework. Some frameworks

derive all of their classes from a single root class. zApp does not use that approach. However, one still needs to derive classes from zStorable in order to support zApp-style streaming (also referred to as persistence) of those classes. By default, NeoAccess does not support zApp-style persistence; thus, by default CNeoPersist does not have a superclass. The compile time symbol `qNeoZappPersist` is automatically defined when NeoAccess is compiled in the environment where the compile time symbol `Z_NP` is not defined. In this case CNeoPersist inherits from zStorable and CNeoPersistZA implements the `io` method which allows zApp-style persistence support. (See the Persistence section of zApp Programmer's Guide for more information about zApp-style persistence.) The environment-specific header file for the framework will define the typedef `CNeoPersistbase` to be a synonym for the

base class, if any.

The zApp-specific header file defines the typedef `CNeoPersistNative` to be a synonym for CNeoPersistZA. Application-specific persistence classes should have a base class of `CNeoPersistNative`.

Introduction

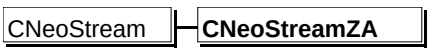
See the discussion of the CNeoPersist class for information on NeoAccess persistence and how it is provided by CNeoPersist.

Using CNeoPersistZA

The only method that NeoAccess's persistent classes override from zApp's zStorable base class is `io`. The `io` method is a virtual function that must be overridden to implement the input and output of class data.

CNeoStreamZA

Heritage



The Heritage and Ancestry of CNeoStreamZA

Introduction

Most application frameworks, the zApp included, provide a streams mechanism to serialize the state of objects and other data. Classes of objects that can be serialized to a stream usually need to override a set of methods, much like CNeoPersist's `readObject` and `writeObject` methods, to preserve and restore the values of persistent data members.

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.

NeoAccess's zApp-specific support includes an environment-specific stream class which eliminates the need for subclasses of CNeoPersistZA to override the `io` method. The stream, CNeoStreamZA, maps operations that would normally call an object's `io` method to instead call its `readObject` and `writeObject` methods.

Using CNeoStreamZA

There is really nothing that an application developer needs to do in order to take advantage of NeoAccess's stream mapping support in the zApp. A call to a persistent object's `io` method will invoke the CNeoPersistZA implementations. The implementation of these methods will cause the zApp-based stream (e.g. zArchive) to be mapped into a CNeoStreamZA. After that the persistent object's `readObject` or `writeObject` method is called with a CNeoStreamZA as the first argument.

Please note, however, that NeoAccess itself is not using CNeoStreamZA internally and developers need not use it to store data in the standard NeoAccess database file. This class stream class is provided to facilitate inputting/outputting of CNeoPersistZA's subclasses to the streams included with zApp (i.e. subclasses of zArchive).

Photographer's Assistant

Prefix

This section was originally printed under the title "NeoAccess, Object Persistence Made Simple" in the Fall, 1992 issue of Frameworks, the journal of software developers using object technology, published by the Software Frameworks Association, formally called MADA. It was written in the first person by NeoAccess's primary architect, Bob Krause, as a factual, though rather tongue-in-cheek, recounting of the development of the NeoDemo sample program, which is referred to here as "Photographer's Assistant".

Even the name raises the hair on the back of the uninitiated's necks — Object-Oriented Database Engine. Though it is new to Macintosh developers, an ODBMS named NeoAccess is generating quite a bit of

excitement because it is both powerful and easy to use. This article shows how you can use it to reduce resource requirements, provide exceptional performance, organize objects and their relationships to one another and allow you to focus on those aspects of your application that make it truly unique.

Introduction

So you have this way cool idea for a new Macintosh application, right? One that will change the way that people use and think about their Mac. You've gone on the obligatory walks on the beach and hikes in the woods to flesh out the details of how you're going to implement this beast. You want to do it right, so first off you know that the app is going to be object-oriented and that you are going to use either the THINK Class Library, PowerPlant or MacApp as an application framework.

But a problem has begun to set in. Way cool

app's need to manipulate complex sets of objects with intricate inter-relationships. That's fine, but the state of some of these objects need to be preserved across this gulf referred to as 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. You knew that you were going to need to implement Save As and Open menu items; you just didn't think it was going to be so complicated. I mean, look at Greg Dow's Art Class. That was easy, wasn't it?

The more you think about it, the more complicated the issues become. You need a file format. A file format! What has that got to do with way cool? OK, OK, focus… You can just put all instances of a particular class together in the file, then immediately follow that with the next class, and so on. That gets the data written out to the file. (You can worry about subclasses and variable length objects later, right?) But

what about the inter-relationships that exist between objects in memory? Don’t those need to be saved so that they can be recreated when the objects are read back in? You’ve got one-to-one, one-to-

many and many-to-many connections that need to be maintained. And if each object is written out one after another, then they need to be read in serially as well. When you get right down to it, you need to read in the entire file at once. Otherwise the app needs to do this dance every time it tries to reference an object — Is the object in memory? No? Then go back to the beginning of the file to read it in… If you don't do that then your application partition is going to be 2MB. Even under System 7 that's a lot of memory. And you know that virtual memory isn't an option. What a nightmare!

