RHIC/AP/77

A Reference Manual for the DB++ Class Library

Steven Tepikian
November, 1995

INTRODUCTION

DB++ is a collection of C++ classes for database access through a SYBASE database server. Using these classes allows you to access the database with only a few lines of code. This is illustrated with an example shown on the last page of this technote. The user first finds out all the user tables in a particular database and writes out an SQL "create table" script for each table. Along with this example are the manual pages (which are also on line) describing most of the classes in DB++. Future versions will build on this functionality and retain the original methods as much as possible leading to virtually no changes to ones code as DB++ is upgraded.

DB++(3)

NAME

DB++ - An introduction to the DB++ Classes

SYNOPSIS

#include <DB++/db_base.H>

DESCRIPTION

A collection of C++ classes for accessing, updating and creating tables in a *SYBASE* database. These classes contain various constructors and methods for:

(1) Initializing access to a *SYBASE* database,
(2) Connecting as a given user to a server,
(3) Changing to a given database,
(4) Querying the database with *SQL* commands and storing the result in tables in program memory.
(5) Copying the contents of a table and storing the results in program memory.
(6) Truncating the table in the database and transfering the contents of data from program memory to the database table.

CLASSES

DB_init

An object of this class initializes the *SYBASE* Client-Library for accessing *SYBASE* servers. Properties can be read, set or cleared using the available methods. The output stream or file for where *SYBASE* or *DB++* messages are sent can be changed through this class. The *DB_messages* class is transparently used for this purpose.

DB_user

Login information such as user's name and password are stored in this object. A connection to a given server using the login information can be open. Request and response capabilities can be obtained or changed, as well as properties and options.

DB_command

Includes methods to submit an *SQL* queries and command to the server. The results (if any) are stored program memory as tables. Tables can be read or written directly using bulk copy for more efficient access. Access to the *DB_tables* objects is provided for the various tables stored in memory using a subscript operator. Also, some of the properties can be changed.

DB_table

This object stores the table structure and the data using *DB_column* objects for each column in a table. Provides access to the *DB_column* objects using a subscript operator.

DB_column

Stores the data and information about the structure of the column. Provides access to the data using the *DB_primitive* objects with the subscript operator.

DB_primitive

Is the base class for various data types such as *DB_double, DB_string,* etc. that holds individual data items.

DB_messages

Holds the output stream for messages from the classes as well as the server. The default output stream goes to the file $DBAPPLACE/dbmessages which can be changed.

DB_array

This is an array of pointers which keeps track of the *DB_table* objects, *DB_column* objects and *DB_primitive* data objects. The *DB_array* objects along with the various constructors and destructors handles the memory management.

FILES

The various files for accessing the libraries and header files can be found in the directory given in the enviromental variable *DBAPPLACE.*

The header files needed are (Note, you only need to include *DB++/db_base.H,* all the others are

automatically included):

$DBAPPLACE/include/DB++/db_base.H
$DBAPPLACE/include/DB++/dbprimitive.H
$DBAPPLACE/include/DB++/db_array.H
$SYBASE/include/bkpublic.h
$SYBASE/include/csconfig.h
$SYBASE/include/cspublic.h
$SYBASE/include/cstypes.h
$SYBASE/include/ctpublic.h

The source files are:

$DBAPPLACE/DB++/theLib/db_base.C
$DBAPPLACE/DB++/theLib/dbprimitive.C
$DBAPPLACE/DB++/theLib/db_array.C

Example files of using these libraries

$DBAPPLACE/DB++/example/testdb.C
$DBAPPLACE/DB++/example/testdb1.C
$DBAPPLACE/DB++/example/dbread.C
$DBAPPLACE/DB++/blkcpy/blkcpy.C
$DBAPPLACE/DB++/querydb/querydb.C
$DBAPPLACE/bin/$ARCH/blkcpy
$DBAPPLACE/bin/$ARCH/querydb

Other files that are used:

$DBAPPLACE/lib/$ARCH/libdb++.a
$DBAPPLACE/dbmessages
$DBAPPLACE/.dblogins
$HOME/.dblogins
$SYBASE/lib/libcs.so
$SYBASE/lib/libct.so
$SYBASE/lib/libcomn.so
$SYBASE/lib/libblk.so
$SYBASE/lib/libintl.so
$SYBASE/lib/libtcl.so

SEE ALSO

DB_init(3), DB_user(3), DB_command(3), DB_table(3), DB_column(3), DB_primitive(3), DB_properties(3), DB_options(3), DB_capabilities(3)

CAVEATS

The C++ Database classes can not tell the difference between the *SYBASE* data types of *char* and *varchar* or between the data types of *binary* and *varbinary*. Indexing information is lost when the table is recalled from *SYBASE*. Additional information, such as setting special default values is not transfered to the tables in memory.

REFERENCES

SYBASE: "Open Client Client-Library/C Programmers Guide"
SYBASE: "Open Client Client-Library/C Reference Manual"
SYBASE: "Open Client/Server Quick Reference Guide"
SYBASE: "Open Client and Open Server Common Libraries Reference Manual"

DB_capabilities(3)

NAME

DB_capabilities - Response and Request capabilities of the connection.

SYNOPSIS

#include <DB++/db_base.H>

DESCRIPTION

This is a list of request and response capabilities for accessing and sending information to the server. Note, request capabilities cannot be changed. These are taken from the tables "CS_CAP_REQUEST Capabilities" and "CS_CAP_RESPONSE Capabilities" in "Open Client Client-Library/C Reference Manual", starting on page 3-33.

REQUEST

CS_CON_INBAND: In-band (non-expedited) attentions.
CS_CON_OOB: Out-of-band (expedited) attentions.
CS_CSR_ABS: Fetch of specified absolute cursor row.
CS_CSR_FIRST: Fetch of first cursor row.
CS_CSR_LAST: Fetch of last cursor row.
CS_CSR_MULTI: Multi-row cursor fetch.
CS_CSR_PREV: Fetch previous cursor row.
CS_CSR_REL: Fetch specified relative cursor row.
CS_DATA_BIN: Binary datatype.
CS_DATA_VBIN: Variable-length binary datatype.
CS_DATA_LBIN: Long binary datatype.
CS_DATA_BIT: Bit Datatype.
CS_DATA_BITN: Nullable bit values.
CS_DATA_BOUNDARY: Secure Server boundary datatypes.
CS_DATA_CHAR: Character datatype.
CS_DATA_VCHAR: Variable-length character datatype.
CS_DATA_LCHAR: Long character datatype.
CS_DATA_DATE4: Short datetime datatype.
CS_DATA_DATE8: Datetime datatype.
CS_DATA_DATETIMEN: NULL datetime values.
CS_DATA_DEC: Decimal datatype.
CS_DATA_FLT4: 4-byte float datatype (float).
CS_DATA_FLT8: 8-byte float datatype (double).
CS_DATA_FLTN: Nullable float values.
CS_DATA_IMAGE: Image datatype.
CS_DATA_INT1: Tiny integer datatype (1-byte).
CS_DATA_INT2: Small integer datatype (2-byte).
CS_DATA_INT4: Integer datatype (4-byte).
CS_DATA_INTN: NULL integer values.
CS_DATA_MNY4: Short money datatype.
CS_DATA_MNY8: Money datatype.
CS_DATA_MONEYN: NULL money values.
CS_DATA_NUM: Numeric datatypes.
CS_DATA_SENSITIVITY: Secure Server sensitivity datatypes.
CS_DATA_TEXT: Text datatype.
CS_OPTION_GET: Are option values from server obtainable.
CS_PROTO_BULK: Tokenized bulk copy.
CS_PROTO_DYNAMIC: Descriptions for prepared statements come back at prepare time.
CS_PROTO_DYNPROC: Client-Library prepends SQL to a Dynamic SQL prepare statement.
CS_REQ_BCP: Bulk copy requests.
CS_REQ_CURSOR: Cursor requests.

