14

C DOM API for XMLType

The C DOM API for `XMLType` lets you operate on `XMLType` instances using a DOM in C.

- Overview of the C DOM API for XMLType
  The C DOM API for `XMLType` is a DOM API that is used for Oracle XML Developer's Kit (XDK) and Oracle XML DB. You can use it for XML data that is inside or outside the database.
- Access to XMLType Data Stored in the Database Using OCI
  Oracle XML DB provides support for storing and manipulating XML instances using abstract data type `XMLType`. These XML instances can be accessed and manipulated on the client side using the Oracle Call Interface (OCI) in conjunction with the C DOM API for XML.
- Creating XMLType Instances on the Client
  You can construct new `XMLType` instances on the client side using the C DOM API methods `XMLCreateDocument()` and `XmlLoadDom()`.
- XML Context Parameter for C DOM API Functions
  An *XML context* is a required parameter for all the C DOM API functions. This opaque context encapsulates information about the data encoding, the error message language, and so on. The contents of the context are different for Oracle XML Developer's Kit applications and Oracle XML DB.
- Initializing and Terminating an XML Context
  An example illustrates a C program that uses the C DOM API to construct an XML document and save it to Oracle Database.
- Using the C API for XML with Binary XML
  You can use the C API for XML to read or write XML data that is encoded as binary XML from or to Oracle XML DB. Doing so involves the usual read and write procedures.
- Using the Oracle XML Developer's Kit Pull Parser with Oracle XML DB
  You can use the Oracle XML Developer's Kit pull parser with `XMLType` instances in Oracle XML DB. When you use this parser, parsing is done on demand, so your application drives the parsing process.
- Common XMLType Operations in C
  Common XML operations are provided by the C API for XML.

Overview of the C DOM API for XMLType

The C DOM API for `XMLType` is a DOM API that is used for Oracle XML Developer's Kit (XDK) and Oracle XML DB. You can use it for XML data that is inside or outside the database.

DOM refers to compliance with the World Wide Web Consortium (W3C) DOM 2.0 Recommendation.

The C DOM API for `XMLType` also includes performance-improving extensions that you can use in XDK for traditional storage of XML data, or in Oracle XML DB for storage as an `XMLType` column in a table.

> **Note:**
>
> C DOM functions from releases prior to Oracle Database 10g Release 1 are supported only for backward compatibility.

The C DOM API for `XMLType` is implemented on `XMLType` in Oracle XML DB. In the W3C DOM Recommendation, the term "document" is used in a broad sense (URI, file system, memory buffer, standard input and output). The C DOM API for `XMLType` is a combined programming interface that includes all of the functionality needed by Oracle XML Developer's Kit and Oracle XML DB applications. It provides XSLT and XML Schema implementations. Although the DOM 2.0 Recommendation was followed closely, some naming changes were required for mapping from the objected-oriented DOM 2.0 Recommendation to the flat C namespace. For example, method `getName()` was renamed to `getAttrName()`.

The C DOM API for `XMLType` supersedes older Oracle APIs. In particular, the `oraxml` interface (top-level, DOM, SAX, and XSLT) and `oraxsd.h` (Schema) interfaces will be deprecated in a future release.

The reference documentation for the C and C++ Application Programming Interfaces (APIs) that you can use to manipulate XML data is *Oracle Database XML C API Reference*, and *Oracle Database XML C++ API Reference*.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL APIs for XML
> - *Oracle Database XML Java API Reference* for information about Java APIs for XML

Access to XMLType Data Stored in the Database Using OCI

Oracle XML DB provides support for storing and manipulating XML instances using abstract data type `XMLType`. These XML instances can be accessed and manipulated on the client side using the Oracle Call Interface (OCI) in conjunction with the C DOM API for XML.

You can bind and define `XMLType` values using the C DOM structure `xmldocnode`. This structure can be used for binding, defining and operating on XML values in OCI statements. You can use OCI statements to select XML data from the server, which you can then use with C DOM API functions. Similarly, values can be bound back to SQL statements directly.

The main flow for an application program involves initializing the usual OCI handles, such as server handle and statement handle, and then initializing an XML context parameter. You can then either operate on XML instances in the database or create new instances on the client side. The initialized XML context can be used with all of the C DOM functions.

Related Topics

- XML Context Parameter for C DOM API Functions
  An *XML context* is a required parameter for all the C DOM API functions. This opaque context encapsulates information about the data encoding, the error message language,

and so on. The contents of the context are different for Oracle XML Developer's Kit applications and Oracle XML DB.

Creating XMLType Instances on the Client

You can construct new `XMLType` instances on the client side using the C DOM API methods `XMLCreateDocument()` and `XmlLoadDom()`.

You can construct empty `XMLType` instances using `XMLCreateDocument()`. This is similar to using `OCIObjectNew()` for other types.