You finally decide that the Macintosh and object-oriented programming have not yet evolved far enough to support way cool ideas. Or if something like this can be done it has to be by a team of 40 engineers in a cold room up in Redmond. It's still a good idea. Maybe when Bedrock shows up…

NeoAccess

Each and every developer that has tried to write an app has had to face the issue of persistence. It is a difficult problem with many complications. Up until recently every developer has had to face the issue alone. But in this article I would like to show you how to use NeoAccess, an object-oriented database / persistence mechanism that developers can embed into their applications.

As a freelance consultant for many years, I was running through this nightmarish scenario over and over again. I bit the bullet a few times and developed a structured file format, came up with an organization on disk and wrote routines to read and write data. But on the next project I still ended up using only a portion of this code. It seemed that every project was different. Each had a different set of objects that need to be saved, different accessing needs and patterns and different relationships to maintain.

I finally said enough is enough and developed an object persistence mechanism, indeed, full blown database engine called NeoAccess. Because of my previous experiences, I've designed the system to be versatile. I make no assumptions about the type of data that needs to persist or the connections between them. Things are very optimized toward storing and retrieving objects, but un-formatted, variable-length blobs of data can also be mixed in. I also tried to avoid making assumptions about how objects are organized in the system. How are objects sorted? How are they indexed? How do objects relate to one another? Are they accessed serially or randomly? There are defaults, but for the most part, these questions can be answered by the application designer. The programming interface is designed to show minimum visible complexity. Years of consulting on object-oriented projects have shown that no matter how sophisticated a company's products may

be, the one thing they all share is a common struggle to minimize complexity. Of course performance is also a major concern. I wondered whether it was possible to provide versatility and performance in the same system. Now I believe that it is, and NeoAccess is proof of that. Finally, the system was designed with portability in mind. The first version of NeoAccess was developed using THINK C (which wasn't even real C++!) and the THINK Class Library. It has since gone cross-platform. It is supported in numerous development environments including MacApp 3.0 and 3.1, PowerPlant, QuickApp, ObjectWindows and zApp. And it will be moved to Bedrock if/when it shows up. You might even be seeing it under Prograph.

In this article I'm going to show how to use NeoAccess in a way cool application that Bob Ackerman and I built (using the TCL) that I've nicknamed Photographer's Assistant (its real,

though less imaginative, name is NeoDemo).

So check this out… I've got this mythical photographer friend, Toby, who likes to use the Mac in his work. He has his film developed by a service bureau that also digitally scans

his photographs and presses them onto CD-ROMs. The images are stored on the CD in a database that is readable by Photographer's Assistant.

My app displays images in a window that looks like the standard scrapbook DA on steroids. I've added a lot of extra fields that the DA doesn't have so that Toby can keep track of which camera, lens and film he used to take a shot. He has a journal that he keeps all this information, in which he enters into NeoDemo when the CD arrives. The elephant image shown below is a good example.