CS_REQ_DYN: Dynamic SQL requests.
CS_REQ_LANG: Language requests.
CS_REQ_MSG: Message commands.
CS_REQ_MSTMT: Multiple server commands per Client-Library language command.
CS_REQ_NOTIF: Registered procedure notifications.
CS_REQ_PARAM: Use PARAM/PARAMFMT TDS streams for requests.
CS_REQ_URGNOTIF: Send notifications with the "urgent" bit set in the TDS packet header.
CS_REQ_RPC: Remote procedure requests.

RESPONSE

CS_CON_NOINBAND: No in-band (non-expedited) attentions.
CS_CON_NOOOB: No out-of-band (expedited) attentions.
CS_DATA_NOBIN: No binary datatype.
CS_DATA_NOVBIN: No variable-length binary datatype.
CS_DATA_NOLBIN: No long binary datatype.
CS_DATA_NOBIT: No bit Datatype.
CS_DATA_NOBOUNDARY: No Secure Server boundary datatypes.
CS_DATA_NOCHAR: No character datatype.
CS_DATA_NOVCHAR: No variable-length character datatype.
CS_DATA_NOLCHAR: No long character datatype.
CS_DATA_NODATE4: No short datetime datatype.
CS_DATA_NODATE8: No datetime datatype.
CS_DATA_NODATETIMEN: No NULL datetime values.
CS_DATA_NODEC: No decimal datatype.
CS_DATA_NOFLT4: No 4-byte float datatype (float).
CS_DATA_NOFLT8: No 8-byte float datatype (double).
CS_DATA_NOIMAGE: No image datatype.
CS_DATA_NOINT1: No tiny integer datatype (1-byte).
CS_DATA_NOINT2: No small integer datatype (2-byte).
CS_DATA_NOINT4: No integer datatype (4-byte).
CS_DATA_NOINT8: No 64 bit integer datatype (8-byte).
CS_DATA_NOINTN: No NULL integer values.
CS_DATA_NOMNY4: No short money datatype.
CS_DATA_NOMNY8: No money datatype.
CS_DATA_NOMONEYN: No NULL money values.
CS_DATA_NONUM: No numeric datatypes.
CS_DATA_NOSENSITIVITY: No Secure Server sensitivity datatypes.
CS_DATA_NOTEXT: No text datatype.
CS_RES_NOEED: No extended error results.
CS_RES_NOMSG: No message results.
CS_RES_NOPARAM: Don't use PARAM/PARAMFMT TDS streams for requests.
CS_RES_NOSTRIPBLANKS: The server shouldn't strip blanks when returning data from nullable fixed length character columns.
CS_RES_NOTDSDEBUG: No TDS debug token in response to certain "dbcc" commands.

SEE ALSO

DB_user(3)

REFERENCES

SYBASE: "Open Client Client-Library/C Reference Manual"

NAME

DB_column - The C++ class for working with columns stored in program memory.

SYNOPSIS

```
#include <DB++/db_base.H>

class DB_column
{
public:
  // Retrieving the data.
  DB_primitive &      operator[] (CS_INT i);
  const DB_primitive & operator[] (CS_INT i) const;

  // Properties of the data column.
  DB_string         name() const;
  CS_BOOL             isColumn(const char *nn) const;
  CS_BOOL             canBeNull() const;
  CS_BOOL             isHidden() const;
  CS_BOOL             isIdentity() const;
  CS_BOOL             isKey() const;
  CS_BOOL             isUpdatable() const;
  CS_BOOL             isVersionKey() const;
  CS_BOOL             isTimeStamp() const;
  CS_BOOL             isUpdateCol() const;
  CS_BOOL             isInputValue() const;
  CS_BOOL             isReturn() const;
  CS_INT            numRows() const;
  DB_types          type() const;
  CS_INT            cstype() const;
  CS_INT            width() const;
  CS_BOOL             areTheirNulls() const;

  // Copying the column data to array variables.
  CS_INT            colcpy(CS_CHAR **&vec) const;
  CS_INT            colcpy(CS_BYTE **&vec) const;
  CS_INT            colcpy(long *&vec) const;
  CS_INT            colcpy(CS_FLOAT *&vec) const;
  CS_INT            colcpy(CS_BYTE *&vec) const;
  CS_INT            colcpy(CS_REAL *&vec) const;
  CS_INT            colcpy(CS_SMALLINT *&vec) const;

  DB_column() { }
  ~DB_column();
};
```

DESCRIPTION

This object holds structure information and provides storage and access to the data items for a given column.

DATA HANDLING

To access an individual data item, use the subscript operator for a given row number starting from zero. If the row number is out of bounds then then a *DB_void* data object is returned.

```
DB_primitive &      operator[] (CS_INT i);
const DB_primitive & operator[] (CS_INT i) const;
```

STATUS

```
DB_string       name() const;
CS_INT          numRows() const;

CS_BOOL           isColumn(const char *nn) const;
CS_BOOL           canBeNull() const;
CS_BOOL           isHidden() const;
CS_BOOL           isIdentity() const;
CS_BOOL           isKey() const;
CS_BOOL           isUpdatable() const;
CS_BOOL           isVersionKey() const;
CS_BOOL           isTimeStamp() const;
CS_BOOL           isUpdateCol() const;
CS_BOOL           isInputValue() const;
CS_BOOL           isReturn() const;
DB_types        type() const;
CS_INT          cstype() const;
CS_INT          width() const;
CS_BOOL           areTheirNulls() const;
```

From the above methods we can find out the name of the column, number of rows of data stored and the status (either CS_TRUE or CS_FALSE). Furthermore, we can find the type of column as a *DB_primitive* type or in terms of the *Client-Server SYBASE* type and its width. Finally, we can find out whether some of the data members are NULL.

COPYING TO ARRAYS

Given an empty pointer, the following methods will allocate memory to the pointer through the C++ new command and store the column data into the array. The member function returns the number of array elements created. Be sure to apply delete to these arrays after you are finished in order to avoid memory leaks.

```
CS_INT          colcpy(CS_CHAR **&vec) const;
CS_INT          colcpy(CS_BYTE **&vec) const;
CS_INT          colcpy(long *&vec) const;
CS_INT          colcpy(CS_FLOAT *&vec) const;
CS_INT          colcpy(CS_BYTE *&vec) const;
CS_INT          colcpy(CS_REAL *&vec) const;
CS_INT          colcpy(CS_SMALLINT *&vec) const;
```

SEE ALSO

DB++(3), DB_table(3), DB_primitive(3)

NAME

DB_command - The C++ class to send and obtain data from the database.