You construct a non-empty `XMLType` instance using `XmlLoadDom()`, as follows:

1. Initialize the `xmlctx` as in Example 14-1.
2. Construct the XML data from a user buffer, local file, or URI. The return value, a (`xmldocnode*`), can be used in the rest of the common C API.
3. If required, you can cast (`xmldocnode *`) to (`void*`) and provide it directly as the bind value.

XML Context Parameter for C DOM API Functions

An *XML context* is a required parameter for all the C DOM API functions. This opaque context encapsulates information about the data encoding, the error message language, and so on. The contents of the context are different for Oracle XML Developer's Kit applications and Oracle XML DB.

For Oracle XML DB, OCI functions `OCIXmlDbInitXmlCtx()` and `OCIXmlDbFreeXmlCtx()`, respectively, initialize and terminate an XML context.

- OCIXmlDbInitXmlCtx() Syntax
  OCI function `OCIXmlDbInitXmlCtx()` initializes an XML context.
- OCIXmlDbFreeXmlCtx() Syntax
  OCI function `OCIXmlDbFreeXmlCtx()` terminates an XML context.

OCIXmlDbInitXmlCtx() Syntax

OCI function `OCIXmlDbInitXmlCtx()` initializes an XML context.

The syntax of `OCIXmlDbInitXmlCtx()` is as follows:

```
xmlctx *OCIXmlDbInitXMlCtx (OCIEnv        *envhp,
                            OCISvcHp      *svchp,
                            OCIError      *errhp,
                            ocixmldbparam *params,
                            ub4           num_params);
```

Table 14-1 describes the parameters.

Table 14-1 OCIXmlDbInitXMlCtx() Parameters

Parameter	Description
`envhp (IN)`	The OCI environment handle.

Table 14-1 (Cont.) OCIXmlDbInitXMlCtx() Parameters

Parameter	Description
`svchp (IN)`	The OCI service handle.
`errhp (IN)`	The OCI error handle.
`params (IN)`	An array of optional values: • OCI duration. Default value is `OCI_DURATION_SESSION`. • Error handler, which is a user-registered callback: `void (*err_handler) (sword errcode, (CONST OraText *) errmsg);`
`num_params (IN)`	Number of parameters to be read from `params`.

OCIXmlDbFreeXmlCtx() Syntax

OCI function `OCIXmlDbFreeXmlCtx()` terminates an XML context.

The syntax of `OCIXmlDbFreeXmlCtx()` is as follows, where parameter `xctx (IN)` is the XML context to terminate.:

```
void OCIXmlDbFreeXmlCtx (xmlctx *xctx);
```

Initializing and Terminating an XML Context

An example illustrates a C program that uses the C DOM API to construct an XML document and save it to Oracle Database.

Example 14-1 shows this. The document constructed is stored in table `my_table`. OCI functions `OCIXmlDbInitXmlCtx()` and `OCIXmlDbFreeXmlCtx()` are used to initialize and terminate the XML context, respectively. These functions are defined in header file `ocixmldb.h`.

The code uses helper functions `exec_bind_xml`, `init_oci_handles`, and `free_oci_handles`, which are not listed here. The complete listing of this example, including the helper functions, can be found in Oracle-Supplied XML Schemas and Examples, Initializing and Terminating an XML Context (OCI).

The C code in Example 14-1 assumes that the following SQL code has first been executed to create table `my_table` in database schema `capiuser`:

```
CONNECT CAPIUSER
Enter password: password

Connected.

CREATE TABLE my_table OF XMLType;
```

Example 14-4 queries table `my_table` to show the data that was inserted by Example 14-1.