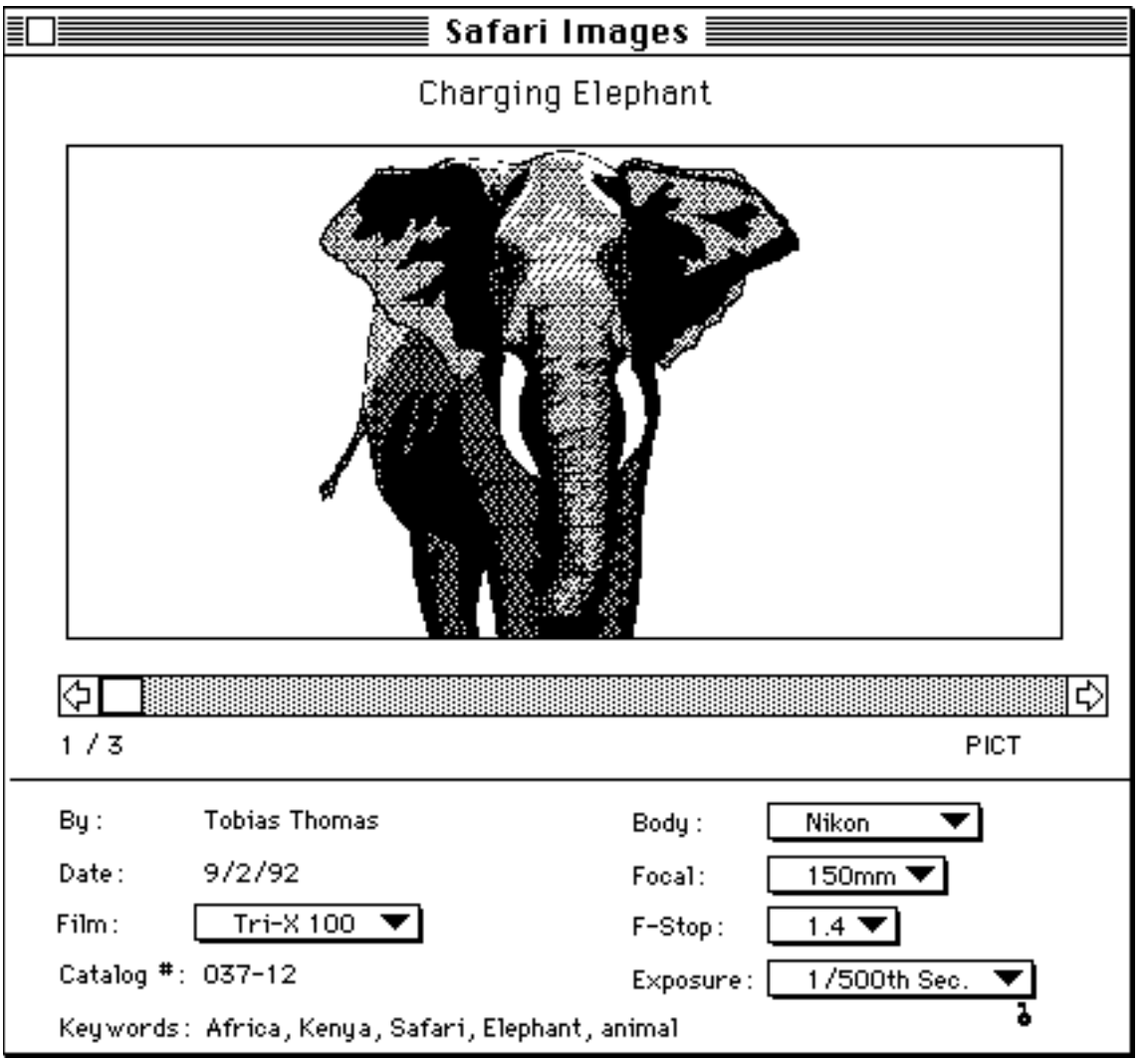


FIGURE 1 - NEODEMO WINDOW

The window is titled “Safari Images” because that is the name of the database that the images and ancillary data are in. Just above the image is the image title, “Charging Elephant” (there’s a great story to go along with this one…). Toby

can scroll through all the images in the database using the scrollbar. The indicator just below the scrollbar on the left side shows that this is the first of three images. The indicator on the right shows that its format is PICT. The fields in the lower portion of the window were set by Toby. He took the picture on September 2, 1992 with his Nikon using a 150mm lens, and so on. He has assigned this picture a catalog number and a set of keywords. The keywords are handy because NeoDemo has a nice feature that allows him to find all images that have a common keyword, like all animal images, or all those taken in Kenya.

The flag control in the bottom right-hand corner of the window opens up the search feature. When the flag is down the window expands to include a text edit in which to enter a keyword, a popup to indicate the kind of images to include in the search (PICT, TIFF, GIF or Any), and a find button to start the search.

