

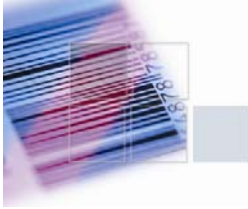
*Coding C++ Applications on
Cache-Forward Architecture
White Paper*

PROGRESS
SOFTWARE

Real Time Division

Table of Contents

<i>Introduction</i>	<i>3</i>
<i>Getting Started with the ObjectStore C++ Interface.....</i>	<i>5</i>
<i>Creating and Opening Databases.....</i>	<i>5</i>
<i>Creating Transactions</i>	<i>6</i>
<i>Creating Persistent Objects</i>	<i>6</i>
<i>Creating Collections of Objects.....</i>	<i>7</i>
<i>Creating Entry Points to Databases</i>	<i>8</i>
<i>Retrieving Entry Points to Databases</i>	<i>8</i>
<i>Creating Cursors for Navigating Collections</i>	<i>8</i>
<i>Querying Collections</i>	<i>9</i>
<i>Optimizing Queries</i>	<i>10</i>
<i>Defining Relationships Among Objects.....</i>	<i>10</i>
<i>Using Persistent Objects.....</i>	<i>11</i>
<i>Conclusion</i>	<i>12</i>
<i>Glossary.....</i>	<i>12</i>
<i>About Progress Real Time Division</i>	<i>15</i>



Introduction

Progress' Cache-Forward™ Architecture (CFA) can help you achieve the performance required by applications that:

- Manage data for rich C++ object models
- Cache enterprise data in the middle tier
- Build real time, high-performance event processing systems, such as financial trading systems, fraud detection, or Radio Frequency Identification

When you are writing applications that take advantage of CFA, CFA's single-level storage model simplifies the way you approach stored data because:

- Data you operate on is in memory.
- There is no mapping from one data format to another.
- You can work with object models that are as complex as you need them to be.
- The interface for writing applications is standard C++.

There is a tight integration between CFA and C++. This lets developers who understand C++ easily learn how to write applications that harness CFA. Writing CFA applications is just like programming with standard C++. With CFA, there is no separate data definition language or data manipulation language – the programming language is the interface.

CFA is the foundation for Progress® ObjectStore® Enterprise, Progress® Apama® EventStore™, and Progress® ObjectStore® PSE Pro®. ObjectStore Enterprise (typically referred to as ObjectStore) is a multiuser, enterprise-class, object database that stores objects in their native format. Progress Apama EventStore is a real-time, Event Stream Processing (ESP) product that manages data for event-driven applications in real time. ObjectStore PSE Pro is a small-footprint object database that is a replacement for file-based storage.

This white paper provides an overview of the ObjectStore C++ API, which is common to all CFA products. The code fragments are part of a complete application that you can download at http://www.progress.com/realtime/publications/coding_on_cfa.

This document is for potential customers who are experienced C++ developers or development managers. No knowledge of ObjectStore or CFA is assumed. New CFA developers will also find this information useful.

For an overview of the ObjectStore Java API see *Coding Java Applications on Cache-Forward Architecture*. You can download both white papers at http://www.progress.com/realtime/publications/coding_on_cfa.

For a detailed description of CFA, see *Introducing Cache-Forward Architecture* <http://www.progress.com/realtime/products/objectstore/index.ssp>.

Getting Started with the ObjectStore C++ Interface

The ObjectStore C++ interface is designed for the development of C++ applications that require data management services. You can use a variety of C++ compilers, together with the CFA C++ interface. The C++ interface is a library of classes that help you write applications that take advantage of CFA's features. Also included are global functions, such as an overloading of operator `new()` that allows persistent storage for any type of object.

Creating persistent data is a lot like creating transient data with plain C++ — you allocate memory and assign a value to that memory. The only difference is that if you want to store persistent data, you allocate persistent memory. What sort of memory you allocate (persistent or transient) is independent of the type of value you want to store.

You can store any type of value in persistent or transient memory. If you can describe it with C++, you can store it. CFA lets you use the full power of C++ to define, create, and manipulate objects. This means full support for the complete language, including inheritance, polymorphism, encapsulation, and standard libraries.