Example 14-1 Using OCIXMLDBINITXMLCTX() and OCIXMLDBFREEXMLCTX()

```
#ifndef S_ORACLE
#endif
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif
#ifndef XML_ORACLE
#include <xml.h>
#endif
#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <string.h>

typedef struct test_ctx {
        OCIEnv *envhp;
        OCIError *errhp;
        OCISvcCtx *svchp;
        OCIStmt *stmthp;
        OCIServer *srvhp;
        OCIDuration dur;
        OCISession *sesshp;
        oratext *username;
        oratext *password;
} test_ctx;

/* Helper function 1: execute a sql statement which binds xml data */
static sword exec_bind_xml(OCISvcCtx *svchp,
                           OCIError *errhp,
                           OCIStmt *stmthp,
                           void *xml,
                           OCIType *xmltdo,
                           OraText *sqlstmt);

/* Helper function 2: Initialize OCI handles and connect */
static sword init_oci_handles(test_ctx *ctx);

/* Helper function 3: Free OCI handles and disconnect */
static sword free_oci_handles(test_ctx *ctx);

void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  xmldocnode *doc = (xmldocnode *)0;
  ocixmldbparam params[1];
  xmlnode *quux, *foo, *foo_data, *top;
  xmlerr err;
  sword status = 0;
  xmlctx *xctx;
```

```
oratext ins_stmt[] = "insert into my_table values (:1)";
oratext tlpxml_test_sch[] = "<TOP/>";
ctx->username = (oratext *)"capiuser";
ctx->password = (oratext *)"***********"; /* Replace with real password */

/* Initialize envhp, svchp, errhp, dur, stmthp */
init_oci_handles(ctx);

/* Get an xml context */
params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
params[0].value_ocixmldbparam = &ctx->dur;
xctx = OCIXmlDbInitXmlCtx(ctx->envhp, ctx->svchp, ctx->errhp, params, 1);

/* Start processing - first, check that this DOM supports XML 1.0 */
printf("\n\nSupports XML 1.0? : %s\n",
       XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
       "YES" : "NO");

/* Parse a document */
if (!(doc = XmlLoadDom(xctx, &err, "buffer", tlpxml_test_sch,
                       "buffer_length", sizeof(tlpxml_test_sch)-1,
                       "validate", TRUE, NULL)))
{
  printf("Parse failed, code %d\n", err);
}
else
{
  /* Get the document element */
  top = (xmlnode *)XmlDomGetDocElem(xctx, doc);

  /* Print out the top element */
  printf("\n\nOriginal top element is :\n");
  XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

  /* Print out the document. The changes are reflected here */
  printf("\n\nOriginal document is :\n");
  XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

  /* Create some elements and add them to the document */
  quux = (xmlnode *) XmlDomCreateElem(xctx ,doc, (oratext *) "QUUX");
  foo = (xmlnode *) XmlDomCreateElem(xctx, doc, (oratext *) "FOO");
  foo_data = (xmlnode *) XmlDomCreateText(xctx, doc, (oratext *) "data");
  foo_data = XmlDomAppendChild(xctx, (xmlnode *) foo, (xmlnode *) foo_data);
  foo = XmlDomAppendChild(xctx, quux, foo);
  quux = XmlDomAppendChild(xctx, top, quux);

  /* Print out the top element */
  printf("\n\nNow the top element is :\n");
  XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

  /* Print out the document.  The changes are reflected here */
  printf("\n\nNow the document is :\n");
  XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

  /* Insert the document into my_table */
  status = OCITypeByName(ctx->envhp, ctx->errhp, ctx->svchp,
```

```
                          (const text *) "SYS", (ub4) strlen((char *)"SYS"),
                          (const text *) "XMLTYPE",
                          (ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
                          (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                          (OCIType **) &xmltdo);
    if (status == OCI_SUCCESS)
    {
      exec_bind_xml(ctx->svchp, ctx->errhp, ctx->stmthp, (void *)doc, xmltdo,
                    ins_stmt);
    }
  }
  /* Free xml ctx */
  OCIXmlDbFreeXmlCtx(xctx);

  /* Free envhp, svchp, errhp, stmthp */
  free_oci_handles(ctx);
}
```

The output from compiling and running this C program is as follows:

```
Supports XML 1.0? : YES

Original top element is :
<TOP/>

Original document is :
<TOP/>

Now the top element is :
<TOP>
  <QUUX>
    <FOO>data</FOO>
  </QUUX>
</TOP>

Now the document is :
<TOP>
  <QUUX>
    <FOO>data</FOO>
  </QUUX>
</TOP>
```

This is the result of querying the constructed document in `my_table`:

```
SELECT * FROM my_table;

SYS_NC_ROWINFO$
---------------
<TOP>
  <QUUX>
    <FOO>data</FOO>
  </QUUX>
</TOP>

1 row selected.
```

Using the C API for XML with Binary XML

You can use the C API for XML to read or write XML data that is encoded as binary XML from or to Oracle XML DB. Doing so involves the usual read and write procedures.

Binary XML is a compact, XML Schema-aware encoding of XML data. You can use binary XML as a storage model for `XMLType` data in the database, but you can also use it for XML data located outside the database.

Binary XML data is XML Schema-aware, and it can use various encoding schemes, depending on your needs. In order to manipulate binary XML data, you must have both the data and this metadata about the relevant XML schemas and encodings.

For `XMLType` data stored in the database, this metadata is also stored in the database. However, depending on how your database and data are set up, the metadata might not be on the same server as the data it applies to. If this is the case, then, before you can read or write binary XML data from or to the database, you must carry out these steps:

1. Create a context instance for the metadata.
2. Associate this context with a data connection that you use to access binary XML data in the database. A data connection can be a dedicated connection (`OCISvcCtx`) or a connection pool (`OCICPool`).