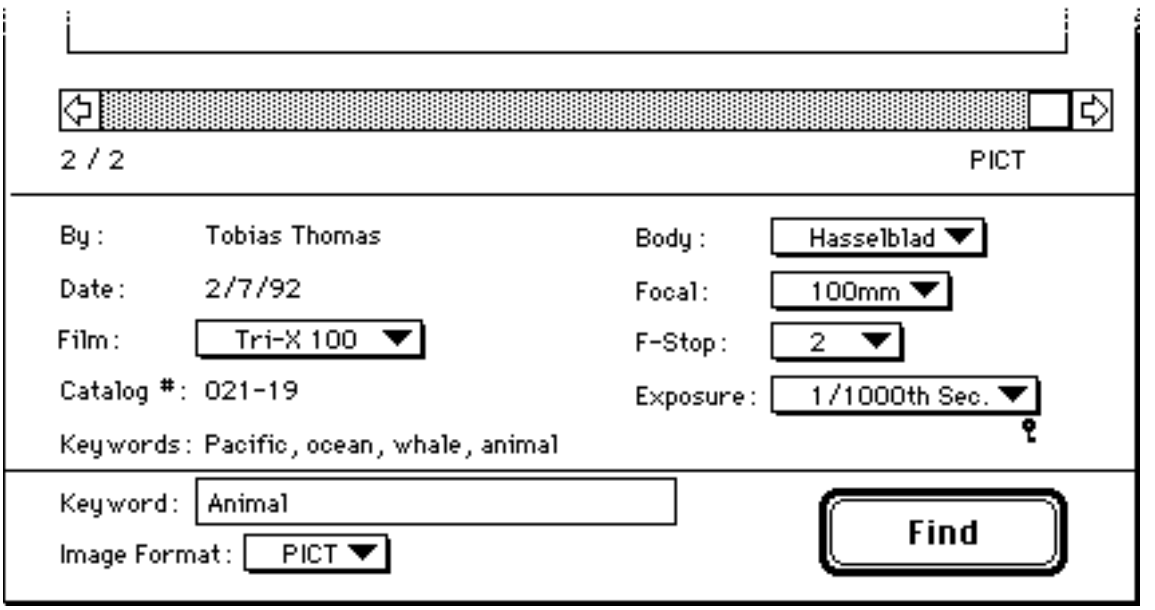


FIGURE 2 - EXTENDED NEODEMO WINDOW

Multiple windows can be open at once. That way you can copy an image from one document to another. This is how Toby creates a database to send back to the service bureau for printing

with just a single image in it. Not only is the image copied, but so is the technical info like where it was shot and how. That way all the information about the image stays with it. Of course if the scrap is pasted into some other application that doesn't know about image objects, then only the PICT data is carried over.

The NeoAccess Class Tree

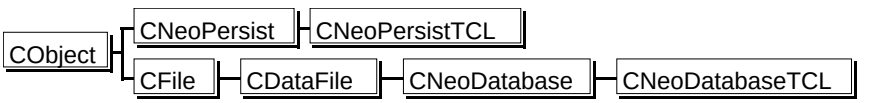


FIGURE 3 - SIMPLIFIED NEOACCESS CLASS TREE

Let's look under the covers to see how NeoAccess works. The best place to start is by looking at the definitions of some of its classes. The current version of the TCL includes an abstract class, CObject, from which all other classes descended. CObject's prospects for a continued existence are dim though. Most frameworks are heading toward a "mix-in" approach which eliminates the need to a single

common root class. The class CDataFile is a member of the core suite of classes in the THINK Class Library. CDataFile provides know-how for manipulating the data fork of Macintosh files as an extensible and randomly accessible stream of bytes.

In order to make it easy to port, NeoAccess's core classes were designed to be environment-neutral. CNeoDatabase, for example, is an environment-neutral class. That means that the interface to CNeoDatabase makes no assumptions about what environment it might be used in. However, providing a seamless integration of CNeoDatabase into the

TCL might require adding additional overrides beyond the general support provided in CNeoDatabase.

NeoAccess provides this intimate level of support in numerous environments by defining environment-specific subclasses of CNeoDatabase and other key classes. NeoAccess defines the symbol `CNeoDatabaseNative` to refer to the environment-specific database class for a particular environment (CNeoDatabaseTCL in our case).

CNeoPersist is environment-neutral as well. The symbol CNeoPersistNative refers to the environment-specific class from which all persistent subclass are derived. In environments which don't require an environment-specific subclass, CNeoPersistNative is simply equal to CNeoPersist.

1. CNeoDatabase

The class CNeoDatabase builds on the capabilities of CDataFile to provide a mechanism for storing, organizing and retrieving persistent objects.

If you consider for a moment what the responsibilities of the Macintosh File Manager are, you get a pretty good picture of the kinds of things that the CNeoDatabase class does as well. The File Manager manages allocation blocks on a volume. Most of a volume's blocks are used to store information contained in files; resources and data. The rest of the space is either unused or used to store volume catalog and extents information and desktop-management related data — the organization of folders and files, a list of used blocks, a list of blocks allocated to a particular database, the locations of folder windows on the desktop and the shape and location of icons in windows — all of this needs to be magically maintained for

the user.

The primary objectives of the File Manager is to reliably administer a volume without burdening the user (or developer) with unnecessary details. The overall efficiency with which it accomplishes these tasks is a major concern. It has to be versatile enough to satisfy the needs of its clients (the Finder and other application programs). And finally, it needs to be built in such a way that it can evolve in response to future needs.

Instead of managing allocation blocks on a volume, CNeoDatabase supervises space in a database. Most of this space is used to store the permanent data members of persistent objects. But most of the complexity of CNeoDatabase's charter involves the manipulation of its internal data structures. Just as the File Manager must track folders and their contents, CNeoDatabase must keep track of classes, subclasses and objects. Classes are related to one another to

create a class hierarchy. Objects are organized using indices. Finally, CNeoDatabase keeps track of the free space in the database.

