RC21
Relational Database Class Library
Release 8.1

Vermont Database Corporation
400 Upper Hollow Hill Road
Stowe VT 05672
802-253-4437

RC21 Release 7.6

DISCLAIMER OF WARRANTY

This manual and associated software are sold "as is" and without warranties as to performance or merchantability. The seller's salespersons may have made statements about this software. Any such statements do not constitute warranties.

This program is sold without any express or implied warranties whatsoever. No warranty of fitness for a particular purpose is offered. The user is advised to test the program thoroughly before relying on it. The user assumes the entire risk of using the program. Any liability of seller of manufacturer will be limited exclusively to replacement of diskettes defective in materials or workmanship.

Vermont Database Corporation
400 Upper Hollow Hill Road
Stowe, VT 05672
802-253-4437
http://www.vermontdatabase.com

RC21 Programmer's Guide and Reference Manual

RC21

This manual printed 6/14/99.

Table of Contents

Programmer's Guide

A General Description

RC21 is a Relational Database Management System (RDBMS) for C and C++ programmers. It is presented as a number of surfaces - primarily as a C++ class library, a C function library, and an ODBC function library.

RC21 started out as a C function library called Pinnacle Relational Engine (PRE). When C++ was more standardized and C++ compilers became the rule, the functionality was implemented as a class library. Actually, PRE had a class wrapper that closely resembles RC21, but now the kernel code is written in C++. This has improved the robustness and maintainability of the code.

An RC21 database is a single file in the file system of your operating system. The database contains tables. Each table has a name, a description, and other attributes that may be added by the programmer. A table contains zero or more columns (actually there is always at least ONE (hidden) column). Each column also has a name, a description, optionally a data type, a printf format code, optionally a reference column that contains a list of valid values for the column. The set of tables and their columns and all their attributes are contained in two user-accessible tables named "_tables" and "_columns". Thus, all the relevant information about the structure of the database is contained in the database itself and is available to the programmer. This enables complete manipulation of the database - including dictionary operations - without the necessity of invoking special vendor-provided utility routines.

The RC21 database file is physically divided into self-describing "pages" of a size that can be specified at the time of creation of the database. The programmer may find it optimal to specify a size that is equivalent to the natural page size of the file system. This may improve disk access. Since each page is self-describing, sharing databases in a heterogeneous network environment is possible.

All operations on the database are cached in RAM or in temporary disk storage until a commit operation is invoked by the programmer. Commits are guaranteed to be atomic - that is - they are either all or nothing. If a commit operation is interrupted by a power outage or some other catastrophe, it will be completed, if possible, automatically when the database is accessed the next time. If the commit cannot be completed, the database reverts to its state upon completion of the last previous commit.

C++ API Reference Card

The following is an edited version of the header file for RC21. Implementational details have been omitted.

```
// Special Types:
typedef CHAR FAR *DBSTRING;
typedef const CHAR FAR * CDBSTRING;
typedef void FAR *DBVOID;
typedef const void FAR * CDBVOID;
typedef UCHAR FAR *DBNBYTES;
typedef DBNBYTES DBBLOB;
typedef DBNBYTES DBUSER;
typedef const void FAR * CDBBLOB;
typedef const void FAR * CDBUSER;
typedef long DBHANDLE;
typedef long FILEADDR;
typedef enum DBERROR;
class Database *DB;
class Table *DBTAB;
class Column *DBCOL;
class SearchObject *DBSEARCH;

// Column Attribute Word
typedef unsigned long COLATTRIBS;
const COLATTRIBS
   TypeMask=0x0000ffff,
   Integer=0x00000001, Real=0x00000002, String=0x00000004,
   DBString=0x00000004,   NBytes=0x00000008, Blob=0x00000008,
   User=0x00000010, Null=0x00000020, Key=0x00000040,
   STRING=0x00000080, USTRING=0x00000080, Eblob=0x00000100,

   IndexMask=0x03000000,
   Indexed=0x01000000,
   NonIndexed=0x02000000L,

   NoNulls=0x04000000,
   Nulls=0x08000000,
   NullsMask=0x0c000000,

   Unique=0x10000000,
   NonUnique=0x20000000,
   UniqueMask=0x30000000,

   NonShared=0x40000000,
   Shared=0x80000000,
   SharedMask=0xc0000000,

   NOATTRIBS=0
   ;
#define KeyLen(q) ((COLATTRIBS)q<<16)
#define KeyLenMask KeyLen(0xffL)
#define GetKeyLen(q) ((unsigned int)(((q)&KeyLenMask)>>16))


// dictionary and table iterator macros
#define ForAllRows(_x) for (ResetRow(_x);NextRow(_x,1);)
#define ForAllRowsBackwards(_x) for (ResetRow(_x);NextRow(_x,-1);)
#define ForAllTables(_d) for (FirstTable(_d);NextTable(_d);)
#define ForAllColumns(_t) for (FirstColumn(_t);NextColumn(_t);)


class DBVALUE  // a self-describing database value
   {
   public:
      DBSIZE size(void) {return n;}
      DBBLOB ptr(void) {return v.b;}
      operator DBINTEGER();
      operator DBSTRING();
      operator DBREAL();
```

```
      operator int();
      operator float();
      DBVALUE(void);
      ~DBVALUE(void);
      DBVALUE(DBVALUE&V);
      DBVALUE(DBTYPE);
      DBVALUE& operator=(DBVALUE&V);
      DBVALUE& operator=(CDBSTRING);
      DBVALUE& operator=(DBINTEGER);
      DBVALUE& operator=(DBREAL);
   };

// create a new database
DBERROR Create (CDBSTRING path, unsigned pagesize=P_DefaultPagesize,
   unsigned nhash=P_DefaultNhash);

// remove a database
DBERROR Remove (CDBSTRING path);

class Database: public RCObject
   {
   public:
      // constructor, destructor
      Database (CDBSTRING dbname, CDBSTRING access=(CDBSTRING)"rw",
         int npages=20);
      Database (CDBSTRING dbname, int npages);
      ~Database(void);

      // conversions
      operator DB (void);

      // state of this Database
      int Changed(void);  // returns nonzero if database has been changed.
      int CachePages(void);
      CDBSTRING Path(void);
      CDBSTRING Name(void);
      CDBSTRING Release(void);
      unsigned PageSize(void);
      unsigned long HashSize(void);

      // adding, deleting a table
      DBERROR AddTable(CDBSTRING name,
         CDBSTRING description=(CDBSTRING)"no description",
         CDBSTRING view = (CDBSTRING)"");
      DBERROR DeleteTable(CDBSTRING name);

      // commit, rollback
      void Finish(void); // finish current transaction
      // find out resources required for commit
      DBERROR CommitStatus(unsigned long &dbsize, unsigned long &tempsize);
      DBERROR Commit(void);
      int AutoCommit(int onoff);
      int AutoCommit(void);
      DBERROR RollBack (void);

      // traversing list of tables in database
      DBTAB CurrentTable (void);
      void FirstTable (void);
      int NextTable (void);

      // copy contents of database to another database
      DBERROR Copy(DB to);

      // OPERATORS
      Database& operator =(Database& fromdb);
```

```
      class Table& operator[] (CDBSTRING);
   };


// find out the release number of a named database
CDBSTRING Release (CDBSTRING dbname);

class EXPORT Table: public RCObject
   {
   public:
      // constructors, destructor
      Table (DB db, CDBSTRING TableName);
      Table (CDBSTRING TableName); // table in current database
      Table (void);  // a placeholder for a table
      Table& operator=(Table& rtab);  // copies a table
      ~Table (void);

      // conversions
      operator DBTAB (void);  // converts to pointer to this
      operator DB (void);  // return pointer to Database this Table belongs to

      // erase all the rows in a table.
      DBERROR Erase (void);

      // attributes
      CDBSTRING Name (void);  // get the name
      DBERROR Name (CDBSTRING NewName);  // change the name
      CDBSTRING Description (void);  // get the description
      DBERROR Description (CDBSTRING NewDescription);  // change description
      CDBSTRING View (void);  // get the view expression
      DBERROR View (CDBSTRING v);  // change the view expression

      // adding, deleting columns
      DBERROR AddColumn(CDBSTRING Name,
         COLATTRIBS attr=NOATTRIBS,
         CDBSTRING description=(CDBSTRING)"no description",
         CDBSTRING format=(CDBSTRING)"",  // printf format
         const DBCOL reference=NULL // referential integrity column
         );
      DBERROR DeleteColumn(CDBSTRING n);

      // does a certain column exist for this table?
      int ColumnExists(CDBSTRING colname);

      // add a composite key to the table
      DBERROR AddKey (CDBSTRING Name, CDBSTRING Keys, COLATTRIBS attr=NOATTRIBS);

      // Adding, Deleting Rows
      DBERROR AddRow (void);
      DBERROR DeleteRow(void);
      void operator ++(void);  // same as AddRow

      // count the columns or rows in the table (not including internal or
      // deleted
      DBROWID CountColumns (void);
      DBROWID CountRows (void);

      // copying operations
      DBERROR Copy (CDBSTRING newtab); // make a copy of a table
      DBERROR CopyRow (DBROWID); // copy row to another row in same table.
      DBERROR CopyRow (DBTAB); // copy to current row of another table;
                               // values go to same-named columns
      DBERROR DuplicateRow (void);   // add a row and copy to it.
```

```
      // emptying a table or removing it ...
      void Empty(void);  // remove all the rows in a table
      DBERROR Zap (void);  // remove the table from the database

      // specify the order-by column for this table
      inline DBERROR OrderBy (CDBSTRING colname);
      DBERROR Unordered(void);

      // select or filter operations
      void ClearFilters (void);  // remove all filters
      DBERROR Filter (CDBSTRING f ...);
      DBERROR vFilter (CDBSTRING f, va_list ap);
      int Find (CDBSTRING filter ...);
      int vFind (CDBSTRING filter, va_list ap);
      int RowExists (CDBSTRING filter ...);
      int vRowExists (CDBSTRING filter, va_list ap);
      int PassesFilter (CDBSTRING filter ...);
      int vPassesFilter (CDBSTRING filter, va_list ap);

      // determine if current row has been deleted
      int IsaRow(void);

      // traversing a table
      DBERROR ResetRow(void);
      DBROWID CurrentRow(void) {return CurrentStatus.rowid;}
      int NextRow (int inc=1);  // skip rows if inc != 1 or -1
      int PrevRow (void);  // go backwards
      DBERROR GotoRow (DBROWID RowID);  // go to row with given Row ID

      // GetRow produces formatted string -- for PERL operations
      CDBSTRING GetRow (CDBSTRING column_list, char sep);

      // Traversing list of columns in a table
      void FirstColumn (void);
      int NextColumn (void);
      DBCOL CurrentColumn (void);

      // Complete operations on the current row
      DBERROR FinishRow (void);

      // reference column object with given name
      Column& operator [] (CDBSTRING ColumnName);
   };

class EXPORT Column: public RCObject
   {
   public:
      // constructors
      Column (DBTAB TableObject,  CDBSTRING ColumnName);
      Column (void); // place holder
      Column (DBCOL q);
      Column& operator=(DBCOL X);
      ~Column();

      // conversions
      operator DBCOL (void);
      operator DBTAB (void);  // return pointer to Table object for this column
      operator DB (void);  // return pointer to Database object for this column
      DBTAB Tab (void); // table to which this belongs

      // fetching, changing attributes
      CDBSTRING Name (void);
      DBERROR Name (CDBSTRING NewName);
      CDBSTRING Description (void);
```

```
DBERROR Description (CDBSTRING NewDescription);
CDBSTRING Format (void);
DBERROR Format (CDBSTRING NewFormat);
COLATTRIBS Attributes (void);  // fetch attributes
DBERROR Attributes(COLATTRIBS);  // change attributes
DBCOL Reference (void);  // referential integrity column

// storing a value in a column
DBERROR PutNull(void);  // store a Null (empty) value
DBERROR PutNULL(void);
DBERROR PutReal(DBREAL x);
DBERROR PutInteger(DBINTEGER i);
DBERROR PutString(CDBSTRING s);
DBERROR PutSTRING(CDBSTRING s);
DBERROR PutBLOB(CDBBLOB b,DBSIZE s);
DBERROR PutUser(const DBUSER b,DBSIZE s);
DBERROR PutUser(DBVALUE *u);
DBERROR PutEBLOB(FILEADDR f);
DBERROR Put(DBINTEGER i);
DBERROR Put(DBREAL x);
DBERROR Put(CDBSTRING s);
DBERROR Put(CDBBLOB b, DBSIZE s);
DBERROR Put(const DBUSER b,DBSIZE s);
DBERROR Put(const DBVALUE&);
DBINTEGER operator << (DBINTEGER x);  // store x in the column
int operator << (int x);
DBREAL operator << (DBREAL x);
CDBSTRING operator << (CDBSTRING x);

// fetching a value from a column
DBINTEGER GetInteger(void);
DBREAL GetReal(void);
CDBSTRING GetString(void);
DBVALUE& GetValue(void);
COLATTRIBS GetType (void);
DBSIZE GetSize (void); // how much RAM required to store value, when fetched
DBSIZE Size (void);
DBVALUE Get (void);
// conversions that fetch values
operator int(void);
operator long(void);
operator const char*(void);
operator double(void);
operator float(void);


// order this column's table by this column
DBERROR OrderBy(void);

// value operations on entire column
void Erase (void);  // erase all the values in a column
int HasAnyValues(void);  // are there any non-Nulls anywhere in the column

// copy contents of a column
DBERROR Copy (CDBSTRING newcol);

// copy just one value in current row to another column
DBERROR Copy (DBCOL);

// low-level operations tofind things in a table.
int Find (DBSEARCH so);
DBBOOL FindNext (DBSEARCH dbsearch, int direction);
DBBOOL KeyExists(DBSEARCH dbsearch);

// increment the value in the column
```

```
      DBINTEGER Inc(DBINTEGER _i);
      DBREAL Inc(DBREAL _i);

   };

// column-get operators
CDBSTRING operator << (CDBSTRING &s, Column& c);
DBINTEGER operator << (DBINTEGER &i, Column& c);
DBREAL operator << (DBREAL &x, Column& c);

// iostream operators with columns
ostream& operator << (ostream &o, Column& c);
istream& operator >> (istream& i, Column& c);

// SearchObject for limited search operations
class SearchObject: public RCObject
   {
   SearchObject (DBCOL c, COLATTRIBS attribs, const void *valptr,
      const CHAR *relation, size_t nbyte_length=0);
   ~SearchObject();
   };

#define E_SEEK_SET   0   /* Seeks from beginning of Eblob */
#define E_SEEK_CUR   1   /* Seeks from current position */
#define E_SEEK_END   2   /* Seeks from end of Eblob */

class eblob
   {
   public:
      eblob (DBCOL cc);  // construct an eblob from eblob value in the column
      ~eblob (void);
      int seek(FILEADDR f, int fromwhere=E_SEEK_SET);  // like filesystem seek
      int seek(void) {return seek (lfCurrent);}
      FILEADDR tell(void);
      size_t read (void* buf, size_t len);
      size_t write (void* buf, size_t len);
      FILEADDR inject (void* buf, FILEADDR len);
      FILEADDR extract (FILEADDR len);
   };
```