Before you can access persistent memory, you must set the stage by performing a few operations:

- ⇒ Create or open a database
- ⇒ Start a transaction
- ⇒ Retrieve or create a database root

The database features that let your applications scale to handle the ever increasing quantity and complexity of objects include:

- ⇒ Collections
- ⇒ Queries
- ⇒ Relationships

Creating and Opening Databases

CFA applications store objects in databases. Databases are files that a CFA application must create and open. An application must open a database to read from it or write to it.

You create a database by calling the static member function `os_database::create()`. This function also opens the newly created database. For example:

```
db = os_database::create(argv[1], 0644, 1);
```

In this example, the first argument to this function indicates the pathname of the database to be created. The pathname must be UNIX or Windows style according to the type of machine that is hosting the database. The second argument specifies the permissions on the database. The third argument is a Boolean value that indicates whether to overwrite the database if it already exists.

The return value is a pointer to an object of type `os_database`.

To read or write objects to an existing database, you first open that database. You open a database by calling the `os_database::open()` function. For example:

```
db = os_database::open(argv[1]);
```

Creating Transactions

A program must start a transaction before it accesses persistent data. While a transaction is in progress, a program can read and write persistent objects. At any time, a program can commit or abort a transaction to end it. When a program commits a transaction, the server makes permanent in the database any changes to persistent data made during the transaction. The server also makes any updates visible to other processes. When a program aborts a transaction, the server undoes or rolls back any changes made to persistent data during that transaction.

As you can see, transactions do two things:

- ⇒ They identify code sections whose effects can be undone.
- ⇒ They identify functional program areas that are isolated from changes made by other processes (clients). The data seen by a transaction remains consistent throughout that transaction and does not change if another update transaction commits during this transaction. From the point of view of other processes, these functional sections execute either all at once or not at all. That is, other processes do not see intermediate results.

To create a transaction, you declare the transaction boundaries. Below is the syntax for declaring a lexical (as opposed to a dynamic) transaction:

```
OS_BEGIN_TXN(txn-tag, tix-exception, txn-type)
// body
OS_END_TXN(txn-tag)
```

The `txn-tag` argument is an identifier for the transaction. The value you specify in the `OS_BEGIN_TXN` macro must match the value of the argument to the companion `OS_END_TXN` macro.

The `tix-exception` argument points to the location of the exception message if there is an exception. A value of 0 indicates that you are not using this argument. The `txn-type` argument can be `update` or `read_only`. For example:

```
OS_BEGIN_TXN(tx1, 0, os_transaction::update)
// body
OS_END_TXN(tx1)
```

Creating Persistent Objects

When an ObjectStore program stores an object, persistence is not part of the type of the object. You can transiently allocate on the heap or persistently allocate in a database any C++ data type. This includes built-in types, such as integers and character strings, as well as arbitrary user-defined structures, which might contain pointers, virtual functions, and multiple inheritance. There is no need to inherit from a special persistent object base class. Objects of the same type can be persistent or transient within the same program.

An overloading of the C++ `new` operator initializes memory and persistently allocates it. You use `new` to allocate persistent memory just as you would use it to allocate transient memory, except that you supply some additional arguments. The simplest form of `new` requires you to supply two arguments. The `os_database*` argument indicates the database in which to store the new persistent object. The `os_typespec*` argument specifies the type of object you are creating. For example:

```
Customer* aCustomer = new(db,
    Customer::get_os_typespec()) Customer("James Jones");
```

The recommended way to create `os_typespec` objects is to invoke the static method `get_os_typespec()`, which returns the `os_typespec` object. Declare this method in the object's header file and `ObjectStore` generates an implementation of this method:

```
class Customer
{
private:
    // ... private data members ...
public:
    // .. other public methods ...
    static os_typespec* get_os_typespec();
};
```

`ObjectStore` lets you specify where to store a particular object. You can use this facility to improve performance in several ways:

- ⇒ Store particular objects in a way that improves concurrency. Storing objects so that they are not near each other can ensure that access to one object does not interfere with access to some other particular object.
- ⇒ Store particular objects in a way that improves locality. Storing objects near each other can reduce the disk I/O required to access those objects.

Space in a database is divided into containers called segments, and segments are divided into subcontainers called clusters. Whenever an application creates an object in a database, the object belongs to a particular `os_cluster` in that database.

Storing related objects in the same cluster can improve performance because the objects tend to be physically near each other on the disk.