SYNOPSIS

```
#include <DB++/db_base.H>

class DB_command
{
public:
  // Open up a process to the database for access.
  DB_command(DB_user &srvr);

  // Close the process to the database and free all tables memory.
  ~DB_command();

  // Setting properties.
  CS_RETCODE setProp(CS_INT property, CS_INT result);
  CS_RETCODE setProp(CS_INT property, char *result);

  // Getting properties.
  CS_RETCODE getProp(CS_INT property, CS_INT &result);
  CS_RETCODE getProp(CS_INT property, char *result, CS_INT buffsize);

  // Clearing properties,
  CS_RETCODE clearProp(CS_INT property);

  // Set to a given database.
  CS_RETCODE setDatabase(const char *name);

  // Working with SQL commands.
  void      appendSQL(const char *cmd);
  void      appendQuoteSQL(const char *cmd);
  void      clearSQL();
  void      execSQL(const char *nme, CS_INT &cmd_fails, CS_INT &cmds);
  const CS_CHAR* sqlCommand();

  // Bulk copy access.
  void      setMultRows(CS_INT numRF);
  CS_RETCODE bulkGetTable(const char *tname);
  CS_RETCODE bulkStoreTable(const char *tname, CS_INT &out_row);
  CS_RETCODE bulkStoreTable(const char *tname, CS_INT occur, CS_INT &out_row);

  // Dealing with tables on the database.
  CS_RETCODE describeTable(const char *nme);
  CS_RETCODE getTable(const char *nme);
  CS_RETCODE truncateTable(const char *nme);

  // Working with a Table in memory.
  CS_INT    numTableMemory(const char *table_name);
  CS_INT    numTableMemory();
  void      removeLastTable();

  // Accessing a table in memory.
  DB_table&  operator [] (CS_INT i);
  DB_table&  operator () (const char *tname, CS_INT occur = 1);
```

```
  // Using the database for conversion of data to and from an ascii format.
  CS_RETCODE store(DB_primitive &dt, const CS_CHAR *str);
  CS_RETCODE recall(const DB_primitive &dt, CS_CHAR *str, CS_INT size);
};
```

DESCRIPTION

Once the connection is established using *DB_user,* the database can be opened with a *DB_command* object. Through the various methods available, one could query the database for the data it contains or read a table directly. You don't have to worry about the structure of the data returned from the query, this is all taken care of for you. Additionally, you can retrieve or write data to tables using bulk copy.

CONSTRUCTOR

The constructor requires a *DB_user* object after a connection to the database server is established.

PROPERTIES

Some of the properties can be changed (note, their are some properties that can only be changed with *DB_init* objects or *DB_user* objects) through the following methods:

```
CS_RETCODE setProp(CS_INT property, CS_INT result);
CS_RETCODE setProp(CS_INT property, char *result);
```

where char strings are assumed to be NULL terminated. For example to turn on ANSI binds we use the following code segment:

```
DB_init syb;
DB_user lgn(syb);

// Connecting to server ....

DB_command tbl(lgn);

// Some more stuff ....

if (tbl.setProp(CS_ANSI_BINDS, CS_TRUE) != CS_SUCCEED) {
  // Handle the error ...
}

// And continue ...
```

To find out what a property is set to, use the following methods:

```
CS_RETCODE getProp(CS_INT property, CS_INT &result);
CS_RETCODE getProp(CS_INT property, char *result, CS_INT buffsize);
```

where buffsize is the amount of space allocated to the char *result in the second getProp method. An example of getting the name of the declared cursor:

```
char cname[50];

// Some more stuff ....

DB_init syb;
DB_user lgn(syb);

// Connecting to server ....

DB_command tbl(lgn);
```

```
// Some more stuff ....

if (tbl.getProp(CS_CUR_NAME, cname, 50) != CS_SUCCEED) {
  // Handle the error ...
}

// And continue ...
```

Properties can be cleared, returning to the default values with:

```
CS_RETCODE clearProp(CS_INT property);
```

CHANGING DATABASE

The database that one is accessing can be changed with the following method.

```
CS_RETCODE setDatabase(const char *name);
```

SQL COMMANDS

SQL quries or commands can be sent to the server with the following methods:

```
void     appendSQL(const char *cmd);
void     appendQuoteSQL(const char *cmd);
void     clearSQL();
void     execSQL(const char *nme, CS_INT &cmd_fails, CS_INT &cmds);
const CS_CHAR* sqlCommand();
```

The first method, appendSQL, appends the command - given as the argument - to a buffer in program memory.

The method appendQuoteSQL is similar to appendSQL except that the command string is enclosed in double quotes before it is appended.

The third method, clearSQL, clears the command buffer.

The fourth command, exeSQL, sends the command to the server, but does not clear the command buffer and fetches data if any. The first argument is the name of tables that are stored in memory if data is returned. The second argument returns the number of failed returns of result sets. The final argument, returns the total number of results sets. If no data is returned, the first argument is ignored.

The last method, sqlCommand, will return the pointer to the sql command string stored in the buffer.

BULK COPY

To send and receive tables efficiently from the server, bulk copy is available. The methods are listed below. Note, you must enable bulk copy in the *DB_user* object before establishing the connection and creating this *DB_command* object. Furthermore, the the database itself must have the bulk copy option set.

```
void     setMultRows(CS_INT numRF);
CS_RETCODE bulkGetTable(const char *tname);
CS_RETCODE bulkStoreTable(const char *tname, CS_INT &out_row);
CS_RETCODE bulkStoreTable(const char *tname, CS_INT occur, CS_INT &out_row);
```

The method, setMultRows, sets the number of rows and the buffer size the program uses to send or receive data per transfer. Nominally, this is set to 100. Larger numbers may result in better performance, but would require a larger buffer. Note, the memory for the buffer is allocated before the transfer begins and is deallocated after the transfer is completed, this applies equally well to transfers of data using *SQL* commands.

The next method, bulkGetTable, will receive a table from the server named according to the argument and store it in program memory.

Finally, the last two methods, bulkStoreTable, will store the occurance, "occur", of table "tname" in program memory to a database table named "tname" with the "out_rows" argument returning the number of rows stored. If "occur" is not included, it is the first occurance.

DATABASE TABLES

These methods are used to act on tables that are in the database. Note, tables are stored in the order they are created.

```
CS_RETCODE describeTable(const char *nme);
CS_RETCODE getTable(const char *nme);
CS_RETCODE truncateTable(const char *nme);
```

The first method, describeTable, will create an empty table in program memory with the column descriptions such as the column names, width, etc. Note, this method creates a temporary procedure in the "tempdb" database in order to get the table description.

The next method, getTable, will copy the entire copy of the table from the database to program memory. If bulk copy is enabled, then bulk copy methods will be used.