Overview of C++ Functions – How to ...

Create a Database

You create a database with the **Create** function which has the prototype

```
DBERROR Create (CDBSTRING path, unsigned pagesize=P_DefaultPagesize,
   unsigned nhash=P_DefaultNhash);
```

Pagesize and *nhash* may be omitted, which is the typical case. In fact, you may alter the defined constants for *pagesize* and *nhash* in the header file, RC21.H to fit your environment. Here is an example:

```
Create ("test.db");
```

Delete a Database

To delete a database, use the **Remove** function.

```
DBERROR Remove (CDBSTRING path);
```

Remove is essentially the same as the **remove** function in **stdio.h** but it is provided for symmetry with **Create** and also to provide safety in a multi-user (client-server) environment.

Access a Database

To access a database that has been **Create'd**, instantiate an object of the class **Database**. The constructor has the prototypes:

```
Database (CDBSTRING dbname, CDBSTRING access=(CDBSTRING)"rw",   int npages=20);
Database (CDBSTRING dbname, int npages);
```

The database object may be allocated with the **new** operator or may be declared as an object on the stack with a simple declaration. If you allocate an object using **new**, use the datatype **DB** which is simply **Database*** type. The *access* argument may have the values "r" for read access, "w" for write access, and "rw" or "wr" for read/write database access. The default access is "rw" and is typically omitted. The *npages* argument specifies the number of pages of RAM will be set aside for database buffering. Usually, the larger number yields better performance. However, be aware that an extremely large number here could lead to a virtual memory buffer which is really swapped in and out of disk. Example:

```
DB test = new Database("test.db");
```

or

```
Database test("test.db");
```

If the file exists and is an RC21 database, a valid object will be created. Otherwise, an *invalid* object will be created and an exception will be raised.

Add a Table to a Database

To add a table to a database, use the **AddTable** member function, which has the prototype:

```
DBERROR AddTable(CDBSTRING name,   CDBSTRING description=(CDBSTRING)"no
description",
CDBSTRING view = (CDBSTRING)"");
```

Name is the name of the table, which follows the RC21 naming rules, as is unique within a database. *Description* is optional and for documentation purposes only. If the table is really a *view* into another table, supply a *view expression* which is a string. For a discussion of views, see VIEWS on page 29. Here is an example:

```
Database db("test.db);
db.AddTable ("new_table", "new table");
```

Access an Existing Table

To access a table in a database, instantiate an object of class **Table**. The constructor has the prototype:

```
Table (DB db, CDBSTRING TableName);
Table (CDBSTRING TableName); // table in current database
```

The arguments are self-explanatory. Example:

```
Database db("test.db);
Table test(db, "test");
```

or

```
DB db = new Database("test.db");
DBTAB test = new Table (db, "test");
```

Add a Column to a Table

To add a column to a table, use the **AddColumn** member function. This has the prototype:

```
DBERROR AddColumn(CDBSTRING Name,
   COLATTRIBS attr=NOATTRIBS,
   CDBSTRING description=(CDBSTRING)"no description",
   CDBSTRING format=(CDBSTRING)"",  // printf format
   const DBCOL reference=NULL // referential integrity column
   );
```

Name is the name of the column in the table. Within a table, columns must have unique names. *Attr* is a value of type **COLATTRIBS**. The value is constructed by or'ing together certain constants to indicate *data-type*, *storage-type*, *indexing*, *unique*, etc. Here is an example:

```
Create ("test.db");
Database D("test.db");
Table T(D, "test");
T.AddColumn("LastName", String | Indexed | Shared);
T.AddColumn("FirstName", String | Shared);
T.AddColumn("Salary", Integer, "Monthly Salary", "%ld");
```

Add a Composite Key to a Table

A composite key is a virtual combination of all or part of columns that is inserted in an index. Once it has been declared for the table it is maintained automatically by RC21 as its constituent parts are written or modified. It is created using the **AddKey** member function of the Table class. The prototype is:

```
DBERROR AddKey (CDBSTRING Name, CDBSTRING Keys, COLATTRIBS attr=NOATTRIBS);
```

Here is an example:

```
Create ("test.db");
Database D("test.db");
Table T(D, "test");
T.AddColumn("LastName", String | Shared);
T.AddColumn("FirstName", String | Shared);
T.AddKey("LastFirst", "+Last+First", Unique);
```

For a complete discussion of this topic, see Composite Keys on page 23.

Access a Column in a Database

Once a column has been added to a table, it may be accessed by instantiating a **Column** object. The prototype for the constructor is:

```
Column (DBTAB TableObject,  CDBSTRING ColumnName);
```

Here is an example:

```
Create ("test.db");
Database D("test.db");
Table T(D, "test");
T.AddColumn("LastName", String | Shared);
Column LastName(DBTAB(T), "LastName");
```

Add A Row to a Table

Add a row to a table by using the **AddRow** member function of the Table class:

```
DBERROR AddRow (void);
```

Once the row has been added, the table cursor is placed at this newly-added row and the values in all the columns in this row have the *null* value. Here is an example:

```
Create ("test.db");
Database D("test.db");
Table T(D, "test");
T.AddColumn("LastName", String | Shared);
T.AddColumn("FirstName", String | Shared);
T.AddKey("LastFirst", "+Last+First", Unique);
Column LastName(DBTAB(T), "LastName");
Column FirstName(DBTAB(T), "FirstName");
T.AddRow();
```

After the row has been added in the above example, data must be stored into the columns to avoid a *unique* violation.