Then, when your application needs to encode or decode binary XML data on the data connection, it automatically fetches the metadata needed for that. As is illustrated by Example 14-2, the overall sequence of actions is as follows:

1. Create the usual OCI handles for environment (`OCIEnv`), connection (`OCISvcCtx`), and error context (`OCIError`).
2. Create one or more metadata contexts, as needed. A metadata context is sometimes referred to as a metadata repository, and `OCIBinXMLReposCtx` is the OCI context data structure. You use `OCIBinXMLCreateReposCtxFromConn` to create a metadata context from a dedicated connection and `OCIBinXMLCreateReposCtxFromCPool` to create a context from a connection pool.
3. Associate the metadata context(s) with the binary XML data connection(s). You use `OCIBinXmlSetReposCtxForConn` to do this.
4. (Optional) If the XML data originated outside of the database, use `setPicklePreference` to specify that XML data to be sent to the database from now on is in binary XML format. This applies to a DOM document (`xmldomdoc`). If you do not specify binary XML, the data is stored as text (`CLOB`).
5. Use OCI libraries to read and write XML data from and to the database. Whenever it is needed for encoding or decoding binary XML documents, the necessary metadata is fetched automatically using the metadata context. Use the C DOM API for XML to operate on the XML data at the client level.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide*

Example 14-2 Using the C API for XML with Binary XML

```
. . .
/* Private types and constants */
#define SCHEMA        (OraText *)"SYS"
#define TYPE          (OraText *)"XMLTYPE"
#define USER          (OraText *)"oe"
#define USER_LEN      (ub2)(strlen((char *)USER))
#define PWD           (OraText *)"oe"
#define PWD_LEN       (ub2)(strlen((char *)PWD))
#define NUM_PARAMS    1
static void checkerr(OCIError *errhp, sword status);
static sword create_env(OraText *user, ub2 user_len, OraText *pwd, ub2 pwd_len,
                        OCIEnv **envhp, OCISvcCtx **svchp, OCIError **errhp);
static sword run_example(OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp,
                         OCIDuration dur);
static void cleanup(OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp);

int main (int argc, char *argv[])
{
  OCIEnv     *envhp;
  OCISvcCtx  *svchp;
  OCIError   *errhp;
  printf("*** Starting Binary XML Example program\n");
  if (create_env(USER, USER_LEN, PWD, PWD_LEN, &envhp, &svchp, &errhp))
    {
      printf("FAILED: create_env()\n");
      cleanup(envhp, svchp, errhp);
      return OCI_ERROR;
    }
  if (run_example(envhp, svchp, errhp, OCI_DURATION_SESSION))
    {
      printf("FAILED: run_example()\n");
      cleanup(envhp, svchp, errhp);
      return OCI_ERROR;
    }
  cleanup(envhp, svchp, errhp);
  printf ("*** Completed Binary XML example\n");
  return OCI_SUCCESS;
}

static sword create_env(OraText *user, ub2 user_len,
                        OraText *pwd,  ub2 pwd_len,
                        OCIEnv **envhp, OCISvcCtx **svchp, OCIError **errhp)
{
  sword       status;
  OCIServer  *srvhp;
  OCISession *usrp;
  OCICPool   *poolhp;
  OraText    *poolname;
  ub4         poolnamelen;
  OraText    *database =(OraText *)"";
  OCIBinXmlReposCtx *rctx;
  /* Create and initialize environment. Allocate error handle. */
  . . .
  if ((status = OCIConnectionPoolCreate((dvoid *)*envhp, (dvoid*)*errhp,
                                        (dvoid *)poolhp, &poolname,
                                        (sb4 *)&poolnamelen,
                                        (OraText *)0,
                                        (sb4) 0, 1, 10, 1,
                                        (OraText *)USER,
                                        (sb4) USER_LEN,
```

```
                                        (OraText *)PWD,
                                        (sb4) PWD_LEN,
                                        OCI_DEFAULT)) != OCI_SUCCESS)
    {
      printf ("OCIConnectionPoolCreate - Fail %d\n", status);
      return OCI_ERROR;
    }
  status = OCILogon2((OCIEnv *)*envhp, *errhp, svchp, (OraText *)USER,
                     (ub4)USER_LEN, (const oratext *)PWD, (ub4)PWD_LEN,
                     (const oratext *)poolname, poolnamelen, OCI_CPOOL);
  if (status)
    {
      printf ("OCILogon2 - Fail %d\n", status);
      return OCI_ERROR;
    }
  OCIBinXmlCreateReposCtxFromCPool(*envhp, poolhp, *errhp, &rctx);
  OCIBinXmlSetReposCtxForConn(*svchp, rctx);
  return OCI_SUCCESS;
}

static sword run_example(OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp,
                         OCIDuration dur)
{
  OCIType   *xmltdo = (OCIType *)0;
  OCIStmt   *stmthp;
  OCIDefine *defnp;
  xmldocnode *xmldoc = (xmldocnode *)0;
  ub4        xmlsize = 0;
  text      *selstmt = (text *)"SELECT doc FROM po_binxmltab";
  sword      status;
  struct xmlctx *xctx = (xmlctx *) 0;
  ocixmldbparam params[NUM_PARAMS];
  xmlerr xerr = (xmlerr) 0;
  /* Obtain type definition for XMLType. Allocate statement handle.
     Prepare SELECT statement. Define variable for XMLType. Execute statement. */
  . . .
  /* Construct xmlctx for using XML C API */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &dur;
  xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, NUM_PARAMS);
  /* Print result to local string */
  XmlSaveDom(xctx, &xerr, (xmlnode *)xmldoc, "stdio", stdout, NULL);
  /* Free instances */
  . . .
}
```