CNeoDatabase's primary objectives are very similar to that of the File Manager. Reliability and minimizing visible complexity top the list, followed closely by performance. But it also needs to be versatile and be capable of evolving in future directions.

Many of the capabilities of CNeoDatabase, such as specifying, opening and closing a Macintosh file, getting and setting the file mark and the file length are actually provided in whole or in part by its parent classes. I won't go into too much detail on what they do or how they do it. The operations we would like to consider are those having to do with objects, classes and the management of file space.

In order to access objects contained in a database, a developer must understand how

information is organized. Figure 4 presents a schematic illustration of the reference hierarchy within NeoAccess.

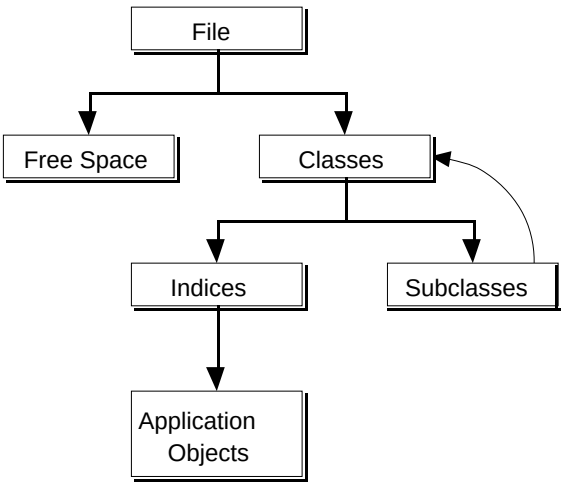


FIGURE 4 - SIMPLIFIED NEOACCESS REFERENCE TREE

Information is typically organized in a database primarily by class. C++ and most other object-oriented languages support the concept of inheritance. So does NeoAccess. When you define a subclass in C++ you indicate its parent class. When you add object of a particular class to a database NeoAccess also records what its parent class is. NeoAccess uses this information to maintain a tree that usually matches the C++ inheritance tree.

Objects are organized within a database according to class. This organization is

supported by a construct called an index. Indices keep objects in sorted order. There is a default collating sequence, but the sort order can be changed to support the needs of the application. NeoAccess even provides the ability to maintain multiple indices of a class. Though developers rarely need to know the details, indices are implemented as extended btrees. This allows NeoAccess to locate objects quickly by using binary search algorithms.

2. CNeoPersist

Application-specific objects encapsulate the intelligence of your application. They are the value that you add to the Macintosh. The *raison d'être* of your application is to provide a mechanism that allows users to manipulate these objects. The important application-specific objects are generally persistent objects. Users create something that they can come back to and work with again later. In order for this to

happen, applications need to include a mechanism that preserves the state of these objects after the application has quit, and which can be used to locate the objects again later.

Using NeoAccess, all persistent classes are based on the class CNeoPersist. The methods and data members of CNeoPersist provide the know-how and state that allows NeoAccess to manage an object's persistence. The abbreviated class definition of CNeoPersist given below shows the kind of operations one might use to manipulate an object's persistence.

```
class CNeoPersist {
public:
                    /** Instance Methods **/
                    CNeoPersist(void);
 virtual            ~CNeoPersist(void);
 virtual NeoID      getClassID(void) const;

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

```
                                const NeoTag aTag);
virtual Boolean     commit(CNeoDatabase *aDatabase,
                           const Boolean aCompletely,
                           const Boolean aCompress);

                    /** Searching Methods **/
static void *       Find(CNeoDatabase *aDatabase, const NeoID aClassID,
                         CNeoSelect *aKey, const Boolean aDeeply,
                         NeoTestFunc1 aFunc = nil, void *aParam = nil,
                         const NeoLockType aLock = kNeoDefaultLock);
static void *       FindByID(CNeoDatabase *aDatabase,
                             const NeoID aClassID, const NeoID aID,
                             const Boolean aDeeply,
                             NeoTestFunc1 aFunc = nil,
                             void *aParam = nil,
                             const NeoLockType aLock = kNeoDefaultLock);

                    /** Persistence Methods **/
virtual void        add(void);
virtual void        remove(void);
virtual void        relocate(const NeoMark aNewMark);
void                setDirty(const NeoDirty aReason = kNeoChanged);

                    /** Concurrency Methods **/
void                referTo(void);
void                unrefer(void);
void                autoReferTo(void);
void                autoUnrefer(void);