Store Data into a Column

Data is stored in each column of the current row by using the type-specific **Column** member functions **PutInteger**, **PutReal**, **PutString**, **PutBlob**, or **PutUser**. Better yet, use the polymorphic **Put** function or the **operator<<**, as in the following example:

```
Create ("test.db");
Database D("test.db");
Table T(D, "test");
T.AddColumn("LastName", String | Shared);
T.AddColumn("FirstName", String | Shared);
T.AddKey("LastFirst", "+Last+First", Unique);
Column LastName(DBTAB(T), "LastName");
Column FirstName(DBTAB(T), "FirstName");
T.AddRow();
LastName << "Smith";
FirstName << "John";
T.AddRow();
LastName << "Jones";
FirstName << "Trent";
```

Save Your Database Changes

Database changes, including changes to the dictionary (new tables, columns) are temporary until the data is commited to the physical database using the **Commit** member function of class **Database**:

```
DBERROR Commit(void);
```

as in the following example:

```
Create ("test.db");
Database D("test.db");
Table T(D, "test");
T.AddColumn("LastName", String | Shared);
T.AddColumn("FirstName", String | Shared);
T.AddKey("LastFirst", "+Last+First", Unique);
D.Commit();  // data written to physical database file
```

Traverse All the Rows in a Table

To visit every row in a table, use the **ResetRow** and the **NextRow** Table member functions or the **ForAllRows** iterator macro:

```
DBERROR ResetRow(void);
```

```
int NextRow (int inc=1);  // skip rows if inc != 1 or -1
int PrevRow (void);  // go backwards
#define ForAllRows(_x) for (ResetRow(_x);NextRow(_x,1);)
```

The **ResetRow** member function prepares to iterate over the entire table, row-by-row. It doesn't actually position to the first row. This is accomplished by the first **NextRow**. Each **NextRow** moves to the next available row (deleted rows are skipped as are filtered-out rows). Thus, the idiom for visiting all rows in a table is:

```
for (ResetRow(_x);NextRow(_x,1);)
   {
   .
   .
   .
   }
```

This idiom is conveniently encapsulated in the **ForAllRows** macro. Thus we may simply write:

```
ForAllRows(table)
   {
   .
   .
   .
   }
```

Get a Bookmark for the Current Row

A row may be accessed by some kind of *key*, or a reference to the current row may be obtained by using the **CurrentRow** member function of the Table class. Each row has a built-in unique (within a table) identifier of type **DBROWID**. This may be retrieved by using the **GetRow** member function:

```
DBROWID CurrentRow(void) {return CurrentStatus.rowid;}
```

The **DBROWID** thus retrieved may be later used to return to that row using the **GotoRow** member function. Note that bookmarks of type **DBROWID** are only valid while the database is open. The **DBROWID** for a particular row is not guaranteed to stay the same over time, except while the database is open. The **DBROWID** for a particular row will probably change if the table is copied. Thus it would be unwise to store data of type **DBROWID** in the database itself, except for purely temporary use.

Go to a Particular Row with a Bookmark

To go to a particular row in a table use the **GotoRow** member function with a previously-assigned **DBROWID** value:

```
DBERROR GotoRow (DBROWID RowID);
```

The result is to set the table's current-row to the given row. If the table is not already positioned at the given row, end-of-row processing will occur for the current-row before moving to the new row.

Read the Value in a Column

The intersection of a row and a column (often called a *field*) may contain a value or may have the special value Null (no value). The Table has a *current row*. Certain member functions of the Column class can be used to fetch the value in the column at the *current row*. The prototypes are:

```
DBINTEGER GetInteger(void);
DBREAL GetReal(void);
CDBSTRING GetString(void);
DBVALUE& GetValue(void);
COLATTRIBS GetType (void);
DBSIZE GetSize (void);
DBSIZE Size (void);
DBVALUE Get (void);
```

The member functions **GetInteger**, **GetReal**, and **GetString** have obvious meaning. They fetch the data from the field and convert it (if possible) to the return type. Note that when pointers or references are returned, as in **GetString**, these pointers point to data in a temporary buffer. The pointer is only valid until the next call to an RC21 function.

The member function **GetValue** returns a reference to a **DBVALUE** object. **DBVALUE** is a class designed to hold both the value and its type and size, and so is useful for fetching the value of **BLOB**s because we need to fetch both the value and its length.

GetType is used to determine the type of a field if it is not known. Note that it is not necessary to assign a type to a column, so the type of a particular field in that column may be unknown.

GetSize returns the number of bytes that would be required to store the returned value.

Get returns a **DBVALUE**. Any pointers in the **DBVALUE** object point to data that are owned by the object.

An additional way implicitly to fetch data from a field is to use the **ostream operator<<** which is prototyped:

```
ostream& operator << (ostream &o, Column& c);
```

Here is an example:

```
#include <rc21.h>
#include <iostream.h>

char *LongSpell (unsigned long num);

void main(void)
   {
   Remove ("test.db");
   Create ("test.db");
   Database test("test.db");
   test.AddTable ("newtab");
   Table newtab(test, "newtab");
```

```
newtab.AddColumn ("col1", Integer);
newtab.AddColumn ("col2", String);
Column col1(newtab, "col1"), col2(newtab, "col2");
// fill the table
for (long i=1; i<100; ++i)
   {
   newtab.AddRow();
   col1 << i;
   col2 << LongSpell(i);
   }
// retrieve the values
ForAllRows (newtab)
   {
   cout << col1;
   CDBSTRING s;  s << col2;
   cout << ' ' << s << endl;
   }
test.Commit();
}
```

Select (Find) a Particular Row in a Table

To find a row in a table that fulfills certain criteria, use the **Find or vFind** member f unctions of the **Table** class. These have the prototypes:

```
int Find (CDBSTRING filter ...);
int vFind (CDBSTRING filter, va_list ap);
```

The *filter* argument is a string that contains an infix expression that has operators and operands that resemble those in the C language. For a complete discussion of the syntax of filter expressions, see Selecting Rows In a Table - Filters on page 24.

The variable-length argument list in the **Find** function is to accommodate replaceable parameters that may be present in the *filter* argument. For example, we may have

```
table.Find ("LastName == 'Smith' && FirstName == 'John');
```

or we could have

```
table.Find ("LastName == %S && FirstName == %S, "Smith", "John");
```

The second function, **vFind**, is useful for encapsulating calls to the filter mechanism when an argument list may be passed through several layers of function calls.

The effect of the **Find** or **vFind** functions is to position the *current-row* of the table at the row that matches the criteria specified in the *filter* argument. If no such row is found, the *current-row* is not changed.

Select a Set of Rows in a Table

When it is desired to select more than one row in a table that match certain criteria, use the **Filter or vFilter** member functions. These have the prototypes:

```
DBERROR Filter (CDBSTRING f ...);
DBERROR vFilter (CDBSTRING f, va_list ap);
```

These have the effect of screening out all rows in a table that *do not* satisfy the criteria described in the *filter* argument. These functions do not alter the *current-row* of the table. Once the filter has been applied, simply traverse the table using the ForAllRows iterator macro. To remove the filter constraint, simply call **Filter** with a NULL argument. Note: filters may be stacked and the **Find** member function works on the subset of rows that satisfy the filters in effect. Example:

```
// examp203.cpp
#include <rc21.h>
#include <iostream.h>

void main(void)
   {
   Remove ("test.db");
   Create ("test.db");
   Database test("test.db");
   test.AddTable ("newtab");
   Table newtab(test, "newtab");
   newtab.AddColumn ("col1", Integer);

   for (long i=1; i<100; ++i)
      {
      newtab.AddRow();
      col1 << i;
      }

   // show all rows:
   ForAllRows (newtab)
      cout << col1 << endl;

   // apply filter and show again (only even rows)
   newtab.Filter ("col1 % 2 == 0");
   ForAllRows (newtab)
      cout << col1 << endl;

   test.Commit();
   }
```

Join a Table to Another Table

The SQL "join" operation is achieved in RC21 procedurally with nested loops containing **Filter** or **Find** operations. Please see a complete discusion of this topic on page 30.

Delete a Row from a Table

To delete the row at the *current-row* of a table, use the **DeleteRow** member function:

```
DBERROR DeleteRow(void);
```

The current row is deleted. The table cursor, however, remains at the deleted row. A **NextRow()** will be required to move to the next or previous row. Here is an example based upon the example previously used to illustrate the **Filter** member function:

```
// examp204.cpp - deleting rows from a table
#include <rc21.h>
#include <iostream.h>
```

```
void main(void)
   {
   Remove ("test.db");
   Create ("test.db");
   Database test("test.db");
   test.AddTable ("newtab");
   Table newtab(test, "newtab");
   newtab.AddColumn ("col1", Integer);
   Column col1(newtab, "col1");

   for (long i=1; i<100; ++i)
      {
      newtab.AddRow();
      col1 << i;
      }

   // show all rows:
   ForAllRows (newtab)
      cout << col1 << endl;

   // apply filter and delete even rows:
   newtab.Filter ("col1 % 2 == 0");
   ForAllRows (newtab)
      newtab.DeleteRow();

   // show all rows again:
   newtab.Filter(NULL);
   ForAllRows (newtab)
      cout << col1 << endl;

   test.Commit();
   }
```

Delete a Column from a Table

To delete a column from a table, use the DeleteColumn member function:

```
DBERROR DeleteColumn(CDBSTRING n);
```

Note that this is different from the **delete** operation on a **Column** object. Rather, it removes the database column from the database table. So, we could have

```
table.DeleteColumn("col1");
```

A side-effect of this operation is that any objects that refer to this column are marked invalid.

Delete a Table from a Database

To delete a table from a database, use the DeleteTable member function:

```
DBERROR DeleteTable(CDBSTRING name);
```

Note that this is different from the **delete** operation on a **Table** object. Rather, it removes the database table from the database. So, we could have

```
db.DeleteTable("table_1");
```

A side-effect of this operation is that any objects that refer to this table are marked invalid.

Erase the Data in a Field

The value in a field may be erased by using the PutNull function of the column class. The value at the *current-row* in the column is set to Null (no value).

```
DBERROR PutNull(void);  // store a Null (empty) value
```

Erase All the Data in a Column

To erase *all* the values in a column, you may either traverse the entire table, writing Nulls into the column's value with the PutNull function, or you may use the Erase member function:

```
void Erase (void);
```

as in

```
ForAllRows(table)
   column.PutNull();
```

or

```
column.Erase();
```

Erase All the Data in a Table

To delete all the rows in a table, use the Erase member function:

```
void Erase (void);
```

as in

```
table.Erase();
```

Fetch the Attributes of a Database

The attributes of a database may be fetched by using the following Database member functions:

```
CDBSTRING Path(void);
CDBSTRING Name(void);
CDBSTRING Release(void);
unsigned PageSize(void);
unsigned long HashSize(void);
int CachePages(void);
int Changed(void);
```

Both **Path** and **Name** return the same - the full path of the database file, to the extent it is available from the operating system. In some cases, full path qualification is not available.

