Using OCCI:
An Introduction to Objects

An Oracle White Paper
July 2003

ORACLE

Using Objects with OCCI

INTRODUCTION

With the Oracle C++ Call Interface (OCCI) and the Oracle Database Server, application developers can finally use Object Oriented techniques easily and naturally. Many C++ developers are already aware of the benefits of using Objects to represent logically grouped pieces of information; these developers can now leverage their knowledge and experience with the strength, security and scalability of the Oracle Database to create new types of Object Oriented applications that manage their information stores with the database, assuring reliability and transactional consistency for their data.

Designed from the ground up to be Object Oriented, OCCI naturally supports Objects created in the Oracle Database as naturally as other C++ objects. The Oracle Database enhances the versatility of a relational database with the convenience of native Objects to manage the ever increasingly complex information needed in the current environment.

Support for Objects using OCCI comes in two different forms: Associative Access and Navigational Access. **Associative Access** allows C++ applications that already access the Oracle Database using standard SQL to easily add the benefits of Oracle Objects to manage more complex information. For applications that are built to be completely Object Oriented, **Navigational Access** provides a fully Object Oriented API to seamlessly integrate Oracle Database functionality within the application.

This paper gives an overview of Objects and how to use them in your C++ applications. Code snippets are included to concretely demonstrate how to use OCCI for the different operations. However, the paper does not cover every facet of OCCI usage. Please consult the *Oracle C++ Call Interface Programmer's Guide* for more complete details of the areas that most interest you.

OBJECT CHARACTERISTICS AND TYPES OF ACCESS

Oracle Objects (denoted as "Objects" with a capital O throughout the paper) allow developers to group information into single entities using the Object Oriented Programming paradigm. For example, a Person object (an instance of the Person object type) may have information related to a particular person such as the name and age. Manipulating Person objects often prove much easier than individually keeping track of the separate variables for the name and age for every Person object.

Instances of Object types can be viewed as one of two types: **Transient Objects** and **Persistent Objects**. Transient Objects are simply copies of the Object data that exist on the client application side. Persistent Objects are both Object data and their link to the Oracle Database server at a specific location in storage.

Access to Objects is also one of two types, depending on the type of the Object accessed. When we want transient Objects, we use **Associative Access**. On the other hand, persistent Objects require the use of **Navigational Access**. For the rest of the paper, many sections will be split into two parts, one for Navigational Access to persistent Objects and another for Associative Access to transient Objects.

OBJECT USAGE

Creating Types

In order to manipulate Objects, the types must first be known by the database. As an example, we will create the Employee type:

```
create type EMPLOYEE_T as OBJECT
(last_name varchar2(30),
 first_name varchar2(30),
 years number(3));
```

Once created in the database, we can use the Object Type Translator (OTT) to create the C++ representations of the types. OTT will generate the appropriate C++ headers and source files to allow your application to interface with your newly created types. For a more in-depth discussion and the usage of OTT, please see the *Oracle C++ Call Interface Programmer's Guide, Chapter 7*.

With the option of attribute access set to private, OTT generates:

```
/*************************************************************/
//  generated declarations for the EMPLOYEE_T object type.
/*************************************************************/

class Employee : public oracle::occi::PObject {
private:
   OCCI_STD_NAMESPACE::string LAST_NAME;
   OCCI_STD_NAMESPACE::string FIRST_NAME;
   oracle::occi::Number YEARS;

public:

   //accessors
   OCCI_STD_NAMESPACE::string getLast_name() const;
   void setLast_name(const OCCI_STD_NAMESPACE::string &value);
   OCCI_STD_NAMESPACE::string getFirst_name() const;
   void setFirst_name(const OCCI_STD_NAMESPACE::string &value);
   oracle::occi::Number getYears() const;
   void setYears(const oracle::occi::Number &value);

   //constructors
   void *operator new(size_t size);
   void *operator new(size_t size,
        const oracle::occi::Connection * sess,
        const OCCI_STD_NAMESPACE::string& table);
   void *operator new(size_t, void *ctxOCCI_);
   OCCI_STD_NAMESPACE::string getSQLTypeName() const;
   Employee();
   Employee(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   //auxiliary
   static void *readSQL(void *ctxOCCI_);
   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
   static void writeSQL(void *objOCCI_, void *ctxOCCI_);
   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
   ~Employee();
};
```

The Employee class inherits from the oracle::occi::PObject class so that persistent Objects can be created. Specific details of the PObject class are beyond the scope

of this paper. See *Oracle C++ Call Interface Programmer's Guide, Appendix A* for more details.

Now that the database knows of the Employee type, we can begin employing Employee Objects in our applications.