                    /** Instance Variables **/
NeoID               fID;                // Symbolic ID of this object
Boolean             fLeaf       : 1;    // True if not an inode.
Boolean             fRoot       : 1;    // Is the root of the tree?
Boolean             fBusy       : 1;    // Object being manipulated
NeoDirty            fDirty      : 2;    // Memory/file states differ
NeoMark             fMark;              // Location in database
CNeoPersist *       fParent;            // Object's parent
NeoRefCnt           fRefCnt;            // Purgeable when zero
};
```

a. Adding an Object

Let's look at some of the methods under the "Persistence Methods" category. The method `add` allocates space for the object in the database and verifies that the object has a non-zero identity. An object's identity is a 4-byte value often used to uniquely identify an object in the database. By default, objects of a particular class are sorted in ascending order by id value. So an object's identity may be important. But note that the object's `add` method is not the method an application developer would use to make an object persistent. When we look at the database class a bit later you'll see that its `addObject` method, which eventually calls the object's `add` method, should be used instead.

b. Deleting an Object

Immediately below `add` is the method `remove`. Inevitably, an application will need to delete objects from a database. But again, application developers use the database's

`removeObject` method to remove an object from the database. The object's `remove` method is called during this process to free the file space allocated for the object being removed. Note though, that an object continues to exist in memory after it has been removed from a database. 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.

c. Locating Objects

We'll come back and discuss some of the other persistence methods in a moment. But first let's look at a couple of methods under "Searching Methods". `FindByID` is used to locate an object (or set of objects) having a given identity. For example, an image object refers to a camera by its id. NeoDemo uses `FindByID` to locate the camera using the following call:

```
camera = (CNDCamera *)CNeoPersist::FindByID(gNeoDatabase, kNDCameraID,
    aCameraID, FALSE);
```

The first argument is the database to search. The global variable `gNeoDatabase` always refers to the database for the current document. The second argument is the class of object we're looking for. In this case, we're looking for a camera. The constant `kNDCameraID` is the class id used to refer to the CNDCamera class. Just as an object's identity is defined by an object id, class ids refer to classes. The third argument is the identity of the camera object we are looking for. The fourth argument indicates whether all subclasses of CNDCamera should be searched as well. This ability to search for an object according to any of its base classes is very useful. In this example there are no subclasses of CNDCamera, so this argument is false.

`FindByID` is capable of locating a camera very quickly because camera objects are indexed by identity. This means that NeoAccess can use a binary search, which is very fast. But sometimes an application needs to locate objects using a selection criterion that is not a key. For example, Toby often looks at only those images that have a particular keyword. It is fairly easy for NeoDemo to do this even though images are not indexed by keyword. That's because the database object has a very powerful method, `findObject`, which can be used to locate objects based on any criteria you can imagine. NeoAccess optimizes queries to search the database using a index if at all possible. But if the search can't be done using an index, like searching for an image by keyword, then a linear search of the database is done.

d. Changes to an Object's State

Another common occurrence in an application is when the permanent state of an object changes. For example, Toby may add a keyword to an image. That changes the object's state. This change needs to propagate back to the database. When the keyword list of an object is changed, the object's `setDirty` method is called. This marks the object as being different than its state in the database. Newly added objects are also marked dirty by NeoAccess. If dirty objects are not written back out to the database, then they revert back to their previous state when the database is closed and then reopened. The process of synchronizing the on-disk state of objects with their in-memory state is performed by the `commit` method of the database object. It is called in NeoDemo when Toby does a Save or Save As. Notice that applications don't need to keep track of which objects are dirty; NeoAccess does that for them. And unlike most persistence mechanisms which rewrite the

entire database at once, only those objects that are dirty need to be written out when saving a database.

e. Object Sharing

CNeoPersist provides a sharing property to its subclasses that greatly simplifies intra-application concurrency issues. Every persistent object has a reference count which is used to insure that an object is not deleted from memory while there are still references to it. The reference count is initialized to one when an object is instantiated. A reference is automatically added by the database's `addObject` method and by the searching methods, and is decremented by the database's `removeObject` method. When the object's `unrefer` method is call the count is decremented, but the object is only freed if the reference count is zero (meaning all references are deleted). The end result is that one component of an application doesn't need to be aware of whether an object that it refers to is referred to by another component. The object stays in memory as long as it needs to, and no longer.

As you can see, adding, removing, locating and changing objects in a NeoAccess database is fairly easy. Notice that the logistics of how objects are organized and where they are located in the database is for the most part transparent to the programmer. Also note that only those objects that are of immediate interest to the application need to be in memory. Yet simplicity and compactness does not compromise versatility. Indeed, applications such as NeoDemo are object-driven, so versatility is increased while complexity is reduced.

The NeoDemo Class Tree

Now that we have an understanding of what NeoAccess does, let's take a look at how NeoDemo uses these capabilities. The first thing to consider is the set of persistent classes that the application defines and their relationship to the CNeoPersist class and to one another.

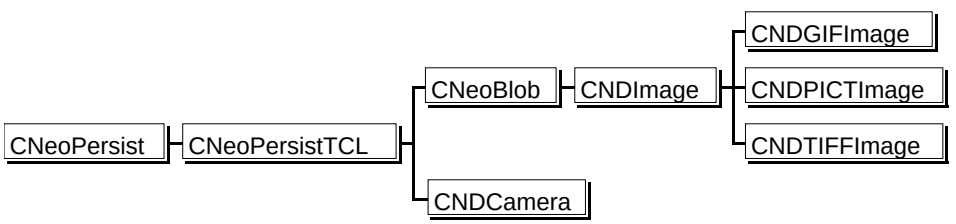


FIGURE 5 - NEODEMO PERSISTENT CLASS TREE

As you might expect, CNeoPersistTCL is the

base class of all persistent classes. One of its immediate subclasses is called CNeoBlob. Though object-oriented developers strive to make everything an object; the fact remains that not everything can be. NeoAccess provides an abstract class, CNeoBlob, in recognition of this fact. CNeoBlob is an abstract base class of persistent objects that is used to store and locate free-form, variable-length, non-object entities in a NeoAccess database.

If you include the environment-specific and environment-neutral derivatives of CNeoPersist, there are a total of eight classes in NeoDemo's database. Five of these are specific to NeoDemo. These application-specific classes are of two general types; images and cameras. There is one camera class and a base image class with three subclasses; CNDImagePICT, CNDImageTIFF and CNDImageGIF. The image classes are based on CNeoBlob. It is significant to note that the only difference

between a PICT image and a

`TIFF` image is the implementation of their draw method. The other capabilities and data members are inherited from CNDImage.

Class objects are added to the database in `CNeoDemoDoc::NewFile` around the time it is created. I would like to extend NeoDemo so that Toby can add new cameras to the database. But for now the four cameras that he typically uses are added automatically at the time the classes are added.