The Release member function returns a string that represents the release number of the RC21 software that was used to create the database file. Please se a discussion of the meaning of "Release Numbers" on page 23.

The **PageSize** and **HashSize** return the parameters with which the database was created.

The **CachePages** function returns the parameters with which the database object was instantiated. In some cases this number is different from the Database constructor argument, for example, if the requested number of pages is not available.

The above functions are provided mainly for symmetry.

The **Changed** member function returns a nonzero value if the database has been changed since the last **Commit**.

Since a given database file must be accessed with a compatible release of the RC21 software it must be possible to determine the release number of the software without actually instantiating a Database object. This is accomplished by using the **Version** nonmember function:

```
CDBSTRING Version (CDBSTRING dbname);
```

Example:

```
// examp205.cpp - fetching database attributes
#include <rc21.h>
#include <iostream.h>

void main(void)
   {
   Remove ("test.db");
   Create ("test.db", 2048, 4096);
   Database test("test.db");


   cout << "Changed: " << test.Changed() << endl;
   cout << "Pages: " << test.CachePages() << endl;
   cout << "Path: " << test.Path() << endl;
   cout << "Name: " << test.Name() << endl;
   cout << "Release: " << test.Release() << endl;
   cout << "Page Size: " << test.PageSize() << endl;
   cout << "Hash Table Size: " << test.HashSize() << endl;
   }
```

Fetch the Attributes of a Table

The name, description, and view-expression (if any) of a table may be fetched by using the following Table member functions:

```
CDBSTRING Name (void);
CDBSTRING Description (void);
CDBSTRING View (void);
```

Change the Attributes of a Table

The name and the description of a table may be changed by using the member functions **Name** and **Description** with arguments:

```
DBERROR Name (CDBSTRING NewName);
DBERROR Description (CDBSTRING NewDescription);
DBERROR View (CDBSTRING NewView);
```

Fetch the Attributes of a Column

All the attributes of a column that were established when the column was created with the **AddColumn** function may be retrieved using the following Column member functions:

```
CDBSTRING Name (void);
CDBSTRING Description (void);
CDBSTRING Format (void);
COLATTRIBS Attributes (void);
DBCOL Reference (void);
```

Name(), **Description()**, and **Format()** return the name, description, and printf format specifier for the column.

Attributes() returns a **COLATTRIBS** containing information about datatype, indexing and so forth. See COLATTRIBS - Column Attributes on page 24.

DBCOL() returns a **DBCOL** pointer to a **Column** object for the reference column for this column (see Referential Integrity on page 24).

Change the Attributes of a Column

The name, description, and format of a column may be changed using the following:

```
DBERROR Name (CDBSTRING NewName);
DBERROR Description (CDBSTRING NewDescription);
DBERROR Format (CDBSTRING NewFormat);
```

The **COLATTRIBS** attribute of a column contains information about indexing. This is the only attribute that may be changed. The bits in the *NewAttributes* argument will be masked for the indexing bits and replace the corresponding bits in the attributes for the column. In this way, a column may be indexed after it is created. Then, if desired, it may be unindexed, saving the overhead of maintaining an index.

```
DBERROR Attributes(COLATTRIBS NewAttributes);
```

Traverse all the Tables in a Database

The **FirstTable**, **NextTable**, Database member functions can be used to walk through all the tables in a database much like **ResetRow** and **NextRow** are used to walk through all the rows in a table:

```
void FirstTable (void);
int NextTable (void);
DBTAB CurrentTable (void);
#define ForAllTables(_d) for (FirstTable(_d);NextTable(_d);)
```

FirstTable gets ready to walk through all the tables in the database. **NextTable** moves to the next table. **CurrentTable** creates a new Table object for the current table and returns a **DBTAB** pointer to it. So, we could have

```
Database test("test.db");
for (test.FirstTable(); test.NextTable();)
   {
   DBTAB t = test.CurrentTable();
   delete t;
   }
```

or we could use the provided macro ...

```
Database test("test.db");
ForAllTables (test)
   {
   DBTAB t = test.CurrentTable();
   delete t;
}
```

Here is an example:

```
// examp206.cpp - traversing all the tables
#include <rc21.h>
#include <stdio.h>
#include <iostream.h>

void main(void)
   {
   Remove ("test.db");
   Create ("test.db");
   Database test("test.db");

   // add some tables to the database
   for (int i=0; i<10; ++i)
      {
      char tablename[20];
      sprintf (tablename, "table_%2.2d", i);
      test.AddTable(tablename);
      }

   // now, traverse the database tables
   ForAllTables(test)
      {
      DBTAB t;
      t = test.CurrentTable();
      cout << t->Name() << endl;
      delete t;
      }
   }
```

Note that the **CurrentTable** member function returns <u>all</u> tables, even the special internal tables that RC21 uses to store the data dictionary. These special tables are easily recognized because their names start with underscore (_).

These member functions and the corresponding Table member functions for traversing all the columns in a table can be used to write generalized utilities that work on a database of unknown contents.

Traverse all the Columns in a Table

The Table member functions for traversing all the columns in the given table are:

```
#define ForAllColumns(_t) for (FirstColumn(_t);NextColumn(_t);)
void FirstColumn (void);
int NextColumn (void);
DBCOL CurrentColumn (void);
```

Here is an example that combines **ForAllTables** and **ForAllColumns**:

```
// examp207.cpp - traversing all the tables
#include <rc21.h>
#include <stdio.h>
#include <iostream.h>

void main(void)
   {
   Remove ("test.db");
   Create ("test.db");
   Database test("test.db");

   // add some tables to the database
   for (int i=0; i<10; ++i)
      {
      char tablename[20];
      sprintf (tablename, "table_%2.2d", i);
      test.AddTable(tablename);
      DBTAB t = new Table (test, tablename);
      // for each table, add some columns
      for (int j=0; j<10; ++j)
         {
         char columnname[20];
         sprintf (columnname, "column_%2.2d", j);
         t->AddColumn(columnname);
         }
      delete t;
      }

   // now, traverse the database tables
   ForAllTables(test)
      {
      DBTAB t;
      t = test.CurrentTable();
      cout << t->Name() << endl;
      // traverse all the columns in the table
      ForAllColumns(t)
         {
         DBCOL c = t->CurrentColumn();
         cout << "  " << c->Name() << endl;
         delete c;
         }
      delete t;
      }
   }
```

Work with EBLOBs

BLOBs are Binary Large Objects. BLOBs are a single, atomic entity that cannot be addressed in parts. EBLOBs (Extended Binary Large Objects), on the other hand, can be fetched and stored in parts. Both BLOBs and EBLOBs are values of a single field (cell, if you like) in a database table.

EBLOBs may be very large. They may be fetched and retrieved without requiring the allocation of memory to store the entire value. In fact, EBLOBs can be thought of as embedded files within a database.

EBLOBs support analogs to the standard Unix functions for manipulating files, **lseek**, **read**, and **write**. In addition, EBLOBs support the **inject**, and **extract** functions. This means you can insert a series bytes into an EBLOB stream or delete a series of bytes without having to reconstruct the entire EBLOB.

The field where an EBLOB resides within a table may be thought of as the handle of a file. To place an EBLOB in a field, use the form:

```
eblob *e = new eblob(f);
```

where *f* is a reference to a column or a pointer to a column. The field in the named column (at the current row) will be used as the handle for an EBLOB. If the current value of the field is Null, the eblob e will be a new, empty eblob.

To write data into an eblob, use

```
e->write (buffer, bytecount);
```

To read data at the current position, use

```
e->read(buffer, bytecount);
```

To seek to a specific location within the eblob, use

```
e->seek(address);
```

This may also be written

```
e->seek(address, seekfrom);
```

where *seekfrom* is one of the manifest constants, E_SEEK_SET, E_SEEK_CUR, and E_SEEK_END.

To set or reset a bookmark at the current position, use

```
e->mark("label");
```

To remove the bookmark, use

```
e->unmark("label");
```

To seek using a bookmark, use

```
e->seek(BOOKMARK("label"));
```

BOOKMARK is a helper class that resolves argument ambiguity.

NOTE: Bookmarks continue to point to original position after inject and extract operations.

To insert a sequence of bytes into the EBLOB at the current position, use

```
e->inject (buffer, bytecount);
```

To remove a sequence of bytes, use

```
e->extract (bytecount);
```

Like any other database operation, EBLOB operations are in suspense until a **Commit** operation is performed on the database.

Handle Error Conditions

RC21 provides a general error-handling mechanism that includes the ability to notify you if an error occurs. This is a sensible alternative to checking error codes after every RC21 operation. The capability is provided in the form of the SetErrorFunc function:

```
ERRORFUNC SetErrorFunc(ERRORFUNC arg);
```

Here is an example of its use:

```
#include <rc21.h>
void errorfunc (void)
   {
   PrintError (NULL);
   }

void main (void)
   {
   SetErrorFunc (errorfunc);

   // other operations ...
   }
```

For a more complete discussion of error-handling including structured exception handling, see the discussion later in this document.

RC21 Database Fundamentals

Structure Of The Database

An RC21 database can be viewed as a set of 2-dimensional tables.

Each table has a set of rows and columns.

The intersection of a row and a column is a field. Each field has a datatype and a value. A value can be an integer, a real, a string, or a counted-byte array (sometimes called a BLOB [Binary Large OBject]). A variation of BLOB is USER data, which have their own comparison rules. An additional type, EBLOB, can be thought of as an embedded file type with the usual file operations - read, write, seek - and some additional operations - inject, and extract.

Tables and columns have a number of attributes including name and description. An RC21 database may also have certain index structures that help it to find fields with certain values or rows with combinations of values. These index structures are maintained **automatically** by RC21.

RC21 stores variable-length data like strings and BLOBs according to two different schemes, *Shared* or *Nonshared* (the default scheme).