Related Topics

- XMLType Storage Models
  `XMLType` is an *abstract* data type that provides different *storage models* to best fit your data and your use of it. As an abstract data type, your applications and database queries gain in flexibility: the same interface is available for all `XMLType` operations.

Using the Oracle XML Developer's Kit Pull Parser with Oracle XML DB

You can use the Oracle XML Developer's Kit pull parser with `XMLType` instances in Oracle XML DB. When you use this parser, parsing is done on demand, so your application drives the parsing process.

Your application accesses an XML document through a sequence of events, with start tags, end tags, and comments, just as in Simple API for XML (SAX) parsing. However, unlike the case of SAX parsing, where parsing events are handled by callbacks, in pull parsing your application calls methods to ask for (pull) events only when it needs them. This gives the application more control over XML processing. In particular, filtering is more flexible with the pull parser than with the SAX parser.

You can also use the Oracle XML Developer's Kit pull parser to perform stream-based XML Schema validation.

Example 14-3 shows how to use the Oracle XML DB pull parser with an `XMLType` instance. To use the pull parser, you also need static library `libxml10.a` on UNIX and Linux systems or `oraxml10.dll` on Microsoft Windows systems. You also need header file `xmlev.h`.

See Also:

- *Oracle XML Developer's Kit Programmer's Guide* for information about the Oracle XML Developer's Kit pull parser
- *Oracle XML Developer's Kit Programmer's Guide* for information on using the pull parser for stream-based validation