The method truncateTable, will empty a table in the database but leave the structure. It is not a good idea to drop the table and recreate its structure since, not all the structures such as indexing, data default values, etc. can be recreated from the *DB++* objects.

TABLES IN MEMORY

The number of tables stored in memory is only limited by the available program memory. Furthermore, two or more tables in program memory can have the same name. The following methods help in dealing with the tables in memory.

```
CS_INT    numTableMemory(const char *table_name);
CS_INT    numTableMemory();
void      removeLastTable();
```

Use, numTableMemory("table_name"), to find out how many tables in memory have the name "table_name".

To find out the total number of tables in memory use, numTableMemory without arguments.

Use the method, removeLastTable, to remove the last table from memory.

To access the *DB_table* object directly use the following subscript operators.

```
DB_table&  operator [] (CS_INT i);
DB_table&  operator () (const char *tname, CS_INT occur = 1);
```

You can access tables by count starting from zero to the last table in the order in which they were entered into program memory. Or you can refer to the table by its name, and/or occurrence with the first occurrence by default. The occurrence numbers start from one (first).

CONVERTING DATA

The database conversion utilities can be used for converting some of the data formats to an ascii format in order to store the results in files. This conversion utility is needed if a special date-time format is used in the database. Since the *DB_primitive* objects do not have access to the database, this access must be provided through the *DB_command* object. The following methods achieve these goals.

```
CS_RETCODE store(DB_primitive &dt, const CS_CHAR *str);
CS_RETCODE recall(const DB_primitive &dt, CS_CHAR *str, CS_INT size);
```

SEE ALSO

DB++(3), DB_properties(3), DB_table(3), DB_user(3), DB_primitive(3)

NAME

DB_init - The C++ class to initialize access to the Database Libraries.

SYNOPSIS

```
#include <DB++/db_base.H>

class DB_init
{
public:
  // Setting properties.
  CS_RETCODE setProp(CS_INT property, CS_INT result);
  CS_RETCODE setProp(CS_INT property, char *result);

  // Getting properties.
  CS_RETCODE getProp(CS_INT property, CS_INT &result);
  CS_RETCODE getProp(CS_INT property, char *result, CS_INT buffsize);

  // Clearing properties,
  CS_RETCODE clearProp(CS_INT property);

  // Result of initialization.
  CS_RETCODE state() { return _state; }

  // Change the message file.
  static void messages(const char *fl);
  static void messages(ostream &ops);
  static void terminal(CS_BOOL tf);
  static void messageLimit(CS_INT lmt);

  DB_init(CS_INT vrs = CS_VERSION_100);
  ~DB_init();
};
```

DESCRIPTION

A *DB_init* object must be created to initialize the *SYBASE* Open Client Client-Library and the DB++ classes. A file, $DBAPPLACE/dbmessages (default), is opened for messages that come from *SYBASE* as well as those that come from *DB++* classes. The constructor assumes by default a *SYBASE* of version 10.0 (currently the only available version).

CONSTRUCTOR

There is only one constructor for *DB_init* objects. The default argument chooses the version behavior of the Client-Libraries that the DB++ classes are tuned for. Using another version may lead to unpredictable results.

If their is a failure in creating this object then you can find out from the DB_init::state() method. This will return CS_SUCCEED if there is no problem.

PROPERTY METHODS

The properties can be changed through the following methods:

```
CS_RETCODE setProp(CS_INT property, CS_INT result);
CS_RETCODE setProp(CS_INT property, char *result);
```

where char strings are assumed to be NULL terminated. For example to turn on ANSI binds we use the following code segment:

```
DB_init syb;

// Some more stuff ....
```

```
if (syb.setProp(CS_ANSI_BINDS, CS_TRUE) != CS_SUCCEED) {
  // Handle the error ...
}

// And continue ...
```

To find out what a property is set to, use the following methods:

```
CS_RETCODE getProp(CS_INT property, CS_INT &result);
CS_RETCODE getProp(CS_INT property, char *result, CS_INT buffsize);
```

where buffsize is the amount of space allocated to the char *result in the second getProp method. An example of getting the version string of the current Client-Library:

```
char vers[50];

// Some more stuff ....

DB_init syb;

// Some more stuff ....

if (syb.getProp(CS_VER_STRING, ver, 50) != CS_SUCCEED) {
  // Handle the error ...
}

// And continue ...
```

Properties can be cleared, returning to the default values with:

```
CS_RETCODE clearProp(CS_INT property);
```

MESSAGES

When a *DB_init* object is created, all messages will be sent to the file $DBAPPLACE/dbmessages. This can be changed using either of the methods

```
static void messages(const char *fl);
static void messages(ostream &ops);
static void terminal(CS_BOOL tf);
static void messageLimit(CS_INT lmt);
```

where the char input is used for a new file name or the messages can be sent to an output stream of your choice. Note, many messages are also sent to the standard error of the screen.

The method terminal, can used to turn on or off messages to the terminal. When the argument is CS_TRUE (default) then messages are printed to the terminal, else if the argument is CS_FALSE then messages are only printed to the message file.

The message limit is set to 10 initially. If the message limit is exceeded, DB++ will exit the program. This can be changed with the method messageLimit.

SEE ALSO

DB++(3), DB_properties(3)

DB_options(3)

NAME

DB_options - Options that can changed in DB_user objects.

SYNOPSIS

#include <DB++/db_base.H>

DESCRIPTION

The following is a list of options that can be set, obtained or cleared on *DB_user* objects. These are taken from the table "Symbolic Constants for Server Options" in "Open Client Client-Library/C Reference Manual", starting on page 2-83.

OPTIONS

CS_OPT_ANSINULL

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, then the ANSI behavior is used where "= NULL" and "is NULL" are not equivalent. In standard transact SQL, "= NULL" and "is NULL" are equivalent. Note, "<> NULL" and "is not NULL" are affected in a similar fashion.

CS_OPT_ANSIPERM

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this statement is CS_TRUE, SQL server will be ANSI-compliant with respect to permissions checks on "update" and "delete" statements.

CS_OPT_ARITHABORT

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determine how SQL server behaves when an arithmetic error occurs.

CS_OPT_ARITHIGNORE

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determines whether SQL server displays a message after a divide-by-zero error or a loss of precision.

CS_OPT_AUTHOFF

Argument: char* string, default Not applicable

Turns the specified authorization level off the current server session. When a user logs in, all authorizations granted to that user are turned on.

CS_OPT_AUTHON

Argument: char* string, default Not applicable

Turns the specified authorization level on the current server session. When a user logs in, all authorizations granted to that user are turned on.

CS_OPT_CHAINXACTS

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is CS_TRUE, SQL server uses chained transaction behavior. Unchained transaction behavior requires an explicit "commit transaction" statement to define a transaction. Chained transaction behavior means that each server command is considered to be a distinct transaction.

CS_OPT_CURCLOSEONXACT

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

When set to CS_TRUE, all cursors opened within a transaction space are closed when the transcations completes.

CS_OPT_CURREAD

Argument: char* string, default NULL

Set a security label specifying the current read level.