There are overloads of the `new` operator that you can use to cluster related objects. In place of the `os_database*` argument, you could specify an `os_segment*` or `os_cluster*` argument. In the following example, the code clusters an `OrderLine` object with the `Order` object that it belongs to.

```
anOrderLine = new(os_cluster::with(anOrder),
    OrderLine::get_os_typespec()) OrderLine(aBook);
```

Creating Collections of Objects

A collection is an object that groups together other objects. Collections provide a convenient way to store and manipulate groups of objects. They support operations for inserting, removing, and retrieving elements, in addition to operations such as intersection, union, and subset. Collections are analogous to tables in a relational database.

There are APIs for several different collection types, including sets, bags, lists, arrays, and dictionaries. The example below shows the creation of a dictionary that can contain `Customer` objects. The next topic shows how to add objects to a collection.

```
objectstore::initialize();
os_collection::initialize();
OS_ESTABLISH_FAULT_HANDLER
os_database *db = 0;
os_Dictionary<char*,Customer*> *customerDictionary = 0;

// Create a database and start an update transaction:
db = os_database::create(argv[1], 0644, 1);
OS_BEGIN_TXN(tx1, 0, os_transaction::update)
// Get the default segment in the database:
```

```

os_segment* seg = db->get_default_segment();
// Create dictionary collection in its own cluster in the default segment:
customerDictionary = new(seg->create_cluster(),
    os_Dictionary<char*,Customer*>::get_os_typespec())
    os_Dictionary<char*,Customer*>;

```

Creating Entry Points to Databases

Databases contain roots, which serve as entry points into the database and provide a way of assigning persistent names to objects. When an object has a persistent name, any process can retrieve it by looking it up by that name. After you retrieve one object, you can retrieve any object related to it by navigating from one object to another by following pointers. In addition, if a database root is a collection object, you can query the collection to select all elements that satisfy a specified condition.

Each database typically has a relatively small number of roots, each of which provides access to a large network or collection of related objects. You create a root by calling the `os_database::create_root()` function. This function returns a pointer to an instance of the `os_database_root` class. The first argument for the function must be the database that contains the entry points. The second argument is the string value to be associated with the root. For example:

```

// Create database root:
os_database_root* customersRoot = db->create_root("Customers");
// Associate the root to the dictionary:
customersRoot->set_value(customerDictionary,
    os_Dictionary<char*,Customer*>::get_os_typespec());
// Create Customer object and insert it in the customer collection:
Customer jamesj = new(db, Customer::get_os_typespec()) Customer("James Jones");
customerDictionary->insert(jamesj->getName(), jamesj);

```

Retrieving Entry Points to Databases

Each database root's sole purpose is to associate an entry point with a persistent name. After the association is made, you can retrieve a pointer to the entry point by using the `os_database::find_root()` function. For example:

```

// Retrieve the root:
customersRoot = db->find_root("Customers");
// Retrieve the dictionary that is associated with the root
customerDictionary = (os_Dictionary<char*,Customer*>*)
    customersRoot->get_value(os_Dictionary<char*,Customer*>::get_os_typespec());

```

Creating Cursors for Navigating Collections

The collection facility provides a number of classes that help you navigate within a collection. Cursors, index paths, and ranges all help you to insert and remove elements, as well as to retrieve particular elements or sequences of elements.

A cursor, an instance of `os_Cursor`, designates a position within a collection. You can use cursors to traverse collections, as well as to retrieve, insert, remove, and replace elements. When you create a cursor, you specify its associated collection. The initial position of the cursor is at the collection's first element. The following example uses a cursor to iterate over the dictionary of customers and output information about each one.

```

// Open the database and begin a read-only transaction:
db = os_database::open(argv[1]);
OS_BEGIN_TXN(tx1, 0, os_transaction::read_only)
// Retrieve the database root:
os_database_root* customersRoot = db->find_root("Customers");
// Retrieve the collection of customers:
customerDictionary = (os_Dictionary<char*,Customer*>*)
    customersRoot->get_value(os_Dictionary<char*,Customer*>::get_os_typespec());
// Create a cursor for the collection of customers:
os_Cursor<Customer*> c(*customerDictionary);
// Iterate through the objects in the collection:
cout << "Iterate over all Customer objects: " << endl;
for (Customer* aCustomer = c.first(); c.more();
    aCustomer = c.next())
{
    // Output name of each customer:
    cout << "Customer name = " << aCustomer->getName() << endl;
}
OS_END_TXN(tx1)
db->close();

```

Querying Collections

Collections form the basis of the query facility, which allows you to select those elements of a collection that satisfy a specified condition. Thus, in addition to navigational data access, you can execute ad hoc associative retrievals. You can retrieve objects from the database by specifying simple or complex query expressions that return sets of objects from the database.