Shared data is stored in a value pool. All occurrences of a given string or BLOB are stored only once - in the value pool, and are pointed to by all the fields that have that value. To facilitate this scheme, a hash table is present in a RC21 database. The hash table allows RC21 to discover quickly whether a given string is already present in the value pool. Both the value pool and the associated hash table are inaccessible by ordinary means. That is, they are for the internal use of RC21. However, you need to be aware that they exist, for RC21 performance depends upon the appropriate matching of hash-table size and database size. This hash-table size is under the control of the user at the time he/she creates a new database.

Nonshared data is stored in a heap. There is a one-to-one correspondence between field-values and heap entries, such that, when a field is deleted, the heap entry is deleted - freeing space for reuse by RC21.

The schema for a database, that is, the names of all the tables and columns and their attributes are stored in the Data Dictionary. The Data Dictionary is really just a set of two tables named "_tables", and "_columns". _tables stores data about the tables in the database. _columns stores data about the columns in the database. These two tables can be manipulated just like any other tables. However, special functions are provided to make routine dictionary operations like adding tables and columns simple.

Release Numbers

Each software release of RC21 has associated a *release number*. The release number is of the form xx.yy.zz where xx is the Major Release, yy is the Minor Release and zz is the Sub Release. A Major Release is associated with a database file format. A Minor Release is associated with database operational features. A Sub Release is used to release bug fixes.

The release number of the software with which a given database file was created is stored in the file itself. A given release of software may only access database files with the same Major Release.

The release number of the software can be accessed with the **Release** function.

The release number of a database file can be accessed with the **Release(char* dbname)** function.

Types Of Data You May Store

There are six fundamental types of data that you may store using RC21. These are as follows:

DBINTEGER - 32-bit signed integer. If indexed, these are ordered by numerical value.

DBREAL - double-precision floating point. If indexed, these are ordered by numerical value.

DBSTRING - a null-terminated character string. There are two subtypes of DBSTRING: String and USTRING. If indexed, String data is ordered according to strcmp return values. USTRING data are ordered according to stricmp return values (case insensitive compare).

DBBLOB - an unstructured binary object containing one or more bytes; each byte may contain any value, including the 0 value; DBBYTES may be used to store almost any kind of data - C structures, video or audio images, pictures. DBBLOB data , if indexed, will be ordered by results of memcmp function return values (byte-by-byte compare).

DBUSER - this is like DBBLOB except it will be treated differently for the purpose of comparing with other DBUSER values and inserting in an index; the user may provide a compare routine that evaluates the order of any two DBUSER values and returns +1, 0, or -1.

DBEBLOB - an embedded file type.

When adding a new column to the database, you may specify one of the preceeding types using the keywords Integer, Real, String, Blob, User, or Eblob. When you do this, only the specified type of data may be stored in a column. If you do not specify a data type for a column, any type of data may be stored in that column. We can think of columns as being typed or untyped. Regardless of whether a column is typed, each individual field in that column has a type. The type for a field is simply the type of data that is stored in that field.

We store data in a database using any of a family of type-specific Put functions: PutInteger, PutReal, PutString, PutBlob, and PutUser. The EBLOB type is associated with a field by instantiating an eblob object associated with the given field. Seek and read operations are performed with the eblob object.

Likewise, we retrieve data from a field using any of a family of type-specific Get functions: GetInteger, GetReal, GetString, GetBlob, and GetUser. If the type of data stored in a field does not match the Get type, it will be converted if possible. For example, if we store a number using PutInteger, we may retrieve it using GetReal, or even GetString (a character-string representation of the integer value will be returned). However, it's more efficient and more accurate to Put and Get using symmetric routines. EBLOB objects have a read function.

When we retrieve data that cannot be returned on the stack - String, Blob, and User data, RC21 will return a pointer to that data. This pointer is valid until your next call to a RC21 function. Subsequent calls to RC21 may change the contents of memory at that location.

Get and put functions are somewhat simplified by overloading the << and >> operators. Thus, we may have something like:

```
column << "This is a test\n";
```

or

```
column << 15;
```

Names Of Tables And Columns

Every table and column in a database has a name. Names for tables and columns are strings that contain one or more characters. Thus, a name may contain as many characters as a string. In DOS, this means you could have a name that was thousands of characters long. However, this would probably not be convenient.

In RC21, we restrict the characters that may appear in a name. RC21 names may contain any of the upper or lower-case letters (a..z, A..Z), the digits (0..9), and the underscore (_). Names may not contain embedded spaces. Names should not begin with an underscore. Names that do begin with an underscore are reserved for the use of RC21. Names also may not begin with a digit. You may recognize these rules as the same as the rules for the C language. This was intentional.

The following are valid names Name, Address, Social_Security_Number, SocialSecurityNumber, SSNO, ssno (different from SSNO), AbCdEf123, etc.

The following are not valid names: _ItemNumber (starts with an underscore), 1BC (starts with a digit), ABC.def (contains a period, which is not a legal name character), qwert$yuiop (contains a $, which is not a legal name character).

All tables in a given table must have unique names. That is, no two tables may have the same name.

Columns within a table must have unique names. However, two different tables may have columns that share a name. For example, it is allowed to have a table named "Products" thas has a column called "Name" and another table called "Plants" with a table also called "Name".

A table name and a column name may be combined to refer to a specific column in a specific table by concatenating the table name and column name with a dot. So, we could have a column referred to by the name "Products.Name". Another column could be referred to by the name "Plants.Name". Such names are called *qualified column names*.

Manifest Objects

Expressions of the form

```
database[tablename]
```

are interpreted as references to table objects referring to the database table with the given name. Similarly, expressions of the form

```
table[columnname]
```

refer to an underlying column, and ...

```
database[tablename][columnname]
```

would also refer to a column. These are called manifest objects. Here is an example:

```
// examp101.cpp
#include <rc21.h>
#include <iostream.h>

void main(void)
   {
   Remove ("test.db");
   Create ("test.db");
   Database db ("test.db");
   db.AddTable ("newtab");
   cout << db["newtab"].Name() << endl;  // outputs "newtab"
   db["newtab"].AddColumn("newcol");
   cout << db["newtab"]["newcol"].Name() << endl; // outputs "newcol"
   }
```

One might be tempted to use something like db[“*tablename.columnname”*] to represent a reference to a column. This won’t work simply because the operator [] for a database *always* yields a reference to a table.

Attributes Of A Column

In addition to its name, a column has a description, a C format code, a reference column and other attributes called COLATTRIBS.

The description is simply descriptive information for a field. For example, we may have a column called SSNO that has the description "Social Security Number". The description field is used only for documentation.

The C format code is stored along with the column so that we may have an easy way to write a program that displays the contents of a database without knowing exactly what's in the database.

The reference column is a reference to another column, probably in another table, that contains valid field values for this column. You need not specify a reference column for a column, but if you do, RC21 will enforce the reference rule. That is, you may only store values that appear in the reference column. As an example, consider a table of employees that has a column called JobCode. There is another table in the database called Skills. This table has columns called Code, Description, BasePay, and so forth. The Code column contains all the valid job codes. An employee must have a job code that appears in this table (or possibly no code at all). Now, when we want to add an employee to our Employees table, we want to enter his or her JobCode. RC21 will automatically cross check in the Code column of the Skills table to make sure that a valid code is being entered. This is an aspect of what is called referential integrity.

COLATTRIBS is an aggregate of other information about the column. It tells us whether the column has an index, whether to store it as Shared or Nonshared, whether each value in the column must be unique, its data type, and whether we will allow null values in the field. For a thorough discussion of COLATTRIBS see the reference section of this manual.

The database may be created entirely using C program statements. There is also a command utility, pinnacle.exe, that you may use to create your database. More about that later.

Creating A Database

The operations required to define a database are:

- Create an empty database.
- Add tables to the database.
- Add columns to the tables.

To illustrate this we will create a simple database that contains only one table.

```
//--------------------------- FONEMAKE.CPP -------------------------------
#include "rc21.h"

void errorfunc(void)
   {
   PrintError("WHOOPS");
   exit (-1);
   }
```

```
void main (void)
   {
   SetErrorFunc(errorfunc);

   Remove ("fonelist.db");
   Create ("fonelist.db");
   Database db ("fonelist.db");
   db.AddTable ("PhoneList");
   Table fonelist (db, "PhoneList");
   fonelist.AddColumn ("Last", String);
   fonelist.AddColumn ("First", String);
   fonelist.AddColumn ("Address", String);
   fonelist.AddColumn ("Phone", String);
   fonelist.AddKey ("LastFirst", "+Last+First", Unique);
   db.Commit();
   }
```

Adding Tables to a Database

A table may be added to a database by using the AddTable member function of class database. This function specifies the name and description of the table and the *view expression* if this new table is a *view* rather than a base table (See VIEWS for more about this). An ordinary, or *base* table does not have a view expression. The normal form for AddTable is:

```
DBERROR AddTable(CDBSTRING name,
   CDBSTRING description=(CDBSTRING)"no description",
   CDBSTRING view = (CDBSTRING)"");
```

So, a table has a name and a description. Sometimes it is a *view* into another table, in which case it has a view expression. Once added, a table may be deleted. Thus, it is possible to have temporary tables in a database.

Adding Columns to a Table

A column may be added to a table by using the AddColumn member function of the table class. This function specifies the name of the new column, the description, the C format code, a *referential integrity* column, and a list of other attributes like data type that are contained in one 32-bit word.

```
DBERROR AddColumn(CDBSTRING name,
   COLATTRIBS attr=NOATTRIBS,
   CDBSTRING description=(CDBSTRING)"no description",
   CDBSTRING format=(CDBSTRING)"",
   const DBCOL reference=NULL);
```

Description is for documentation purposes.

Format need not be supplied. You may simply use "". However, supplying a C format code will make it possible to use certain utilities like QUICKTAB.

Reference is the name of another column or a column object that refers to another column that has valid values for this column. *Reference* may be supplied as NULL. If you supply the name

of another column or a DBCOL object, each time data is written into this new column, the validity of the value will be checked against the values in the referential integrity column.