CS_OPT_CURWRITE

Argument: char* string, default NULL

Set a security label specifying the current write level.

CS_OPT_DATEFIRST
Argument: CS_INT value, default CS_OPT_SUNDAY

This option sets the day considered to be the first day of the week.

CS_OPT_DATEFORMAT
Argument: CS_INT value, default CS_OPT_FMTMDY

This option sets the order of the date parts month, day and year for entering "datetime" or "small-datetime" data.

CS_OPT_FIPSFLAG
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, SQL server flags any non-standard SQL commands that are sent.

CS_OPT_FORCEPLAN
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, SQL server joins tables in the order in which the tables are listed in the from clause of the query.

CS_OPT_FORMATONLY
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, SQL server will send back a description of the data, rather than the data itself, in response to a select query.

CS_OPT_GETDATA
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, then on every "insert", "delete" or "update" SQL server returns information (in the form of a message result set and parameters) that an application can use to construct the name of the tempory table that will contain the rows that will be inserted and/or deleted. Note that an update consists of insertions and deletions.

CS_OPT_IDENTITYOFF
Argument: char* string, default NULL

Disables inserts into a table's identity column.

CS_OPT_IDENTITYON
Argument: char* string, default NULL

Enables inserts into a table's identity column.

CS_OPT_ISOLATION
Argument: CS_INT value, default CS_OPT_LEVEL1

This option is used to specify a transaction isolation level. Possible levels are CS_OPT_LEVEL1 and CS_OPT_LEVEL3. Setting CS_OPT_ISOLATION to CS_OPT_LEVEL3 causes all pages of the tables in a select query inside a transaction to be locked for the duration of the transaction.

CS_OPT_NOCOUNT
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

This option causes the SQL server to stop sending back information about the number of rows affected by each SQL statement.

CS_OPT_NOEXEC
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, SQL server processes queries through the compile step but does not execute them. This option is used with CS_OPT_SHOWPLAN

CS_OPT_PARSEONLY
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set, the server checks the syntax of queries, returning error messages as necessary, but does not execute the queries.

CS_OPT_QUOTED_IDENT
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, SQL server treats all strings enclosed in double quotes as identifiers.

CS_OPT_RESTREES
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, SQL server checks the syntax of queries but does not execute them, returning parse resolution trees (in the form of image columns in a regular row resulta set) and error messages as needed to the client.

CS_OPT_ROWCOUNT
Argument: CS_INT value, default 0 (all rows are retuurned)

If this option is set, SQL server returns only a maximum specified number of regular rows for "select" statments. This option does not limit the number of computed rows returned. CS_OPT_ROWCOUNT works somewhat differently from most options. It is always set on, never off. Setting CS_OPT_ROWCOUNT to "0" sets it back to the default, which will return all the rows generated by a select statement. Therefore the way to turn CS_OPT_ROWCOUNT off is to set it on with a count of 0.

CS_OPT_SHOWPLAN
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, SQL server will generate a description of its processing plan after compilation and continue executing the query.

CS_OPT_STATS_IO
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

This option determines whether SQL server internal I/O statistics are returned to the client after each query.

CS_OPT_STATS_TIME
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

This option determines whether SQL server parsing, compilation and execution time statistics are returned to the client after each query.

CS_OPT_STR_RTRUNC
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is CS_TRUE, SQL server will be ANSI-compliant with respect to right truncation of character data.

CS_OPT_TEXTSIZE
Argument: CS_INT value, default 32768

This option changes the value of the SQL server variable @@*textsize*, which limits the size of text or image values that SQL server returns. When setting this option, you supply a parameter which is the length, in bytes, of the longest text or image value that the SQL server should return. @@*textsize* has a default of 32768 bytes. Note that, in programs that allow application users to make ad hoc queries, the user may override this option with the Transact-SQL set textsize command. To set a text limit that the user cannot override, use the Client-Library CS_TEXTLIMIT property instead.

CS_OPT_TRUNCIGNORE

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If this option is set to CS_TRUE, then SQL server ignores truncation errors. This is standard ANSI behavior. Otherwise, SQL server raises an error whenever a conversion results in truncation.

SEE ALSO

DB_user(3)

REFERENCES

SYBASE: "Open Client Client-Library/C Reference Manual"

NAME

DB_primitive - The C++ base class for the data.

SYNOPSIS

```
#include <DB++/db_base.H>

enum DB_types {
  IS_DB_void,
  IS_DB_string,
  IS_DB_binary,
  IS_DB_datetime,
  IS_DB_double,
  IS_DB_byte,
  IS_DB_float,
  IS_DB_long,
  IS_DB_int,
  IS_DB_short,
  IS_DB_char
};

class DB_primitive
{
public:
  DB_primitive() { }
  virtual CS_BOOL isNull() const = 0;
  virtual DB_types type() const = 0;
  virtual void print(ostream &) const = 0;
  virtual void read(istream &) = 0;
  virtual CS_RETCODE store(CS_CONTEXT *cnt, const CS_CHAR *str) = 0;
  virtual CS_RETCODE recall(CS_CONTEXT *cnt, CS_CHAR *str,
                            CS_INT size) const = 0;
  virtual ~DB_primitive() { }

  virtual DB_primitive& operator=(const DB_primitive &vv) = 0;
  virtual DB_primitive& operator=(const CS_CHAR *vv)     = 0;
  virtual DB_primitive& operator=(CS_CHAR vv)          = 0;
  virtual DB_primitive& operator=(const CS_BYTE *vv)     = 0;
  virtual DB_primitive& operator=(CS_BYTE vv)          = 0;
  virtual DB_primitive& operator=(CS_REAL vv)          = 0;
  virtual DB_primitive& operator=(CS_FLOAT vv)         = 0;
  virtual DB_primitive& operator=(CS_SMALLINT vv)      = 0;
  virtual DB_primitive& operator=(int vv)              = 0;
  virtual DB_primitive& operator=(long vv)             = 0;

  virtual operator const CS_CHAR*() const = 0;
  virtual operator const CS_BYTE*() const = 0;
  virtual operator CS_FLOAT()      const = 0;
  virtual operator CS_BYTE()       const = 0;
  virtual operator CS_REAL()       const = 0;
  virtual operator long()          const = 0;
  virtual operator int()           const = 0;
  virtual operator CS_SMALLINT()   const = 0;
  virtual operator CS_CHAR()       const = 0;
};
```

```
inline ostream& operator << (ostream &oo, const DB_primitive &zz)
{
  zz.print(oo);
  return oo;
}

inline istream& operator >> (istream &ii, DB_primitive &zz)
{
  zz.read(ii);
  return ii;
}
```

DESCRIPTION

This is the abstract base class for the following classes:

DB_void
DB_string
DB_binary
DB_datetime
DB_double
DB_byte
DB_float
DB_int
DB_short
DB_char

Given a data object of one of the above types, for instance, by subscripting a *DB_column* object, the type can be found out through the method, type. Use the method, isNull, to find out if the data is NULL.