```
exposure[0] = -125;     /* 1/125 second */
camera = new CNDCamera("\pKodak", 1, exposure);
camera->fID = 3;
((CNeoDatabase *)itsFile)->addObject(camera);
camera->unrefer();
```

Notice how simple it is to add a camera object. The exposures array which indicates the shutter speeds supported by the camera is filled in first. Then a new camera object is created and initialized. The method `addObject` is used to add the camera to the database. We no longer need to refer to a object once it is added to the database, so we call `unrefer` to remove our reference to it.

1. Cutting, Copying and Pasting Images

NeoAccess provides a streams-based mechanism for reading and writing objects. 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 build a file stream class, CNeoFileStream, for preserving and restoring objects in a file, and a TCL-specific class, CNeoScrapStream, for reading from and writing to the clipboard.

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

The method `CNeoDemoDoc::doCutCopy` puts an image object onto the scrap using a scrap stream.

```
Boolean CNeoDemoDoc::doCutCopy(const Boolean aCut)
{
 Boolean         pass    = TRUE;
 Handle          scrap   = nil;
 CNDImage *      image;
 CNeoScrapStream stream('IMGE');     // stream object that reads images

 if (!fIndex)                        // If there is no current image
     return pass;                    // then there is nothing to do.

 image = getImage(fIndex);           // Get the current image.
 if (image) {                        // It better be there!
     updateImage(image);             // Just in case user has changed it.

     image->writeObject(&stream, kNeoAllTag);    // Write it to scrap.

     if (aCut) {                     // Is this a Cut operation?
         ((CNeoDatabase*)itsFile)->removeObject(image);
         removeImage(fIndex);        // Remove image from the database.
     }
     image->unrefer();               // Remove our reference to it.
     pass = FALSE;                   // No need to pass this command on.
 }

 return pass;
}
```

The blob portion is written to the scrap as format `PICT` data. Any application capable of using the image in `PICT` format should be able to accept this type of scrap. The other data members of CNDImage, such as photographer and exposure data, are written as format `IMGE` data. NeoDemo is the only one that understands this format. But having the image object in the scrap allows Toby to copy an image from one NeoDemo document to another without losing this data.

Images are added by pasting them into a document in the method `doPaste`. The arguments to `readObject` refer to the input stream and the amount of data to read. The value `kNeoAllTag` means to get all you can get.