Example 14-3 Using the Oracle XML DB Pull Parser

```
#define MAXBUFLEN 64*1024
void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  ocixmldbparam params[1];
  sword status = 0;
  xmlctx *xctx;
  OCIDefine *defnp = (OCIDefine *) 0;
  oratext sel_stmt[] =
    "SELECT XMLSerialize(DOCUMENT x.OBJECT_VALUE AS CLOB) FROM PURCHASEORDER x where rownum = 1";
  OCILobLocator *cob;
  ub4 amtp, nbytes;
  ub1 bufp[MAXBUFLEN];
  ctx->username = (oratext *)"oe";
  ctx->password = (oratext *)"*************";    /* Replace with real password */

  /* Initialize envhp, svchp, errhp, dur, stmthp */
  init_oci_handles(ctx);

  /* Get an xml context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
```

```
  params[0].value_ocixmldbparam = &ctx->dur;
  xctx = OCIXmlDbInitXmlCtx(ctx->envhp, ctx->svchp, ctx->errhp, params, 1);

  /* Start processing */
  printf("\n\nSupports XML 1.0? : %s\n",
         XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
         "YES" : "NO");

  /* Allocate the lob descriptor */
  status = OCIDescriptorAlloc((dvoid *) ctx->envhp, (dvoid **) &clob,
                       (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0);
  if (status)
  {
    printf("OCIDescriptorAlloc Failed\n");
    goto error;
  }
  status = OCIStmtPrepare(ctx->stmthp, ctx->errhp,
                 (CONST OraText *)sel_stmt, (ub4) strlen((char *)sel_stmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
  if (status)
  {
    printf("OCIStmtPrepare Failed\n");
    goto error;
  }
  status = OCIDefineByPos(ctx->stmthp, &defnp, ctx->errhp, (ub4) 1,
                 (dvoid *) &clob, (sb4) -1, (ub2 ) SQLT_CLOB,
                 (dvoid *) 0, (ub2 *)0,
                 (ub2 *)0, (ub4) OCI_DEFAULT);
  if (status)
  {
    printf("OCIDefineByPos Failed\n");
    goto error;
  }
  status = OCIStmtExecute(ctx->svchp, ctx->stmthp, ctx->errhp, (ub4) 1,
                          (ub4) 0, (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                          (ub4) OCI_DEFAULT);
  if (status)
  {
    printf("OCIStmtExecute Failed\n");
    goto error;
  }
  /* read the fetched value into a buffer */
  amtp = nbytes = MAXBUFLEN-1;
  status = OCILobRead(ctx->svchp, ctx->errhp, clob, &amtp,
                (ub4) 1, (dvoid *) bufp, (ub4) nbytes, (dvoid *)0,
                (sb4 (*)(dvoid *, CONST dvoid *, ub4, ub1)) 0,
                (ub2) 0, (ub1) SQLCS_IMPLICIT);
  if (status)
  {
    printf("OCILobRead Failed\n");
    goto error;
  }
  bufp[amtp] = '\0';
  if (amtp > 0)
  {
     printf("\n=> Query result of %s: \n%s\n", sel_stmt, bufp);
     /********** PULL PARSING ******************/
     status = pp_parse(xctx, bufp, amtp);
     if (status)
       printf("Pull Parsing failed\n");
  }
error:
```

```
  /* Free XML Ctx */
  OCIXmlDbFreeXmlCtx(xctx);

  /* Free envhp, svchp, errhp, stmthp */
  free_oci_handles(ctx);
}
#define ERRBUFLEN 256
sb4 pp_parse(xctx, buf, amt)
xmlctx  *xctx;
oratext *buf;
ub4     amt;
{
  xmlevctx *evctx;
  xmlerr   xerr = XMLERR_OK;
  oratext  message[ERRBUFLEN];
  oratext  *emsg = message;
  xmlerr   ecode;
  boolean  done, inattr = FALSE;
  xmlevtype event;

  /* Create an XML event context - Pull Parser Context */
  evctx = XmlEvCreatePPCtx(xctx, &xerr,
                           "expand_entities", FALSE,
                           "validate", TRUE,
                           "attr_events", TRUE,
                           "raw_buffer_len", 1024,
                           NULL);
 if (!evctx)
  {
    printf("FAILED: XmlEvCreatePPCtx: %d\n", xerr);
    return OCI_ERROR;
  }
  /* Load the document from input buffer */
  xerr = XmlEvLoadPPDoc(xctx, evctx, "buffer", buf, amt, "utf-8");
  if (xerr)
  {
    printf("FAILED: XmlEvLoadPPDoc: %d\n", xerr);
    return OCI_ERROR;
  }
  /* Process the events until END_DOCUMENT event or error */
  done = FALSE;
  while(!done)
  {
    event = XmlEvNext(evctx);
    switch(event)
    {
      case XML_EVENT_START_ELEMENT:
        printf("START ELEMENT: %s\n", XmlEvGetName0(evctx));
        break;
      case XML_EVENT_END_ELEMENT:
        printf("END ELEMENT: %s\n", XmlEvGetName0(evctx));
        break;
      case XML_EVENT_START_DOCUMENT:
        printf("START DOCUMENT\n");
        break;
      case XML_EVENT_END_DOCUMENT:
        printf("END DOCUMENT\n");
        done = TRUE;
        break;
      case XML_EVENT_START_ATTR:
        printf("START ATTRIBUTE: %s\n", XmlEvGetAttrName0(evctx, 0));
        inattr = TRUE;
```

```
        break;
      case XML_EVENT_END_ATTR:
        printf("END ATTRIBUTE: %s\n", XmlEvGetAttrName0(evctx, 0));
        inattr = FALSE;
        break;
      case XML_EVENT_CHARACTERS:
        if (inattr)
          printf("ATTR VALUE: %s\n", XmlEvGetText0(evctx));
        else
          printf("TEXT: %s\n", XmlEvGetText0(evctx));
        break;
      case XML_EVENT_ERROR:
      case XML_EVENT_FATAL_ERROR:
        done = TRUE;
        ecode = XmlEvGetError(evctx, &emsg);
        printf("ERROR: %d: %s\n", ecode, emsg);
        break;
    }
  }
  /* Destroy the event context */
  XmlEvDestroyPPCtx(xctx, evctx);
  return OCI_SUCCESS;
}
```