Conversion to and from C data types is achieved through the operator = and the conversion operator. Note, in some cases where conversion is not possible, a default value is inserted. This default may not be the same as in the database. Storing and retrieving data can be done either through a C++ code entirely using the overloaded operators << or >> to streams.

Some of the conversions, such as date-time conversions, can go through the database by using the store and recall methods on a *DB_command* object. Note, not all of the conversions use the database, thus some defaults may missed.

SEE ALSO

DB++(3), DB_command(3), DB_column(3)

NAME

DB_properties - Properties that can changed in DB++

SYNOPSIS

#include <DB++/db_base.H>

DESCRIPTION

The list of properties that can be read, set or cleared. These properties pertain to the way the *SYBASE* Open Client Client-Library which is the foundation for the DB++ classes will behave. These are taken from the table "Summary of Properties" in "Open Client Client-Library/C Reference Manual", starting on page 2-95.

DB_init PROPERTIES

CS_ANSI_BINDS

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Turning on or off ANSI-Style binds. When ANSI-Style binds are in effect: (1) An application must either bind no items or all items, (2) When fetching data, an indicator must be used for copying NULL data. Otherwise a CS_ROW_FAIL will be returned.

CS_DISABLE_POLL

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determines whether or not ct_poll reports asynchonous operation completions. The default means polling is not disabled. If CS_DISABLE_POLL is CS_TRUE, an application cannot call ct_wakeup.

CS_EXPOSE_FMTS

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Whether or not Client-Library exposes format result sets.

CS_EXTRA_INF

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Whether or not Client-Library returns extra information to fill the structures *SQLCA, SQLCODE* and *SQLSTATE.*

CS_HIDDEN_KEY

Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determines whether *hidden-keys* will be exposed in a results set.

CS_IFILE

Argument: char* string, default NULL

Defines the name and location of the interfaces file.

CS_LOGIN_TIMEOUT

Argument: CS_INT value, default 60

Defines the length of time, in seconds, that Client-Library waits for a login response when making a connection attempt.

CS_MAX_CONNECT

Argument: CS_INT value, default 25

The maximum number of simultaneously open connections that a context can have.

CS_MEM_POOL

Not implemented.

Identifies a pool of memory that the Client-Library can use to satisfy its memory requirements.

CS_NETIO

Argument: CS_INT value, default CS_SYNC_IO

Determines whether the connection is synchronous (CS_SYNC_IO), fully asynchronous (CS_ASYNC_IO) or deferred asynchronous (CS_DEFER_IO). A deferred asynchronous

connection cannot be set in *DB_user. The current version of DB++ only works with CS_NETIO set to* CS_SYNC_IO.

CS_NO_TRUNCATE
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determines whether Client-Library truncates or sequences Client-Library and server messages that are longer than CS_MAX_MSG - 1 bytes. The default is to truncate long messages.

CS_NOINTERRUPT
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Whether or not the application can be interrupted by Client-Library.

CS_TEXTLIMIT
Argument: CS_INT value, default CS_NO_LIMIT

Sets the limit to the size, in bytes, of the largest text or image value that an application wants to receive.

CS_TIMEOUT
Argument: CS_INT value, default CS_NO_LIMIT

Sets the time limit, in seconds, that the Client-Library waits for a server response when making a request.

CS_USER_ALLOC
Not implemented.

CS_USER_FREE
Not implemented.

CS_VER_STRING
Argument: returns char* string

Returns the version of the Client-Library that the application is using.

CS_VERSION
Argument: returns CS_INT

Returns the version of the Client-Library that was requested in the constructor of *DB_init.*

DB_user PROPERTIES

Login properties

CS_APPNAME
Argument: char* string, default NULL

Defines the application name that the connection will use when connecting to the server. This will appear in the *sysprocesses* table in the *master* database.

CS_BULK_LOGIN
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Describes whether or not a connection can perform a bulk copy operations into a database. Applications that allow users to make ad hoc queries may want to avoid setting this property to CS_TRUE, to keep users from initiating a bulk copy sequence via *SQL* commands. Once a bulk copy sequence is begun, it cannot be stopped with an ordinary *SQL* command.

CS_HOSTNAME
Argument: char* string, default NULL

The name of the host machine used when logging into the server.

CS_LOC_PROP
Not implemented

Allows changing the default locale information, such as, language, character sets,

datetime formats and a collating sequence.

CS_PACKETSIZE
Argument: CS_INT value, default 512 (most platforms)

Determines the size for sending Tabular Data Streams (TDS) packets.

CS_PASSWORD
Argument: char* string, default NULL

Defines the password for logging into the server.

CS_TDS_VERSION
Argument: CS_INT value, default is dependant on CS_VERSION

Defines the bersion of the TDS protocol that the connection is using.

CS_USERNAME
Argument: char* string, default NULL

Defines the user name for logging into the server.

Properties that can only be set before connecting to server.

CS_COMMBLOCK
Not implemented.

This property is specific to IBM-370 and is ignored on other platforms.

CS_EXPOSE_FMTS
See DB_init PROPERTIES above.

CS_NETIO
See DB_init PROPERTIES above.

CS_SEC_APPDEFINED
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determines whether or not a connection will use the Open Server application-defined challenge/response security handshaking.

CS_SEC_CHALLENGE
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determines whether or not a connection will use SYBASE-defined challenge/response security handshaking.

CS_SEC_ENCRYPTION
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determines whether or not a connection will use encrypted password security handshaking.

CS_SEC_NEGOTIATE
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determines whether or not a connection will use trusted-user security handshaking.

Retrieve only properties

CS_CHARSETCNV
Argument: returns CS_TRUE or CS_FALSE

Describes whether or not the server is converting between the client and server character sets.

CS_CON_STATUS
Argument: returns CS_INT sized bit mask

CS_CONSTAT_CONNECTED implies the connection is marked open, however, CS_CONSTAT_DEAD implies the connection is marked as dead.

CS_EED_CMD
Not implemented.

Provides a pointer to a CS_COMMAND structure that contains extended error data.

CS_END_POINT
Argument: returns CS_INT

The file descriptor for a connection.

CS_LOGIN_STATUS
Argument: returns CS_TRUE or CS_FALSE

Returns CS_TRUE if connection is open.

CS_NOTIF_CMD
Not implemented

Defines a pointer to a CS_COMMAND structure containing registered procedure notification parameters.

CS_PARENT_HANDLE
Not implemented.

Returns the parent CS_CONTEXT structure.

CS_SERVERNAME
Argument: returns char *

Returns the name of the server.

Other properties

CS_ANSI_BINDS
See DB_init PROPERTIES above.

CS_ASYNC_NOTIFS
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

Determines whether a connection will receive registered procedure notifications asynchronously.

CS_DIAG_TIMEOUT
Argument: CS_TRUE or CS_FALSE, default CS_FALSE

If CS_TRUE, Client-library marks a connection as dead when a Client-library routine generates a timeout error, else Client-library retries indefinitely.

CS_DISABLE_POLL
See DB_init PROPERTIES above.

CS_EXTRA_INF
See DB_init PROPERTIES above.