The final parameter, *attributes*, is a collection of other attributes of the column. The attributes are specified as an integer expression of keywords or'd together to specify the various attributes. The attributes are grouped together into classes. You may only specify one value per class. The classes of attributes you may specify are depicted in the following table:

Datatype	Defaults to untyped - any type of value may be stored in any row
Integer	A 32-bit signed integer
Real	A double-precision floating point number
String	A zero-terminated sequence of characters
USTRING	A string where upper-case and lower-case characters are treated the same for the purpose of indexing, although the original case is preserved
BLOB	A counted array of bytes that may contain zeros
User	A special type of BLOB with user-defined sorting (ordering) rules
Indexing	Defaults to NonIndexed
Indexed	An index will be automatically maintained for this column.
NonIndexed	No index will be maintained (default).
Nulls	Tells whether missing values (nulls) are permitted. Defaults to nulls-permitted.
Nulls	This column may have null (missing) values (default).
NoNulls	This column *must* have a value in every row.
Unique/NonUnique	Tells whether each value in the column must be different from every other value. Defaults to uniqueness not required.
Unique	This column must have unique values for every row.
NonUnique	Values in the column need not be unique (default).
Sharing	Tells whether values for this column will be stored in the *heap*, or the *value pool*. Defaults to storing values in the heap
Shared	Values in this column will be stored in a shared value pool.
NonShared	Values in this column will be stored in a value heap (default).
Embedded Key Length	Used for index performance; defaults to 0.
KeyLen(*key length*)	If there is an index for this column the first *key length* bytes of the value will be embedded in the index; default *key length* is zero. This affects performance only - not the size of a key value.

Here are some examples:

```
t.AddColumn (“Column0”);    // no type, description specified
t.AddColumn ("Column1", Integer, "A New Column", "%ld");
t.AddColumn ("Column2", String, "A New Column", "%-20s");
t.AddColumn ("Column3", USTRING | Indexed | KeyLen(4), "A New Column", "%-20s");
t.AddColumn ("Column4", Integer, "A New Column", "%ld", "Tab1.Codes");
t.AddColumn ("Column5", String | NoNulls, "A New Column", "%-20s");
t.AddColumn ("Column6", Blob | Indexed, "A New Column", "");
t.AddColumn ("Column7", User | Indexed, "A New Column", "%s");
```

Once added to a table a column may be deleted. Thus, you may have temporary columns in a table.

Writing Data Into The Database

Now that we have created a database, we may write data into it. To write data, simply add a row to a table with the AddRow function. The row just added now has no values in it. To store data in each of the columns in the row, use the *put* functions or use the << operator.

```
//--------------------------- FONEADD.CPP --------------------------------
#include <iostream.h>
#include <string.h>
#include "rc21.h"

void errorfunc(void)
   {
   PrintError("WHOOPS");
   exit (-1);
   }

void main (void)
   {
   SetErrorFunc(errorfunc);

   Database db ("fonelist.db");
   Table Fonelist (db, "PhoneList");
   Column Last(Fonelist, "Last"), First(Fonelist, "First"),
      Address(Fonelist, "Address"), Phone(Fonelist, "Phone");

   for (;;)   // exit on ctl-c
      {
      Fonelist.AddRow();
      cout << "\nLast Name: ";
      cin >> Last;
      cout << "First Name: ";
      cin >> First;
      cout << "Address: ";
      cin >> Address;
      cout << "Phone: ";
      cin >> Phone;
      db.Commit ();
      }
   }
```

Here is a simple program to store the numbers from 1 to 50 and their square roots:

```
// examp901.cpp
#include <rc21.h>
#include <math.h>

void main (void)
   {
   Create ("numbers.db");
   Database db ("numbers.db");
   db.AddTable ("Numbers");
   Table t (db, "Numbers");
   t.AddColumn ("Number", Real);
   t.AddColumn ("SQRT", Real);
   Column c1 (t, "Number");
   Column c2 (t, "SQRT");


    DBREAL i;
   for (i=0; i<50; ++i)
      {
      t.AddRow();
      c1 << i;
      c2 << sqrt(i);
      }

   db.Commit();
   }
```

So, in general, to write data in the database, we add a row to a table then use one of the *Put* member functions - **PutInteger**, **PutReal**, **PutString**, **PutBlob**, or **PutUser**. We may *modify* the contents of a field by positioning ourselves at a row, and using one of the Put functions to replace the value. Use the polymorphic **Put** function to simplify things. For the absolute maximum in simplicity, use the form *column* << *value*.

Reading Data From the Database

We put data into a database so we can get it back, right? We read data from an RC21 database by positioning ourselves at a row in a table using a sequence of ResetRow, NextRow function calls. *Table*.ResetRow positions us before the first row of a table. *Table*.NextRow moves to the next row, returning TRUE if there is a next row, and a FALSE if we have reached the end of the table. The *table*.NextRow function takes a +1 or -1 argument to indicate whether to move forward (+1) or backward (-1) through the table.

Data are fetched from named columns using the Get functions. These functions are **GetInteger**, **GetReal**, **GetString**, and **GetBlob**. As a simplification, you may simply convert the column to the appropriate type. If the type of data stored is not the same as the requested Get type, a conversion will be performed if possible. If a conversion is not possible, an error will be set in the global error code, DB_Error.

Here is a program that reads and prints the data from the list of numbers and square roots:

```
#include <rc21.h>
#include <math.h>

void main (void)
   {
```

```
   Database db ("numbers.db");
   Table t (db, "Numbers");
   Column c1 (t, "Number");
   Column c2 (t, "SQRT");

   ForAllRows(t)
      cout << DBREAL(c1) << ' ' << DBREAL(c2) << ' ' << endl;
   }
```

Finding Things In The Database

We find data in the database by specifying a *filter expression* and applying it to a table using the Filter member function. A filter expression is a string that resembles a C expression but with the names of columns instead of the names of variables. We then go through all the rows in a table and the rows that pass through the filter are "found".

Consider the table we've been working with. Let's say we want to find all the even numbers in the first column. Here's the program:

```
// examp906.cpp
#include <rc21.h>
#include <math.h>

void main (void)
   {
   Database db ("numbers.db");
   Table t (db, "Numbers");
   Column c1 (t, "Number");
   Column c2 (t, "SQRT");

   t.Filter("Number%2==0");
   ForAllRows(t)
   cout << DBREAL(c1) << ' ' << DBREAL(c2) << ' ' << endl;
   }
```

ForAllRows is a macro that substitutes for the ResetRow, NextRow sequence.

If there is only one row that would satisfy the search criteria, the Find member function may be used. Find locates any one row that satisfies the filter expression. Let's say we want to find the row where the value in *Numbers* is 25:

```
// examp903.cpp
#include <rc21.h>

void main (void)
   {
   Database db ("numbers.db");
   Table t (db, "Numbers");
   Column c1 (t, "Number");
   Column c2 (t, "SQRT");

   t.Find("Numbers == 25");
   cout << DBREAL(c1) << ' ' << DBREAL(c2) << ' ' << endl;
   }
```

Modifying Values In The Database

Modifying a field is just like writing a field the first time. We simply use one of the Put functions or operators to put a new value. To illustrate, let's take our table of numbers and change all the odd numbers.

```
// examp904.cpp
#include <rc21.h>
#include <math.h>

void main (void)
   {
   Database db ("numbers.db");
   Table t (db, "Numbers");
   Column c1 (t, "Number");
   Column c2 (t, "SQRT");

   t.Filter("Number%2==1");
   ForAllRows(t)
         {
         c1 << DBREAL(c1)*2;
        c2 << sqrt(DBREAL(c1));
        }
   t.Filter(NULL);
   ForAllRows(t)
      cout << DBREAL(c1) << ' ' << DBREAL(c2) << endl;
   }
```

Deleting Rows

We can add rows to a table. We can delete rows. Let's delete all the odd rows from our table.

```
// examp905.cpp
#include <rc21.h>
#include <math.h>

void main (void)
   {
   Database db ("numbers.db");
   Table t (db, "Numbers");
   Column c1 (t, "Number");
   Column c2 (t, "SQRT");

   t.Filter("Number%2==1");
   ForAllRows(t)
      t.DeleteRow();
   t.Filter(NULL);
   ForAllRows(t)
      cout << DBREAL(c1) << ' ' << DBREAL(c2) << endl;
   }
```

The Delete member function deletes the current row in the table.

Indexes and OrderBy

When a column is created it may be declared to be *indexed* by specifying the keyword Indexed in the attributes argument of the call to *table*.AddColumn member function. The effect is that a btree index will be created and automatically maintained for all the values in the column. Later,

the *table*.OrderBy function may be called to indicate that access to the rows in the table will be in the order of the values in that column.

Here's a program that creates a column of pseudo-random numbers, then prints them out in ascending order, then in descending order.

```
// examp907.cpp
#include <iostream.h>
#include <rc21.h>

void main (void)
   {
   Remove ("test.db");
   Create ("test.db");
   Database db("test.db", "rw", 20);
   db.AddTable ("table");
   Table t(db, "table");
   t.AddColumn("column", Integer | Indexed);
   Column c(t, "column");

   // add 100 rows to the table and put in random values.
   DBINTEGER i;
   for (i = 0; i < 100; ++i)
      {
      t.AddRow();
      c << DBINTEGER(rand());
      }

   // now, print out the values in natural order
   cout << "NATURAL:" << endl;
   ForAllRows (t)
      cout << DBINTEGER(c) << ' ';

   // now, print out the values in ascending order
   cout << "ASCENDING:" << endl;
   c.OrderBy();
   ForAllRows (t)
      cout << DBINTEGER(c) << ' ';

   // now, print out the values in descending order
   cout << "DESCENDING:" << endl;
   ForAllRowsBackwards (t)
      cout << DBINTEGER(c) << ' ';
   }
```

There are several things to note here. First, note that the Table ordering is controlled by a Column member function. The semantics is that the Table associated with the Column is now ordered by the Column. This may also be accomplished by using *Table*.OrderBy("column_name");

Next, note that the OrderBy remains in effect until it is replaced by another OrderBy or a *Table*.Unordered().

Finally, note the introduction of the ForAllRowsBackward macro. This is just a variation of the ForAllRows macro. The definition for the macro is in the file.

Composite Keys

Composite keys are virtual columns that allow you to access data in the order of the values of more than one column. An obvious example is the phone book. The phone book table contains four columns: LastName, FirstName, Address, and PhoneNumber. We want to list our phone number by FirstName within LastName. We do this by creating a composite key called LastFirst that is made up of LastName and FirstName. Here is a program that creates the database:

```
//--------------------------- FONEMAKE.CPP -------------------------------
#include "rc21.h"

void main (void)
   {
   Remove ("fonelist.db");
   Create ("fonelist.db");
   Database db ("fonelist.db");
   db.AddTable ("PhoneList");
   Table fonelist (db, "PhoneList");
   fonelist.AddColumn ("Last", String);
   fonelist.AddColumn ("First", String);
   fonelist.AddColumn ("Address", String);
   fonelist.AddColumn ("Phone", String);
   fonelist.AddKey ("LastFirst", "+Last+First", Unique);
   db.Commit();
   }
```

The arguments for the AddKey member function are *key-name*, *key-list*, and *attributes*. The key-list argument is a list of column names each preceeded by either a + sign or a - sign. The plus indicates ascending order. The minus indicates descending order. In the example above, we will have an index in ascending order of LastName and FirstName. An index may contain any combination of other column names with any combination of signs, but the columns must be real columns and not composite keys themselves. The attribute argument may contain the keyword *unique* to indicate that there must be no other key entries with this value. That is, to take our phone list as an example, that there must be no other identical combination of LastName and FirstName. If you attempt to store a value that would be non-unique, an error condition will be created, although the data will be stored anyway. This implies that you should check the global error flag, DB_Error , after you have complete this operation.