Creating an Object Table in the Database

Before applications can use Objects in the database, a table to hold the Objects must be created. To create a table named employees that holds employee_t Objects, use the following SQL:

```
CREATE TABLE employees OF employee_t;
```

After creating the table, we can create an application that can store Objects in the database.

Creating the Environment and Connection

In order to use Objects with OCCI in an application, first we need to create and initialize the OCCI environment for Object support:

```
Environment* env = createEnvironment( Environment::OBJECT );
```

Without initializing the Environment with Object capabilities, we do not have access to the Object capabilities of OCCI. Next, create the Connection to access Objects in the database using the SCOTT/TIGER schema in the ORCL database:

```
Connection* conn = createConnection( "scott", "tiger", "orcl" );
```

With the creation of the Connection, we are ready to make and use Objects in the rest of the application.

Instantiating New Objects

When instantiating an Object, the developer must first decide the level of persistency to assign to the Object. Often, for variables used only on the application side, transient Objects are used. On the other hand, Objects that need to be passed between the application and the server must be declared as persistent Objects. Since the persistency cannot change once the Object is declared, the developer must be careful to choose the appropriate level of persistency.

Associative Access

Transient Objects "look" like the types in the database, but are only temporary Object copies that last for the duration of the application. When these Objects are created, they are not connected to the database in any way. Typically, instantiating transient Objects looks the same as instantiating other C++ objects:

```
Employee* emp = new Employee();
```

Once instantiated, the new transient Object behaves like any other C++ object. Note that transient Objects are assigned to pointers.

Navigational Access

Persistent Objects must be related to a specific connection/table in order to be declared. Now we can create Objects in our C++ applications:

```
Ref<Employee> employee_ref = new(conn, "EMPLOYEES") Employee();
```

This statement connects the newly created employee Object to the EMPLOYEES table. Notice that we use references (Ref<T> class) instead of pointers for persistent Objects. For persistent Objects, references should always be used so that the client-side Object cache can manage its memory correctly.

Retrieving Objects from a Database Table

Associative Access

To get a copy of an Object from the EMPLOYEES table using Associative Access, we need to issue some SQL to retrieve the Object and then convert the results to the C++ Object, as shown in the following code:

```
Statement* stmt = conn->createStatement();

//retrieves the Object from the table
string sel_emp = "SELECT VALUE(e) FROM employees e WHERE
       last_name='DOE' AND first_name='JANE'";
ResultSet* rs = stmt->executeQuery( sel_emp );
rs->next();

//convert result into copy of the Object
//the 1 is the "column" of the result
Employee* emp = rs->getObject(1);
```

Navigational Access

Navigational Access only requires issuing SQL when retrieving the first set of persistent Objects from the database, assuming that the Object hierarchy is designed properly. We can retrieve other Objects using references embedded in the

initial set of Objects retrieved through SQL. Here is an example of retrieving an Employee Object with a name "JANE DOE":

```
//create a statement handle to execute a SELECT
Statement* stmt = conn->createStatement();
string sel_emp = "SELECT REF(e) FROM employees e WHERE last_name='DOE'
       AND first_name='JANE'";
ResultSet* rs = stmt->executeQuery( sel_emp );
rs->next();

//get the reference for the Object
Ref<Employee> emp_ref = rs->getRef(1);
```

Notice that the difference between the Associative and Navigational Access SQL is the use of REF(e) instead of VALUE(e). REF(e) is a reference to a particular Object row in an Object table, while VALUE(e) simply copies the values into a temporary Object. Hence, persistent Employee Objects are always assigned to Ref<Employee> and should never be assigned to Employee pointers.

TIP: Create a method that retrieves Objects with certain attributes such as the last_name to simplify and minimize maintenance of hardcoded SQL.

Manipulating Objects

Since Objects only become useful when they can be manipulated, let us add and change the available manipulators to the Employee class to tailor the usage to our expected behavior and maintain the representation invariant. First, let us first remove the ability to create an Employee Object with no information. Thus, we will remove this constructor:

```
~~Employee::Employee();~~
```

and replace it with:

```
Employee::Employee(const OCCI_STD_NAMESPACE::string &last_name, const
       OCCI_STD_NAMESPACE::string &first_name, const
       oracle::occi::Number &years);
```

Then, if we believe an Employee will never change their name (except for special cases), we should also remove the ability to set the name:

```
~~void setLast_name(const OCCI_STD_NAMESPACE::string &value);~~
~~void setFirst_name(const OCCI_STD_NAMESPACE::string &value);~~
```

Finally, we would like to only be able to add years to a person's length of employment:

```
void setYears(const oracle::occi::Number &value);
void addYears(const oracle::occi::Number &value)
```

```
{
        //add to employee's years of service
        this.YEARS = this.YEARS + value;

        //mark object as modified for flushing later
        this.markModified();
}
```

Given the changes we have made to our code, we can now create and modify Objects. The main differences between the Associative and Navigational Access versions of the following code snippets are the use of pointers/references for Objects and the SELECT of VALUE/REF.