CS_HIDDEN_KEYS
See DB_init PROPERTIES above.

CS_TEXTLIMIT
See DB_init PROPERTIES above.

CS_TRANSACTION_NAME
Argument: char* string, default NULL

Defines a transaction name.

CS_USERDATA
Not implemented.

DB_command PROPERTIES

CS_ANSI_BINDS

See DB_init PROPERTIES above.

CS_CURSOR_ID

Argument: returns CS_INT value

The server identification number assigned to a cursor. Retrieve only, after CS_CUR_STATUS indicates an existing cursor.

CS_CUR_NAME

Argument: returns char* string

The name with which the cursor was declared. Retrieve only, after ct_cursor with CS_CURSOR_DECLARE returns CS_SUCCEED.

CS_CUR_ROWCOUNT

Argument: returns CS_INT value

The number of cursor rows returned to Client-library per internal fetch request. Note this is not the number of rows returned to the application per ct_fetch call. Retrieve only, after CS_CUR_STATUS indicates an existing cursor.

CS_CUR_STATUS

Argument: returns CS_INT sized bit mask

The following values are returned:
CS_CURSTAT_CLOSED A closed cursor exists,
CS_CURSTAT_DECLARED A cursor is currently declared,
CS_CURSTAT_NONE No cursor is declared,
CS_CURSTAT_OPEN An open cursor exists,
CS_CURSTAT_RDONLY The cursor is readonly,
CS_CURSTAT_UPDATABLE The cursor is updatable.

CS_HIDDEN_KEYS

See DB_init PROPERTIES above.

CS_PARENT_HANDLE

Not implemented.

Returns the parent CS_CONNECTION structure.

CS_USERDATA

Not implemented.

SEE ALSO

DB_init(3), DB_user(3), DB_command(3)

REFERENCES

SYBASE: "Open Client Client-Library/C Reference Manual"

NAME

DB_table - The C++ class for working with tables stored in program memory.

SYNOPSIS

```
#include <DB++/db_base.H>

class DB_table
{
public:
  DB_string name()      const;
  CS_BOOL   isEmpty()   const;
  CS_BOOL   isTable(const char *tname) const;

  // Table dimensions.
  CS_INT    numRows()   const;
  CS_INT    numColumns() const;

  // Editing a table in memory
  void      empty();
  void      appendNewRow();
  void      removeLastRow();

  // Writing the structure in various formats
  void      cHeader(ostream &osrt = cout)   const;
  void      sqlScript(ostream &osrt = cout) const;

  // Data retrieval.
  DB_column &       operator () (const char *cname);
  const DB_column &  operator () (const char *cname) const;
  DB_column &       operator [] (CS_INT col);
  const DB_column &  operator [] (CS_INT col) const;

  DB_table();
  ~DB_table();
};
```

DESCRIPTION

A *DB_table* object is used to store the contents and some of the structure information of a table in the database in program memory. The *DB_table* objects are made available through a *DB_command* object. Generally, transference of data between the database and a *DB_table* object will be handled using a *DB_command* object. A stand alone *DB_table* object will not be of much use.

CONSTRUCTORS

There is a default constructor, however this will not be of much use without database access which are available from the *DB_command* objects.

TABLE STRUCTURE

A table consists of rows and columns of data. Each column has a name attached to it and its data has a particular representation. The following methods are used to find out the structure of a table in program memory.

```
  DB_string name()      const;
  CS_INT    numRows()   const;
  CS_INT    numColumns() const;
  void      cHeader(ostream &osrt = cout)   const;
  void      sqlScript(ostream &osrt = cout) const;
```

Each table has a name. The name of the table can be found from the method, name.

The method, numRows, returns the number of rows of data.

The method, numColumns, returns the number of columns.

The method, cHeader, will send a C structure representation of the data to an ostream of your choice.

The method, sqlScript, will send a "CREATE TABLE" *SQL* script of this table to an ostream. Note, this *SQL* script doesn't know the difference between a char and varchar data (assumes varchar) and binary and varbinary data (assumes varbinary). Furthermore, indexing, default values, and other features of in *SQL* are not in this script.

UTILITIES

The following utilities are used to help manage a table in program memory.

```
CS_BOOL   isTable(const char *tname) const;
CS_BOOL   isEmpty()  const;
void      empty();
void      appendNewRow();
void      removeLastRow();
```

The method, isTable, checks the name of a this table.

The method, isEmpty, determines whether any data is stored in the current table.

Using the method, empty, a table can be emptied of all its data leaving the structure information.

Use the method, appendNewRow, to append a row with all NULL data to an existing table. Applying this method to an empty table will lead to a non-empty table, even though all the data is NULL. Data values can then be entered into this new row.

You can also remove the last row in the table using the method, removeLastRow.

DATA HANDLING

Data can be retrieved or sent to a table in program memory through the following subscript operators.

```
DB_column &       operator () (const char *cname);
const DB_column & operator () (const char *cname) const;
DB_column &       operator [] (CS_INT col);
const DB_column & operator [] (CS_INT col) const;
```

A *DB_column* object for a given column is returned given either the column name or its number starting from zero to the last column. The column structure information and its data are stored in the *DB_column* objects.

SEE ALSO

DB++(3), DB_command(3), DB_column(3), DB_primitive(3)

DB_user(3)

NAME

DB_user - The C++ class to connect to the Database Server.

SYNOPSIS

```
#include <DB++/db_base.H>

class DB_user
{
public:
  // Setting options.
  CS_RETCODE setOpt(CS_INT option, CS_INT result);
  CS_RETCODE setOpt(CS_INT option, char *result);

  // Getting options.
  CS_RETCODE getOpt(CS_INT option, CS_INT &result);
  CS_RETCODE getOpt(CS_INT option, char *result, CS_INT buffsize);

  // Clearing options
  CS_RETCODE clearOpt(CS_INT option);

  // Setting properties.
  CS_RETCODE setProp(CS_INT property, CS_INT result);
  CS_RETCODE setProp(CS_INT property, char *result);
  CS_RETCODE enableBulkCopy(CS_INT version = BLK_VERSION_100);

  // Getting properties.
  CS_RETCODE getProp(CS_INT property, CS_INT &result);
  CS_RETCODE getProp(CS_INT property, char *result, CS_INT buffsize);

  // Clearing properties,
  CS_RETCODE clearProp(CS_INT property);

  // Setting and getting capabilties.
  CS_RETCODE getRequest(CS_INT capability, CS_BOOL &tf);
  CS_RETCODE setResponse(CS_INT capability, CS_BOOL tf);
  CS_RETCODE getResponse(CS_INT capability, CS_BOOL &tf);

  // Copying login info from one connection to another.
  CS_RETCODE copyLogin(DB_user &cnt);

  // Establishing the connection.
  CS_RETCODE connect(char *server);
  CS_RETCODE dbloginFile();
  CS_RETCODE dbloginFile(const char *dbase);

  // Result of initialization.
  CS_RETCODE state();

  // Allocating memory for the connection the connection.
  DB_user(DB_init &db);

  // Frees up the login record.
  ~DB_user();
};
```

DESCRIPTION

It is through the *DB_user* object that one connects to the server. It is assumed that a *DB_init* object exists.