The output from compiling and running this C program is as follows:

```
=> Query result of XMLSerialize(DOCUMENT x.OBJECT_VALUE AS CLOB) FROM PURCHASEORDER x where rownum = 1:

<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:noNamespaceSchemaLocation=
                 "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd">
  <Reference>AMCEWEN-20021009123336171PDT</Reference>
  <Actions>
    <Action>
      <User>KPARTNER</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Allan D. McEwen</Requestor>
  <User>AMCEWEN</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Allan D. McEwen</name>
    <address>Oracle Plaza
Twin Dolphin Drive
Redwood Shores
CA
94065
USA</address>
    <telephone>650 506 7700</telephone>
  </ShippingInstructions>
  <SpecialInstructions>Ground</SpecialInstructions>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>Salesperson</Description>
      <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    . . .
  </LineItems>
</PurchaseOrder>

START DOCUMENT
START ELEMENT: PurchaseOrder
```

```
START ATTRIBUTE: xmlns:xsi
ATTR VALUE: http://www.w3.org/2001/XMLSchema-instance
END ATTRIBUTE: xmlns:xsi
START ATTRIBUTE: xsi:noNamespaceSchemaLocation
ATTR VALUE: http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd
END ATTRIBUTE: xsi:noNamespaceSchemaLocation
START ELEMENT: Reference
TEXT: AMCEWEN-20021009123336171PDT
END ELEMENT: Reference
START ELEMENT: Actions
START ELEMENT: Action
START ELEMENT: User
TEXT: KPARTNER
END ELEMENT: User
END ELEMENT: Action
END ELEMENT: Actions
START ELEMENT: Reject
END ELEMENT: Reject
START ELEMENT: Requestor
TEXT: Allan D. McEwen
END ELEMENT: Requestor
START ELEMENT: User
TEXT: AMCEWEN
END ELEMENT: User
START ELEMENT: CostCenter
TEXT: S30
END ELEMENT: CostCenter
START ELEMENT: ShippingInstructions
START ELEMENT: name
TEXT: Allan D. McEwen
END ELEMENT: name
START ELEMENT: address
TEXT: Oracle Plaza
Twin Dolphin Drive
Redwood Shores
CA
94065
USA
END ELEMENT: address
START ELEMENT: telephone
TEXT: 650 506 7700
END ELEMENT: telephone
END ELEMENT: ShippingInstructions
START ELEMENT: SpecialInstructions
TEXT: Ground
END ELEMENT: SpecialInstructions
START ELEMENT: LineItems
START ELEMENT: LineItem
START ATTRIBUTE: ItemNumber
ATTR VALUE: 1
END ATTRIBUTE: ItemNumber
START ELEMENT: Description
TEXT: Salesperson
END ELEMENT: Description
START ELEMENT: Part
START ATTRIBUTE: Id
ATTR VALUE: 37429158920
END ATTRIBUTE: Id
START ATTRIBUTE: UnitPrice
ATTR VALUE: 39.95
END ATTRIBUTE: UnitPrice
START ATTRIBUTE: Quantity
```

```
ATTR VALUE: 2
END ATTRIBUTE: Quantity
END ELEMENT: Part
END ELEMENT: LineItem
. . .
END ELEMENT: LineItems
END ELEMENT: PurchaseOrder
END DOCUMENT
```

Common XMLType Operations in C

Common XML operations are provided by the C API for XML.

Table 14-2 provides the `XMLType` functional equivalent of common XML operations.

Table 14-2 Common XMLType Operations in C

Description	C API XMLType Function
Create empty `XMLType` instance	`XmlCreateDocument()`
Create from a source buffer	`XmlLoadDom()`
Extract an XPath expression	`XmlXPathEvalexpr()` and family
Transform using an XSLT stylesheet	`XmlXslProcess()` and family
Check if an XPath exists	`XmlXPathEvalexpr()` and family
Is document schema-based?	`XmlDomIsSchemaBased()`
Get schema information	`XmlDomGetSchema()`
Get document namespace	`XmlDomGetNodeURI()`
Validate using schema	`XmlSchemaValidate()`
Obtain DOM from `XMLType`	Cast `(void *)` to `(xmldocnode *)`
Obtain XMLType from DOM	Cast `(xmldocnode *)` to `(void *)`

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide* "XML Parser for C"

Example 14-4 shows how to use the DOM to determine how many instances of a particular part have been ordered. The part in question has `Id 37429158722`. See Oracle-Supplied XML Schemas and Examples, Example A-6 for the definitions of helper functions `exec_bind_xml`, `free_oci_handles`, and `init_oci_handles`.