The following code sample queries a collection of books.

```

db = os_database::open(argv[1]);
OS_BEGIN_TXN(tx1, 0, os_transaction::read_only)
// Retrieve the root for the collection of books:
os_database_root* booksRoot = db->find_root("Books");
// Retrieve collection of books:
os_Dictionary<os_coll_int64,Book*>* bookDictionary =
    (os_Dictionary<os_coll_int64,Book*>*) booksRoot->get_value(
        os_Dictionary<os_coll_int64,Book*>::get_os_typespec());
// Run the query:
os_Collection<Book*>& results1 =
    bookDictionary->query("Book*", "cost > 15.00");
// ... Access the collection.
// Cleanup results collection:
delete &results1;
OS_END_TXN(tx1)

```

By default, dictionaries are unordered collections that allow duplicates. Unlike other collections, dictionaries associate a key with each element. The key can be a value of any C++ fundamental type, user-defined type, or pointer type. When you insert an element into a dictionary, you specify the key along with the element. The following code shows how you can retrieve an element with a given key:

```

//Look up a Book object by its key
os_coll_int64 ISBN = 395193958;
Book* aBook = bookDictionary->pick(ISBN);

```

In an ordered dictionary, you can retrieve those elements whose keys fall within a given range. For example:

```
os_Cursor<Data*> c(ordered_dict,
    os_coll_range(os_collection::GE, 10, os_collection::LE, 20));
for (Data* d = c.first(); c.more(); d = c.next())
    do_something(d);
```

This does something to all the dictionary elements with keys between 10 and 20.

Optimizing Queries

The query optimizer minimizes the number of objects examined in response to a query. You can take advantage of query optimization by creating indexes for collections of objects over which queries are to be performed. Each index allows fast associative retrieval based on a given key. The key to an index can be a data member of the set element. For example, suppose you want to optimize the following query:

```
os_Collection<Book*>& results2 =
    bookDictionary->query("Book*", "strcmp(author,\"J.K. Rowling\")==0");
```

The following code adds an index that optimizes this query. The key to the index is the author data member. The query uses the index, rather than scanning all books, to quickly find the books by J.K. Rowling.

```
// Add index:
os_index_path& idxPath = os_index_path::create("Book*", "author", db);
bookDictionary->add_index(idxPath);
// Cleanup the index path:
delete &idxPath;
```

In addition, you can set up a multilevel index on a path, consisting of a series of data members that span multiple objects. For example, a set of books could have an index that permits efficient retrieval based on the name of the publisher.

In an application built on CFA, you can create any data structure that you can create in memory. For example, you can build an application that presents geographic data by means of a quadtree. CFA renders the data in the database exactly as it is represented in memory. Consequently, you can build a specialized index that is uniquely tuned to the data in the quadtree.

A program can add or drop an index at any time.

Note that a dictionary collection is a structure that is already optimized for looking up an object by its key. The query optimization feature described here applies to non-key data members. If the collection is not a dictionary, the query optimization information applies to all data members.

Defining Relationships Among Objects

The relationship facility models binary relationships between pairs of objects and maintains referential integrity when required. The class library contains relationship and collection classes that simplify the definition and maintenance of relationships. You can model one-to-one, one-to-many, and many-to-many relationships among objects. A unidirectional, many-to-one relationship is an ordinary C++ pointer. You do not need to use any macros to define one. Relationships can also be bidirectional. The following code defines a one-to-many relationship between a customer and orders. That is, one customer can place multiple orders.

```

#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/relat.hh>
class Order;
class Customer
{
private:
// ... other data members
// Relationship to the customer's orders:
os_relationship_m_1(Customer,orders,Order,customer,os_List<Order*>) orders;
public:
// ... public methods
};