Note that composite keys, like all other indexes are maintained automatically by RC21. You do not have to build a key entry or anything like that. Simply by storing the data with PutString in the columns LastName and FirstName, the composite key will be maintained. This raises a question: when does the key get built? Does the key get built when we store LastName or when we store FirstName? Or does it get rebuilt as we store each value? The answer is that the composite key gets built when we attempt to move to another row. We move to another row with the next AddRow, Find, ResetRow, or NextRow as well as with several other function calls. The question, then, is how do we know when to check the DB_Error flag for a unique violation? The answer is - we can force the composite key to be built and stored by using the *Table*.FinishRow() function. FinishRow will do all the composite key building and unique checking associated with operations on a given row. So, we can check for errors before we move away from the row. Thus, we will have a chance to correct our mistakes if necessary. Note that

it not necessary for you to call FinishRow to cause the composite key to be built. Its only use is to force this operation *before* you move to another row, possibly complicating error correction.

Substring Keys

A derivation of composite keys is substring keys. Substring keys, like composite keys, comprise a series of strings. In substring keys, you specify a subset of the characters in each component of the key. For a substring key, you specify the beginning character (first character is designated "0") and the number of characters that are to be considered in each component. Here is an example:

```
t.AddKey ("key", "+field1:0:5+field2:3:4");
```

The first five characters of field1 and the fourth through the seventh characters of field2 are the parts of the above key.

Objects, Handles, and Manifest Objects

The common objects in RC21 are **Table** and **Column.**

There are three ways to refer to a Table or Column object:

Direct Reference - define the object:

```
Table a(db, "Table1");
```

Pointer Referral - use the builtin pointer-type (or *handle*) DBTAB, or DBCOL and the **new** operator:

```
DBTAB t = new Table(db, "Table1");
```

Manifest Referral - refer to the table or column using the special syntax for m*anifest objects*.

```
db["Table1"]
```

to refer to a table, or

```
db["Table1"]["Column1"]
```

to refer to a column

Object Confusion

It is possible to have several objects that refer to the same underlying database table. For example:

```
DBTAB t = new Table(db, "Table1"), t2  = new Table(db, "Table1");
Table t1(db, "Table1");
t->Find ("Column1 == 1");
t1.Find ("Column1 == 2");
```

```
t2->Find ("Column1 == 3");
db["Table1"].Find("Column1 == 4");
```

All the objects are positioned at different rows within the (underlying) table. Therefore, the different objects are not interchangeable.

Retrieving and Modifying Database, Table and Column Properties

Databases, Tables and columns aquire names, descriptions, and other attributes when we add them to the database. These properties may be retrieved by using a set of RC21 nember functions. Here are the functions:

int *Database*.Changed(void);	returns nonzero if database has been changed
int *Database*.CachePages(void);	returns the number of cache pages allocated at open
CDBSTRING *Database*.Path(void);	returns the path of the database file
CDBSTRING *Database*.Name(void);	same as path()
CDBSTRING *Database*.Release(void);	returns the release number in the form xx.yy.zz
Unsigned *Database*.PageSize(void);	returns the size of one database page established a Create
Unsigned long *Database*.HashSize(void);	returns the number of elements in the hash table
CDBSTRING *Table*.Name(void);	returns the name of a table
CDBSTRING *Table*.Description(void);	returns the description of a table
CDBSTRING *Table*.View(void);	returns the view expression of a table that is a view
CDBSTRING *Column*.Name(void);	returns the name of a column
CDBSTRING *Column*.Description(void);	returns the description of a column
CDBSTRING *Column*.Format(void);	returns the printf format stored with a column
DBCOL *Column*.Reference(void);	returns handle to associated reference column for this column
COLATTRIBS *Column*.Attributes(void);	returns the other attributes stored in a COLATTRIBS word

The following program illustrates the use of some of these functions:

```
// examp906.cpp
#include <rc21.h>

DEFT_ERROR

void main (void)
{
   SET_DEFT_ERROR;

   Database db("fonelist.db", "rw", 25);
```

```
    Table tab(db, "PhoneList");

    cout << db.Path() << " - Created by version "
       << db.Release() << " of RC21." << endl;
    cout << " Hash table size: " << db.HashSize() << endl;
    cout << " Page size: " << db.PageSize() << endl;
    cout << " Database opened with " << db.CachePages() <<  " pages."
       << endl;
    cout << "Table: " << tab.Name() << " - " << tab.Description() << endl;

    ForAllColumns(tab)
       {
       cout << "Column: " << tab.CurrentColumn()->Name();
       cout << " - " << tab.CurrentColumn()->Description();
       cout << " Format: " << tab.CurrentColumn()->Format();
       cout << endl;
       }
}
```

This program will produce the following text on stdout:

```
fonelist.db - Created by version 7.1 of RC21.
 Hash table size: 512
 Page size: 512
 Database opened with 25 pages.
Table: PhoneList - PhoneList
Column: _status - status of row in table Format: %1d
Column: Last - no description Format:
Column: First - no description Format:
Column: Address - no description Format:
Column: Phone - no description Format:
Column: LastFirst - +Last+First Format:
```

Modifying Table and Column Properties

Once a table or column has been set up you may have a need to change some of its attributes. For example, you may wish to change the name or description of a table or a column. The following functions help you to do this:

DBERROR *Table*.Name(CDBSTRING NewName);	changes the name of a table
DBERROR *Table*.Description(CDBSTRING NewDescription);	changes the description of a table
DBERROR *Table*.View(CDBSTRING NewView);	changes the view expression of a table that is a view
DBERROR *Column*.Name(CDBSTRING NewName);	changes the name of a column
DBERROR *Column*.Description(CDBSTRING NewDescription);	changes the description of a column
DBERROR *Column*.Format(CDBSTRING NewFormat);	changes the printf format stored with a column
DBERROR *Column*.Attributes(COLATTRIBS NewAttributes);	changes the other attributes stored in a COLATTRIBS word

Sorting the Rows in a Table

From time to time it may be necessary to access the rows in a table in the order of the contents of some column. This is normally achieved by declaring that column to be *Indexed* and then doing an **OrderBy** of that column. Then, the rows in the table will be accessed in ascending or descending order by that column. However, let's say that the column in question is *not* indexed. This is easily remedied by changing the attributes of the column so that it *is* indexed. Then you may perform whatever operations on the table are desired. Afterwards, you may change the column back to NonIndexed. This sequence of operations is achieved by using the **Attributes** function with the *Indexed* attribute set. Later, you may revert to non indexed by setting the *NonIndexed* attribute for the column. Alternatively, you may simply use the **Index** or **Uninex** Column member functions.

If you find yourself doing this frequently for a given column you should ask yourself if you should just leave the column indexed. After all, each time you change this attribute to *Indexed*, RC21 must build a complete index. However, if you need the index only once, it is better to remove the index so as to avoid the overhead of maintaining an unnecessary index.

COLATTRIBS - Column Attributes

COLATTRIBS is a typedef for a word that contains attributes for a column. A COLATTRIBS word contains information about the following attributes:

Data type - the type column may be any of the following: Integer, Real, String, USTRING, Blob, User, or Eblob. It may also be unspecified, in which case any type of data may be stored in that column. If a data type is specified, only data of that type may be stored in the column. A reserved type, Key, is used by RC21 to identify composite key columns. The Key attribute may not be specified by the user.

Sharing - data in a column will be stored in a value pool or in a heap depending upon the value of the Sharing attribute. The values are Shared or NonShared. Shared data will be stored in a value pool. Only one copy of a data value will be found in the pool. All fields that share this value will reference it. When a Shared value is written to the database, it must first be located, if possible, in the value pool. This is accomplished by hashing-and-chaining. When a Shared value is deleted, the corresponding entry in the value pool is not deleted. This is because another field may be referencing this value. NonShared data is stored in a heap. There is a one-to-one relationship between heap values and NonShared fields. When the value of a NonShared field is deleted or changed, the corresponding entry in the heap is freed. When an entire page of heaped data is freed, the page is returned to the PINNACLE filespace manager and may be reused. Thus, the advantages of NonShared data are twofold: no chaining necessary to store data; deleted data frees up space. You should choose the Shared or NonShared attributes depending upon your understanding of the nature of the data in the column. If neither Shared or NonShared is specified, Shared is the default attribute.

Nulls - the two values for this attribute are Nulls and NoNulls. Nulls implies that null values are permitted. NoNulls implies that null values are not permitted. If a column has the attribute NoNulls, any attempt to store a null value in a field by the **PutNull** member function or by failure to give a value to a field in a new row after the **AddRow** function will produce an error. Nulls is the default attribute.

Unique - this attribute may have the values Unique and NonUnique. Columns with the Unique attribute will never have two different rows with the same value. An attempt to store a value that would violate this assertion will result in an error.

Indexing - this attribute may have the values Indexed and NonIndexed. Indexed columns will have an automatically-maintained index so that rows in the table may be retrieved in ascending or descending order depending on the value of this column. This is accomplished with the **OrderBy** function. Also, fast access to a rows with the **Find** function or to a set of rows with the **Filter** function is possible. NonIndexed columns have no index. **Find**s and **Filter**s will require linear searching of the column. **OrderBy** will result in chronological access.

Keywords are defined in **RC21.H** for each of these attributes. In addition, mask keywords are defined so that you can retrieve the value of an attribute for a specific class.

Attributes may be combined in a COLATTRIBS word by or'ing together the attribute keywords. The following, for example, is a valid COLATTRIBS expression:

```
String | Indexed | NoNulls | Unique | NonShared
```

Attributes within a TYPE may not be or'd together. Thus, the following is not a valid COLATTRIBS expression:

```
String | Integer | NoNulls | Unique | NonShared
```

Attributes may be and'd with the class masks to yield the attribute value for a given class. For example, the expression

```
(attributes & IndexMask) == Indexed
```

would yield the value TRUE if attributes contained the Indexed attribute. Or, you could write

```
 (attributes & Indexed)
```

which would yield a non-zero value if attributes contained the Indexed attribute.