CONSTRUCTOR

There is only one constructor that uses for its argument a *DB_init* object. This creates a *DB_user* object, which has not yet established a connection to the server. Before the connection is established,the *DB_user* object must be informed about the username, password, etc. This can accomplished through changing the properties or using a method that reads this info from a *.dblogins* file.

Normally, this object is created without a problem, however, if their is a problem after construction one could find out from the method DB_user::state(). This method returns CS_SUCCEED if the constructor succeed.

OPTION METHODS

The options can be changed through the following methods:

```
CS_RETCODE setOpt(CS_INT option, CS_INT result);
CS_RETCODE setOpt(CS_INT option, char *result);
```

where char strings are assumed to be NULL terminated. For example to limit the number of rows returned from a regular row result select statement use:

```
DB_init syb;
DB_user lgn(syb);

// Some more stuff ....

if (lgn.setOpt(CS_OPT_ROWCOUNT, 50) != CS_SUCCEED) {
  // Handle the error ...
}

// And continue ...
```

To find out what a option is set to, use the following methods:

```
CS_RETCODE getOpt(CS_INT option, CS_INT &result);
CS_RETCODE getOpt(CS_INT option, char *result, CS_INT buffsize);
```

where buffsize is the amount of space allocated to the char *result in the second getProp method. An example of checking to see if ANSI NULL's are in effect use (note, CS_BOOL and CS_INT are both 4-byte integers in Client-Library):

```
CS_BOOL tf;

// Some more stuff ....

DB_init syb;
DB_user lgn(syb);

// Some more stuff ....

if (lgn.getOpt(CS_ANSINULL, tf) != CS_SUCCEED) {
  // Handle the error ...
}

// And continue ...
```

Options can be cleared, returning to the default values with:

```
CS_RETCODE clearOpt(CS_INT option);
```

PROPERTY METHODS

The properties can be changed through the following methods:

```
CS_RETCODE setProp(CS_INT property, CS_INT result);
CS_RETCODE setProp(CS_INT property, char *result);
```

where char strings are assumed to be NULL terminated. For example to turn on ANSI binds we use the following code segment:

```
DB_init syb;
DB_user lgn(syb);

// Some more stuff ....

if (lgn.setProp(CS_ANSI_BINDS, CS_TRUE) != CS_SUCCEED) {
  // Handle the error ...
}

// And continue ...
```

To find out what a property is set to, use the following methods:

```
CS_RETCODE getProp(CS_INT property, CS_INT &result);
CS_RETCODE getProp(CS_INT property, char *result, CS_INT buffsize);
```

where buffsize is the amount of space allocated to the char *result in the second getProp method. An example of getting the server name after establishing the connection:

```
char server[50];

// Some more stuff ....

DB_init syb;
DB_user lgn(syb);

// Connecting to server and other stuff ....

if (lgn.getProp(CS_SERVERNAME, server, 50) != CS_SUCCEED) {
  // Handle the error ...
}

// And continue ...
```

Properties can be cleared, returning to the default values with:

```
CS_RETCODE clearProp(CS_INT property);
```

CAPABILITY METHODS

There are two types of capabilities: request and response. Request describes the types of client requests that can be sent on a server connection. Response describe the types of server responses that a connection does not wish to receive. Note, request capabilities can not be changed but only retrieved.

```
CS_RETCODE getRequest(CS_INT capability, CS_BOOL &tf);
CS_RETCODE setResponse(CS_INT capability, CS_BOOL tf);
CS_RETCODE getResponse(CS_INT capability, CS_BOOL &tf);
```

DB_user(3)

COPYING LOGIN INFORMATION

It is possible to copy all the login information from an exiting *DB_user* object to a new *DB_user* object using the following method:

```
CS_RETCODE copyLogin(DB_user &cnt);
```

Note, we did not use the equals operator here since only the login properties were copied.

CONNECTING TO SERVER

There are three methods for establishing a connection to the server:

```
CS_RETCODE connect(char *server);
CS_RETCODE dbloginFile();
CS_RETCODE dbloginFile(const char *dbase);
```

The first method will connect to a server of a given name, it is assumed that all the necessary login properties, such as user name, password, etc, have been entered.

The second method looks for the file $HOME/.dblogins. If this file exists it will read the first line for the login properties. If this file does not exist or if there is a read error in trying to read the first line, then it will look for the the file $DBAPPLACE/.dblogins where it will read the first line. This file may not exist or, there may be a read error in reading the first line, in this case the user-name and password are set to "harmless" and the program uses the DSQUERY enviromental variable for the server name. If the DSQUERY variable does not exist, the method exits with an error message and no connection is established.

The third method is similar to the second method except that the .dblogins file lines are searched for the database name.

SEE ALSO

DB++(3), DB_capabilities(3), DB_init(3), DB_options(3), DB_properties(3), dblogins(1)

REFERENCES

SYBASE: "Open Client Client-Library/C Reference Manual"

```
#include <DB++/db_base.H>

int main(int argc, char* argv[])
{
   DB_string dbname, *tblNames;
   CS_INT i, tbcnt, cmd_fails, cmds;

   // Get database name to query.
   if (argc == 2) {
      dbname = argv[1];
   } else {
      cerr << "Usage:  querydb 'Database Name'" << endl;
      return(0);
   }

   // Initializing the database server.
   DB_init bgn;
   if (bgn.state() != CS_SUCCEED) {
      cerr << "Error -- Could not initialize Access to the database." << endl;
      return(0);
   }
   DB_user srv(bgn);
   if (srv.state() != CS_SUCCEED) {
      cerr << "Error -- Could not initialize the database server." << endl;
      return(0);
   }

   // Connect to the server.
   if (srv.dbloginFile() != CS_SUCCEED) {
      cerr << "Error -- Failed to open a connection to the server." << endl;
      return(0);
   }

   // Opening up the database.
   DB_command tbl(srv);
   if (tbl.setDatabase(dbname) != CS_SUCCEED) {
      cerr << "Error -- Failed to set the database." << endl;
      return(0);
   }

   // Get list of all user tables from database.
   tbl.clearSQL();
   tbl.appendSQL("SELECT name FROM sysobjects WHERE type = 'U' ORDER BY name");
   tbl.execSQL("tableNames", cmd_fails, cmds);
   if (cmd_fails != 0) {
      cerr << "Error -- Unable to get table names from the database."
         << endl;
      return(0);
   }
   tblNames = new DB_string [tbcnt = tbl("tableNames").numRows()];
   for (i = 0; i < tbcnt; i++) {
      tblNames[i] = tbl("tableNames")("name")[i];
      tblNames[i].trimend();
   }
   tbl.removeLastTable();
   cout << tbcnt << " user tables in the database: " << dbname << endl;

   for (i = 0; i < tbcnt; i++) {
      tbl.describeTable(tblNames[i]);
      tbl(tblNames[i]).sqlScript();
      tbl.removeLastTable();
   }
   delete [] tblNames;
   return(1);
}
```