```

To define a class that has relationships, you define a data member using an appropriate relationship macro. The relationship macro defines the appropriate access functions for getting and setting the relationship. In the previous example, the relationship macro is `os_relationship_m_1`. In this example of the macro,

- ⇒ `Customer` is the class that is defining the data member that is one side of the relationship.
- ⇒ `orders` is the relationship data member being declared.
- ⇒ `Order` is the class that defines the data member that has the relationship with orders.
- ⇒ `customer` is the name of the `Order` class data member that has the relationship with orders.
- ⇒ `os_List<Order*>` is the value type of orders. That is, orders is a collection of pointers to `Order` objects.

To complete the relationship definition, the `Order` class must define the other side of the relationship.

Using Persistent Objects

As mentioned earlier, all access to persistent objects must be from within a transaction. Also, you cannot hold or use pointers to persistent objects across transaction boundaries. Except for these constraints, you can use pointers to persistent objects in the same way that you use pointers to transient objects, which have been dynamically allocated in the program's virtual address space. Pointers to persistent objects always take the form of virtual memory pointers.

Obtain initial access to persistent objects through database roots, navigation or an associative query. You then use standard C++ dereference operators to access persistent objects.

For details about the ObjectStore locking model, see the *Introducing Cache-Forward Architecture* white paper. Numerous examples of using persistent objects are in the sample programs. You can download both the white paper and the sample programs at <http://www.progress.com/realtime/products/objectstore/index.ssp>.

Conclusion

When you are writing applications that take advantage of CFA, ObjectStore's single-level storage model simplifies the way you approach stored data because:

- ⇒ Data you operate on is in memory.
- ⇒ There is no mapping from one data format to another.
- ⇒ You can work with object models that are as complex as you need them to be.
- ⇒ The interface for writing applications is standard C++.

The basic operations when coding a CFA application are

- ⇒ Creating and opening databases
- ⇒ Working with objects inside transactions
- ⇒ Creating persistent objects
- ⇒ Creating collections of objects
- ⇒ Creating entry points to databases
- ⇒ Retrieving entry points to databases
- ⇒ Creating cursors for navigating collections
- ⇒ Querying collections
- ⇒ Optimizing queries
- ⇒ Defining relationships among objects
- ⇒ Using persistent objects just as you would any objects in a C++ application

The best way to develop an application that leverages CFA is to

- ⇒ Obtain training from Progress.
- ⇒ Read Progress white papers about applications that are similar to yours.
- ⇒ Apply the appropriate design principles for your goals.

Glossary

This glossary defines CFA concepts as well as features supported by ObjectStore.

archive logging	Records transaction activity, which provides the information you need to recover modifications made after a backup.
backup and restore	Copies data to secondary storage, and restores data from secondary storage to primary storage.
CFA server	CFA servers manage physical data on disk, arbitrate among CFA client processes that are requesting objects, and ensure that all clients have consistent views of data.
CFA client	CFA clients provide the interface between your application and the CFA server. Clients apply your business rules to a logical view of your data. For example, in a real time data processing application, a CFA client can perform a query over a body of data that includes new data coming in as well as historical data.

client page cache	On each client host, CFA maintains a page cache for each client process. The client page cache is an internal CFA file that holds pages that contain objects your application has recently used.
clustering	Lets you specify where to store an object. For example, if there is a group of objects that you frequently use together you can store them near each other to increase locality. Conversely, you can store objects in different locations to ensure concurrency.
CMTL	The C++ Middle-Tier Library provides a data caching technology that moves processing away from the back end and concentrates it in the middle tier, making data readily available to the business logic at in-memory speed while ensuring data consistency and transactional integrity.
collections	Objects such as sets, bags, or lists that serve to group together other objects.
compaction	Frees storage space so it can be used by other objects.
concurrency control	Simultaneous access to objects that ensures that an update to one object does not interfere with an update to another object.
deadlock	Two clients are waiting for resources. Each client is waiting for the resources being used by the other client.
deadlock detection	Detects and breaks deadlocks by aborting one of the transactions involved in the deadlock. This causes the victim's locks to be released so that other processes can proceed.
dump/load	Lets you dump data into an ASCII file and generate a loader executable that is capable of creating, given the ASCII file as input, an equivalent database or group of databases.
failover	Ensures that the failure of a CFA server or the machine it is running on does not prevent service to CFA clients. Failure of a protected server is immediately detected, and its service is picked up by another server running on its own machine, providing clients with continued access to objects.
heterogeneity	A CFA server can support a client of any type regardless of the client's data format, byte ordering, floating-point representation, or data alignment. A client with one processor architecture can generate data to be read by client applications residing on machines with processors that are different from the original client. For example, a Sun server can support Sun, Hewlett-Packard, IBM, and Windows clients simultaneously.
JMTL	The Java Middle-Tier Library provides transparent, high-performance storage for Java objects. JMTL caches objects accessed by client-initiated transactions, maintains the consistency and recoverability of the transactional caches, maintains each transaction's required isolation level, and schedules transactions to optimize throughput.
locality	Degree to which objects that are used together are stored near each other on the CFA server.
metaobject protocol	Library of classes that allows you to access ObjectStore schema information.