```
 image->readObject(&stream, kNeoAllTag);

 pict = (PicHandle)image->getBlob();
 if (pict) {
     showBorders(TRUE);
     ((CNeoDatabase *)itsFile)->addObject(image);// add it to database
     setItems(image);                            // OK, so let's see it
     pass = FALSE;                               // don't pass it on
 }
 else {                                          // false paste
     fImageArray->DeleteItem(fIndex);            // remove from array
     fIndex--;                                   // go back to previous
 }
 image->unrefer();                               // remove reference
```

Scrap data of format `IMGE` is available only if the scrap was put there by NeoDemo. If the image came from some other app that doesn't support `IMGE` data (a.k.a., everybody else), then the image object is initialized with default data. The actual image is in the standard PICT format. The method `CNeoBlob::readObject` calls the stream's `readBlob` method to copy this free form data off of the scrap.

2. Saving a Document

After a new image is pasted into a document, Toby needs to enter the technical details about where it was shot and under what conditions. This changes the state of the image object in memory. Each time a permanent data member of an image changes, the `setDirty` method is called to mark the object as needing updating in the database. Database updating in NeoDemo occurs when the Save or Save As menu items are selected.

The method `DoSave` is where the actual database update occurs. Most of the details having to do with committing changes is taken care of by the `DoSave` method of CNeoDocTCL, NeoAccess's TCL-specific document class. The only thing left for CNeoDemoDoc to worry about is whether all the data has been copied out edit fields of the dialog box.

```
Boolean CNeoDemoDoc::DoSave(void)
{
 CNDImage *  image;

 if (itsWindow &&
     fIndex) {
     image = getImage(fIndex);
     if (image) {
         updateImage(image);
         image->unrefer();
     }
 }

 return NeoInherited::DoSave();
}
```

Objects can be added, deleted and searched for in a memory-based CNeoDatabase object before a Macintosh file has been specified and opened for it. `CNeoDocTCL::DoSave` calls `DoSaveFileAs` to specify and open the database when this is the case. Updating the database is done with a call to the database's `commit` method. The single argument to this call indicates whether the database object should attempt to reduce the amount of file space the database uses on disk. This may slow down the commitment operation but can reduce the database's size dramatically if objects have been deleted.

3. Searching for Images

The usefulness of assigning keywords to images is that NeoDemo includes a facility for selecting a subset of images having a given data format and keyword. Toby asked for this because, though a database may contain hundreds or even thousands of images at a time, he may be interested in only those of a particular type, like PICT, having a specific keyword.

```
short CNeoDemoDoc::setSelectionByKeyword(const NeoID aClassID,
                                         char *aKeyword)
{
 CNeoKeywordSelect * key;

 // get current data into image object before starting search
 updateImage(nil);

 emptyImageArray();

 key = getKeywordKey(aKeyword);
 ((CNeoDatabase*)itsFile)->findObject(aClassID, key,
                                  (aClassID == kNDImageID),
                                  CNDImage::GetImageID,
                                  (void *)fImageArray);

 fIndex = 0;
 fScrollObj->itsHorizSBar->SetMaxValue(fImageArray->numItems -1);

 if (fImageArray->numItems > 0)
     gotoImage(1, FALSE);
 else
     setItems(nil);

 return fImageArray->numItems;
}
```

`CNeoDemoDoc::setSelectionByKeyword` is called when the Find button of the extended window is pressed. The arguments indicate the class id of the images to look for (`kNDImagePICTID`, `kNDImageTIFFID`, `kNDImageGIFID` or `kNDImageID` to indicate all types) and the keyword of interest.

The `getKeywordKey` method normalizes the given keyword to lower case and creates a corresponding select key which is passed to the database's `findObject` method to locate images having the given keyword.

The search mechanisms provided by NeoAccess use a very powerful and flexible selection mechanism based on objects having a base class of CNeoSelect. CNeoKeywordSelect is a subclass of CNeoSelect. It is a type of **selection criterion** (also called **select key**) that's used to locate objects based on their identity.

The first argument to `findObject` is the class id of objects to search. This value was passed to `setSelectionByKeyword`. The second argument is the select key. The third argument being `TRUE` indicates that all subclasses of the class referred to by the first argument should also be searched. So when the base class is CNDImage, then all images classes are searched. The fourth argument is a pointer to the function to be applied to each object in turn. Finally, the last argument is a pointer to data shared between the caller of `findObject` and the function. This can be anything, but in this case it is an array pointer into which the object id of each object having the keyword will be placed.

```
void *CNDImage::GetImageID(CNeoPersist *aObject,
                           const NeoLockType aLock, void *aParam)
{
 CNDImage *  image;

 image = (CNDImage *)aObject;

 ((CArray *)aParam)->InsertAtIndex(&(image->fID), 0x7FFFFFFF);

 return nil;
}
```

The function applied to each object, `GetImageID`, is quite simple. Its arguments are a pointer to an image object and a pointer to the results array. The object id of each image is added to the end of the results array.

Summary

Photographer's Assistant was a very straightforward little application to write. The vast majority of code written for it is necessary to support the user-interface. The storage and retrieval of persistent data and objects in the application is all handled by NeoAccess. This simple application can be used as a pumped up scrapbook if you like. But its real value is as an example of how easily the issue of object persistence can be addressed using NeoAccess.