Associative Access

```
//set up the Environment and Connection
Environment* env = createEnvironment( Environment::OBJECT );
Connection* conn = env->createConnection( "scott", "tiger", "orcl");

//create a transient Employee Object
Employee *new_emp = new Employee( "Doe", "Jane", 0 );

//add 5 years of experience to Jane's employment
new_emp->addYears( 5 );

//retrieve a copy of an Employee Object from the database
Statement* stmt = conn->createStatement();
string sel_emp = "SELECT VALUE(e) FROM EMPLOYEES e WHERE last_name='Q'
        AND first_name='John'";
ResultSet* rs = stmt->executeQuery( sel_emp );
rs->next();
Employee* old_emp = rs->getObject(1);

//add 4 years of experience to John Q's employment
old_emp->addYears( 4 );
```

Navigational Access

```
//set up the Environment and Connection
Environment* env = createEnvironment( Environment::OBJECT );
Connection* conn = env->createConnection( "scott", "tiger", "orcl");

//create a persistent Employee Object for later writing to the database
Ref<Employee> new_emp_ref = new( conn, "EMPLOYEES" ) Employee( "Doe",
        "Jane", 0 );

//add 5 years of experience to Jane's employment
new_emp_ref->addYears( 5 );

//retrieve an Employee Object from the database
Statement* stmt = conn->createStatement();
string sel_emp = "SELECT REF(e) FROM EMPLOYEES e WHERE last_name='Q'
        AND first_name='John'";
ResultSet* rs = stmt->executeQuery( sel_emp );
rs->next();
Ref<Employee> old_emp_ref = rs->getRef(1);

//add 4 years of experience to John Q's employment
old_emp_ref->addYears( 4 );
```

In other words, both transient and persistent Objects look and act just like any other C++ object. In the previous code snippets, the first half creates a new Object for Jane Doe, giving her more experience. The second half retrieves an Object for John Q, adding more experience to his employment as well. In either case, both versions of the code have the C++ object feel that is expected.

NOTE: For retrieved persistent Objects, the client side does a **pinning** of the Object upon the first dereference. The pinning process involves bringing the Object data over to the client side object cache for access and modification of the Object attributes. In the above Navigational Access code, the pin occurs at old_emp_ref->addYears(4), since it is the first dereference of a retrieved object. However, this operation is completely invisible to the application developer.

Marking Modified Objects

Associative Access

Nothing needs to be done for marking modified transient Objects in Associative Access, since they are only copies of Objects and are not tied to any particular Object in the database. Note that the PObject::markModified method has no effect for transient Objects.

Navigational Access

Since persistent Objects are tied to specific rows in the database, but are cached on the client application side, they must be marked as modified for later flushing to the database. In order to mark modified Objects for flushing, use the following call:

```
old_emp_ref->markModified();
```

This call makes sure that changes to this Object will be written to the database server when you flush the Object cache. In the addYears() method code snippet of the Manipulating Objects section, we used the markModified() method to automatically mark Objects that are modified, simplifying bookkeeping.

TIP: One way to mark the objects as needed is to call the markModified() method in every state-changing method.

Flushing/Writing Objects to the Database

Associative Access

Writing a transient Object to the database requires using some SQL DML, such as an INSERT or an UPDATE, depending on if you are inserting a new Object or modifying an existing Object. Inserting a new Object is demonstrated here:

```
stmt->setSQL( "INSERT INTO employees VALUES(:1)" );
stmt->setObject( 1, new_emp );
stmt->executeUpdate();
```

Navigational Access

For flushing Objects, at the appropriate times, you can use either of the following two methods to flush Objects to the server, which preserve the order of modification:

```
//do a flush of all dirty Object Cache Objects
conn->flushCache();

//commit a transaction, flushing all dirty Object Cache entries
conn->commit();
```

The flushCache() method writes the dirty Objects to the server, but does not commit the transaction, so rollbacks can still occur. The commit() method, on the other hand, does a commit of the transaction, and makes sure to write all changes permanently.

Deleting Objects

Associative Access

Newly created transient Objects need only be freed in memory to be deleted, since they are not tied to any specific copy of the Object in the database. Objects in the database can be deleted using the standard SQL DELETE statement.