MVCC	Multi-Version Concurrency Control allows objects that are being accessed by multiple read-only transactions to be simultaneously updated by one update transaction.
notification	Lets a CFA client notify other clients that an event has taken place. Typically, the event is a change in the objects to which other clients have access. Using the notification service, clients can send and receive notifications.
page fault	Operating system mechanism that notifies CFA that a client is trying to access an object. CFA has its own page fault handler.
queries	Queries return a collection that contains those elements of a given collection that satisfy a specified condition.
query indexes	The query optimizer maintains indexes into collections based on user-specified keys, that is, data members, or data members of data members, and so on. With these indexes, implemented as B-trees or hash tables, you can minimize the number of objects examined in response to a query. Formulation of optimization strategies is performed automatically by the system.
recovery	Process of returning a database to a transactionally-consistent state if any process aborts or any host crashes, or in the event of network failure.
relationship facility	Models binary relationships between pairs of objects and maintains referential integrity when required.
schema evolution	Modification of schema information associated with stored objects, and modification of any existing instances of the modified classes.
storage cache	ObjectStore database used for temporary storage during application run time.
transactional cache	Transactional caching is the underlying concept of the CFA middle-tier libraries. Client requests are automatically routed to separate CMTL and JMTL caches, allowing in-memory access to cached objects. Transactional caches ensure data integrity by providing transactional consistency among the caches, even while servicing multiple, concurrent transactions against the same data.
transactions	Execution of a sequence of statements that operate on objects as a logical unit. That is, the operations performed by the statements are performed all together or not at all.
VMM	Virtual Memory Mapping is the element of CFA that dynamically maps objects into memory.



About Progress Real Time Division

The Progress Real Time Division provides event stream processing, data management, data access and synchronization products to enable the real-time enterprise. Our products manage and analyze real-time event stream data for applications such as algorithmic trading and RFID; accelerate the performance of existing databases through sophisticated caching; manage and process complex data in the industry's leading object database; and support occasionally connected mobile users requiring real-time access to enterprise applications. The Progress Real Time Division is an operating unit of Progress Software Corporation (Nasdaq: PRGS), a global software industry leader. Headquartered in Bedford, Mass., they can be reached at www.progress.com/realtime or +1-781-280-4000.

PROGRESS
SOFTWARE

Real Time Division

www.progress.com/realtime

Worldwide and North American Headquarters

Progress Real Time Division, 14 Oak Park, Bedford, MA 01730 USA Tel: +1 781 280 4000

UK Office and Northern Ireland

Progress Real Time Division, 210 Bath Road, Slough, Berkshire, SL1 3XE England Tel: +44 1753 216 300

Central Europe

Progress Real Time Division, Konrad-Adenauer-Str. 13, 50996 Köln, Germany Tel: +49 6171 981 127

France

Progress Real Time Division, 3 Place de Saverne, Les Renardières B, 92901 Paris la Défense Tel: +33 1 41 16 16 56

© 2006 Progress Software Corporation. All rights reserved. Progress, ObjectStore, Apama, and Cache-Forward are trademarks or registered trademarks of Progress Software Corporation, or any of its affiliates or subsidiaries, in the U.S. and other countries. Any other trademarks or service marks contained herein are the property of their respective owners. Specifications subject to change without notice. Visit www.progress.com/realtime for more information.