Example 14-4 Using the DOM to Count Ordered Parts

```
#ifndef S_ORACLE
#endif
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif
#ifndef XML_ORACLE
#include <xml.h>
```

```
#endif
#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <string.h>

typedef struct test_ctx {
        OCIEnv *envhp;
        OCIError *errhp;
        OCISvcCtx *svchp;
        OCIStmt *stmthp;
        OCIServer *srvhp;
        OCIDuration dur;
        OCISession *sesshp;
        oratext *username;
        oratext *password;
} test_ctx;

/* Helper function 1: execute a sql statement which binds xml data */
static sword exec_bind_xml(OCISvcCtx *svchp,
                           OCIError *errhp,
                           OCIStmt *stmthp,
                           void *xml,
                           OCIType *xmltdo,
                           OraText *sqlstmt);

/* Helper function 2: Initialize OCI handles and connect */
static sword init_oci_handles(test_ctx *ctx);

/* Helper function 3: Free OCI handles and disconnect */
static sword free_oci_handles(test_ctx *ctx);

void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  xmldocnode *doc = (xmldocnode *)0;
  ocixmldbparam params[1];
  xmlnode *quux, *foo, *foo_data, *top;
  xmlerr err;
  sword status = 0;
  xmlctx *xctx;
  ub4 xmlsize = 0;
  OCIDefine *defnp = (OCIDefine *) 0;
  oratext sel_stmt[] = "SELECT SYS_NC_ROWINFO$ FROM PURCHASEORDER";
  xmlnodelist *litems = (xmlnodelist *)0;
  xmlnode *item = (xmlnode *)item;
  xmlnode *part;
  xmlnamedmap *attrs;
  xmlnode *id;
  xmlnode *qty;
  oratext *idval;
```

```
oratext *qtyval;
ub4 total_qty;
int i;
int numdocs;

ctx->username = (oratext *)"oe";
ctx->password = (oratext *)"************";   /* Replace with real password */

/* Initialize envhp, svchp, errhp, dur, stmthp */
init_oci_handles(ctx);

/* Get an xml context */
params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
params[0].value_ocixmldbparam = &ctx->dur;
xctx = OCIXmlDbInitXmlCtx(ctx->envhp, ctx->svchp, ctx->errhp, params, 1);

/* Start processing */
printf("\n\nSupports XML 1.0? : %s\n",
       XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
       "YES" : "NO");

/* Get the documents from the database using a select statement */
status = OCITypeByName(ctx->envhp, ctx->errhp, ctx->svchp, (const text *) "SYS",
                       (ub4) strlen((char *)"SYS"), (const text *) "XMLTYPE",
                       (ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
                       (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                       (OCIType **) &xmltdo);
status = OCIStmtPrepare(ctx->stmthp, ctx->errhp,
               (CONST OraText *)sel_stmt, (ub4) strlen((char *)sel_stmt),
               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
status = OCIDefineByPos(ctx->stmthp, &defnp, ctx->errhp, (ub4) 1, (dvoid *) 0,
               (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
               (ub2 *)0, (ub4) OCI_DEFAULT);
status = OCIDefineObject(defnp, ctx->errhp, (OCIType *) xmltdo,
                (dvoid **) &doc,
                &xmlsize, (dvoid **) 0, (ub4 *) 0);
status = OCIStmtExecute(ctx->svchp, ctx->stmthp, ctx->errhp, (ub4) 0, (ub4) 0,
               (CONST OCISnapshot*) 0, (OCISnapshot*) 0, (ub4) OCI_DEFAULT);

/* Initialize variables */
total_qty = 0;
numdocs = 0;

/* Loop through all the documents */
while ((status = OCIStmtFetch2(ctx->stmthp, ctx->errhp, (ub4) 1, (ub4) OCI_FETCH_NEXT,
                               (ub4)1, (ub4) OCI_DEFAULT)) == 0)
{
  numdocs++;

  /* Get all the LineItem elements */
  litems = XmlDomGetDocElemsByTag(xctx, doc, (oratext *)"LineItem");
  i = 0;

  /* Loop through all LineItems */
  while (item = XmlDomGetNodeListItem(xctx, litems, i))
  {
```

```
      /* Get the part */
      part = XmlDomGetLastChild(xctx, item);

      /* Get the attributes */
      attrs = XmlDomGetAttrs(xctx, (xmlelemnode *)part);

      /* Get the id attribute and its value */
      id = XmlDomGetNamedItem(xctx, attrs, (oratext *)"Id");
      idval = XmlDomGetNodeValue(xctx, id);

      /* Keep only parts with id 37429158722 */
      if (idval && (strlen((char *)idval) == 11 )
          && !strncmp((char *)idval, (char *)"37429158722", 11))
      {
        /* Get the quantity attribute and its value.*/
        qty = XmlDomGetNamedItem(xctx, attrs, (oratext *)"Quantity");
        qtyval = XmlDomGetNodeValue(xctx, qty);

        /* Add the quantity to total_qty */
        total_qty += atoi((char *)qtyval);
      }
      i++;
    }
    XmlFreeDocument(xctx, doc);
    doc = (xmldocnode *)0;
  }
  printf("Total quantity needed for part 37429158722 = %d\n", total_qty);
  printf("Number of documents in table PURCHASEORDER = %d\n", numdocs);

  /* Free Xml Ctx */
  OCIXmlDbFreeXmlCtx(xctx);

  /* Free envhp, svchp, errhp, stmthp */
  free_oci_handles(ctx);
}
```

The output from compiling and running this C program is as follows:

```
Supports XML 1.0? : YES
Total quantity needed for part 37429158722 = 42
Number of documents in table PURCHASEORDER = 132
```