Navigational Access

Persistent objects need to be marked for deletion using PObject::markDelete method:

```
old_emp_ref.markDelete();
```

After marking for deletion, the Object will be deleted upon a commit.

Freeing Objects

Associative Access

Transient Objects need to be freed using standard memory management methods at the end of their use.

Navigational Access

Freeing objects requires using the PObject::operator delete method. This method frees the memory, but does not automatically delete persistent objects from the database server:

```
delete old_emp_ref;
```

ADVANCED TOPICS

The rest of the paper deals with more advanced topics related to Complex Objects in relation to Navigational Access. Complex Objects are Objects that themselves contain other Objects as part of their representation. The following techniques are not required to have a functionally correct application. However, using some of these techniques may improve the performance of some intensive applications.

Creating a Relational Table with Object Columns

In addition to Object Tables, Objects can be stored in columns of Relational Tables. Since cells cannot be individually addressed, Object columns can only be used with Associative Access to retrieve copies into transient Objects. Following is a SQL example for creating a table of projects with project names and the associated employee:

```
CREATE TABLE projects
(NAME  varchar2(30),
 EMP employee_t);
```

Accessing Object Columns in a Relational Table

Accessing Object columns is very similar to accessing other Oracle primitive types using SQL. Since Object Columns are only accessible using Associative Access, a Navigational Access example is not given. The following example demonstrates how to retrieve a copy of an Employee Object:

```
stmt->setSQL( "SELECT emp FROM projects WHERE name='blueprint'" );
ResultSet* rs = stmt->executeQuery();

rs->next();

//get the Object from the ResultSet
Employee* emp = rs->getObject(1);
```

The difference from the above code snippet and accessing an Object row from an Object table is the SELECT on a column instead of VALUE(table_alias). All other operations such as UPDATE, INSERT, and DELETE can be achieved using the same syntax as the above example.

Complex Object Retrieval

Often, Objects in the database are composed of some primitive types and a number of other Objects. Using only the mechanisms in the previous sections would lead to severe performance penalties when accessing these complex objects. With complex object retrieval (COR), it is possible to increase the performance of applications that require the use of complex objects. COR is a prefetching

mechanism that saves network roundtrips that would otherwise occur to access objects individually, and thus does not affect functionality, only performance. While the objects are prefetched to the object cache, they are not yet pinned. Subsequent pin calls are local operations, however.

Assuming that these types are defined:

```
CREATE TYPE customer(…);
CREATE TYPE line_item(…);
CREATE TYPE line_item_varray as VARRAY(100) of REF line_item;
CREATE TYPE purchase_order as OBJECT
( po_number           NUMBER,
  cust         REF customer,
  related_orders      REF purchase_order,
  line_items   line_item_varray )
```

it is possible to fetch all elements of a purchase_order using COR to a depth of 5 by executing the following code snippet:

```
Ref<PURCHASE_ORDER> poref;
unsigned int depth = 5;
poref.setPrefetch( depth );
```

In addition, it is possible to set the prefetch limit for individual types in the object. For example:

```
Ref<PURCHASE_ORDER> poref;
poref.setPrefetch("CUSTOMER", 1);
```

fetches the customer data every time a purchase_order is fetched. Thus, you have very fine-tuned control of the prefetching by defining the COR mechanism appropriately:

```
Ref<PURCHASE_ORDER> poref;
poref.setPrefetch("CUSTOMER", 1);
poref.setPrefetch("LINE_ITEM", 1);
```

Remember that the more objects that are prefetched, the more network roundtrips that are saved. However, prefetching too many unused objects will flood the object cache, sometimes resulting in even pinned objects being flushed from the cache, leading to performance degradation rather than enhancement.

Collections

Collections in Oracle come in two flavors: ordered collections(VARRAY) and unordered collections (nested tables).

```
CREATE TYPE ADDR_LIST AS VARRAY(3) OF REF ADDRESS;
CREATE TYPE PERSON AS OBJECT (name VARCHAR2(20), addr_l ADDR_LIST);
```

becomes the following C++ declaration with OTT:

```
class PERSON : public PObject
{
       protected:
              string name;
              vector< Ref< ADDRESS > > addr_l;

       public:
              void* operator new(size_t size);
              void* operator new(size_t size,
                                 const Session* sess,
                                 const string& table);
              string getSQLTypeName( size_t size );
              PERSON (void *ctx) : Pobject(ctx){};
              static void* readSQL(void* ctx);
              virtual void readSQL(AnyData& stream);
              static void writeSQL(void* obj, void* ctx);
              virtual void writeSQL(AnyData& stream);
}
```

Using OCCI: An Introduction to Objects
July 2003
Author: Toliver Jue

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com