A COLATTRIBS expression is used as an argument to the **AddColumn** function when you are adding a column to a table. If you desire not to specify an attributes, and just allow them to default, the value 0 may be used. This has been #define'd to the keyword NOATTRIBS.

Attributes of a column may be subsequently retrieved by the **Attributes** member function. These attributes may be displayed using the **DB_DecodeAttributes** function. The inverse of the the

DB_DecodeAttributes function is the **DB_EncodeAttributes** function, which encodes a character string into a COLATTRIBS word

Some of these attributes may be changed by the **Attributes** function with a non-void argument. Some attributes may not be changed. An attempt to change non-changeable attributes will result in an error. The only attribute that is changeable is the Indexing attribute.

Referential Integrity

Consider the following problem: you have a Vendor table, and a Purchases table. Each row in the Vendor table has a vendor number, name, address, phone number, and so forth. In the Purchases table each row has date, vendor number, item, quantity, price, and so forth. It is essential that the vendor code in the Purchases table corresponds to some valid vendor number in the Vendor table. Guaranteeing that this is the case is an example of what is called "referential integrity".

RC21 supports referential integrity in a simple way. When you add a column to a table you may define a *referential-integrity column* that is to be associated with that column. To use the example in the previous paragraph, we would create the column in the Purchases table as follows:

```
Table t1(db, "Purchases");
table.AddColumn("Vendor", Integer, "Vendor Code", "%4ld", "Vendor.Number");
```

The last argument, "Vendor.Number" specifies the table and column for referential integrity.

When a referential-integrity column exists for a column, RC21 will attempt to ensure that only permitted values will be stored. An error condition will be raised if your program attempts to violate the referential-integrity constraint, and the value will not be stored. However, the row where you attempted to store the data will continue to exist. You must decide whether to delete the row, store another value in the column, or whatever is appropriate.

The Database Classes

Database Types

Selecting Rows In a Table - Filters

A filter is an expression that evaluates to a **TRUE** or **FALSE** value (nonzero or zero respectively) when applied to a row in a table. In other words, it is a *predicate*. It is similar in syntax to a C language expression. There are *operators*, which may be relational or arithmetic operators, and parentheses, and *operands* which may be constants, or column-names.

Filter Application

A filter may be applied to a Table object by invoking theFilter member function as in the following example:

```
Database db("employee.db");
Table tab(db,"EMPLOYEES");
tab->Filter("JOBCODE = 5 || JOBCODE == 7");
tab->OrderBy ("LASTFIRST");

cout << "All employees with jobcode 5 or 7:\n" << endl;
ForAllRows (tab)
   cout <<  CDBSTRING(tab["FirstName"]) << CDBSTRING(tab["LastName"]) << endl;
```

The example above shows how a filter causes only certain rows for which the filter expression is TRUE to be retrievable. In this case, the filter examined the column named "JOBCODE" and tested for certain values.

Filter Operators

The operators permitted in a filter expressions include the following:

Arithmetic operators: +, -, *, /, %
Relational operators: ==, !=, >, <, >=, <=
Wildcard match operators: ?, ~= (these are equivalent), and ?* (case-insensitive).
Wordsearch operator: ??
Substring operator: ?>
Parentheses: (,)

Filter Operands

The operands that may be used in a filter expression are:

- Column name - no quotes
- Strings surrounded by single quotes; strings may include escaped characters preceeded by backslash (\)
- Numbers - all numbers start with a digit; if the first digit is 0 and this is followed by a 'B' or a 'b', the number is a binary number and may contain only 0's or 1's; if the first digit is 0 and this is followed by an 'X' or an 'x', the number is a hexadecimal number and may contain only the hexadecimal digits 0-9, a-f and A-F; if the first digit is zero and this is not a binary or a hexedecimal number, it is taken to be an octal number and may contain only the octal digits 0-7; sign extension is not performed. All other numbers are either integers or reals. If only digits are present in the number, it is an integer. If other characters acceptable to scanf are present, it is a real number.
- Replaceable Parameters - %R, %r, %I, %i, %S, %s, %B, %b - which are replaced with the actual values of the corresponding arguments in the argument list of a call to any of the functions that have filter-expression as an argument

Filter Semantics

A filter expression, when applied to a row in a table, produces a TRUE or FALSE value. Here are the semantic rules:

- There are binary expressions and unary expressions; binary expressions have the form *term1* o*perator term2* Unary expressions have the form *operator term1*
- Parentheses cause the evaluation of the expression inside parentheses as a term
- *term1* + *term2*: add *term1* to *term2*
- *term1* - *term2*: subtract *term2* from *term1*
- *term1* * *term2*: multiply *term1* by *term2*
- *term1* / *term2*: multiply *term1* by *term2*
- *term1* % *term2*: moduls of *term1*, *term2*
- *term1* > *term2*: 1 if *term1* is greater than *term2*, otherwise 0
- *term1* < *term2*: 1 if *term1* is less than *term2*, otherwise 0
- *term1* == *term2*: 1 if *term1* is equal to *term2*, otherwise 0
- *term1* >= *term2*: 1 if *term1* is greater than or equal to *term2*, otherwise 0
- *term1* <= *term2*: 1 if *term1* is lexx than or equal to *term2*, otherwise 0
- *term1* != *term2*: 1 if *term1* is not equal to *term2*, otherwise 0
- *term1* ? *term2*: 1 if *term1* matches the wildcard expression *term2* (see below), otherwise 0
- *term1* ?* *term2*: 1 if *term1* matches the wildcard expression *term2* (see below), otherwise 0; matching is case-insensitive
- *term1* ?? *term2*: 1 if *term1* satisfies the wordsearch string,*term2 (see below)*, otherwise 0
- *term1* ?> *term2*: 1 if *term1* contains the substring *term2*, otherwise 0; substring tests are case-insensitive
- !*term1*: 1 if *term1* is zero; 0 if *term1* is nonzero
- -*term1*: the negative value of *term1*
- after evaluation of the filter expression, a non-zero value is considered TRUE, a zero value is considered FALSE

Filter Examples

```
"LastName == 'Smith'"
"LastName == 'Smith' && FirstName == 'John'"
"Age > IQ"
"Salary > 4*Mortgage && Salary > 25000"
"LastName ? 'Mc*' || LastName ? 'O\'*'"
```

Wildcard Patterns

A wildcard pattern comprises a sequence of wildcard phrases. A wildcard phrase is one of:f

- Any non-null character except asterisk (*), question mark (?), left brace ([), left bracket ({), or circumflex (^). These *ordinary* characters are matched literally.
- An asterisk to match an arbitrary string.
- A question mark to match any signal character.

- A list of wildcard patterns enclosed within braces and separated by commas. This matches any of those wildcard patterns.
- A list of characters or ranges of characters (like "a-z" for "a" to "z") enclosed within square brackets and with no separators except a hyphen between the two characters of a range. The list may be preceeded with a circumflex (^) to match any character BUT the specified ones.

Wildcard Examples

"*"	matches anything.
"a*z"	matches "a" folowed by 0 or more characters followed by "z".
"a?c"	matches "aac", "abc", "acc", "adc", and so forth.
"{if, and, but}"	matches any of "if", "and", or "but"
"[a-f,z]*"	matches any string that starts with "a", "b", "c", "d", "e", "f", or "z".
"[^x]*"	matches any string that doesn't start with an "x".
"\[*]"	matches anything within square brackets. The \ means the next character is quoted. Use the \ just like you would in C.

Wordsearch (Lexical) Operations

The operator "??" indicates a wordsearch operation. The wordsearch operation permits the scanning of text fields for the presence or absence of *words*. The wordsearch operator is much more convenient than the wildcard operator for this sort of application.

A *word* is defined as a sequence of characters comprised entirely of digits, upper or lower case letters, underscores, apostrophes, and dashes. Words do not include whitespace or punctuation.

A *phrase* is defined as a sequence of words.

Wordsearch Operators comprise the following: && (and), || (or), ! (not), () (parentheses), [] (phrase brackets].

The wordsearch operator takes as an operand a string that contains a wordsearch expression. The wordsearch expression consists of words and operators.

```
'Kentucky && racing'
'Kentucky || racing'
'!Kentucky && racing'
'Senate && (welfare || education)'
'[Abraham Lincoln] && abolition'
'O\'Bryan || O\'Malley'
```

Precedence from highest to lowest: [], (), !, &&, ||.

Now, consider that we have a table like this:

TABLE:	LibraryOfCongress	
COLUMN:	CatalogNumber	string
COLUMN:	Title	string
COLUMN:	Author	string
COLUMN:	Text	string
COLUMN:	Illustrations	blob

To find all books in the LC about Abraham Lincoln and abolition, you could use something like:

```
Table tab(db, "LibraryOfCongress");
tab.Filter (db["LibraryOfCongress"], "Text ?? '[Abraham Lincoln] && abolition'");
ForAllRows (tab)
   {
   cout << CDBSTRING(tab["CatalogNumber"]);
   cout << ' ' << CDBSTRING(tab["Title"]);
   cout << ' ' << CDBSTRING(tab["Author"]);
   }
```

The Lex function may be used to determine if a string matches a lexical pattern. See the reference section for further information.

Applying A Filter To A Table

A filter is applied to a table by use of the Filter member function. A filter stays with the table until it is removed by used of the Filter function with a NULL argument. If a filter is applied to a table for which there is *already* a filter in effect, the new filter is stacked on top of the previous filter. The logical effect of stacked filters is as though they were and'd together with &&. So, the following sequence:

```
table.Filter ("JOBCODE = 5 || JOBCODE == 7");
table.Filter ("SEX == 'M'");
```

would produce an effect as though we had written:

```
table.Filter("(JOBCODE = 5 || JOBCODE == 7) && SEX == 'M'");
```

When a filter has been applied to a table, only the rows for which the filter expression is TRUE are accessible.

Removing A Filter From A Table

A filter stays with a table for the duration of the program or until it is removed.

The filter is removed from a table by applying a filter value of NULL or a zero-length string as in the following examples:

```
table.Filter ("");
```

or

```
table.Filter (NULL);
```

So, to continue the previous example:

```
table.Filter ("JOBCODE = 5 || JOBCODE == 7");
table.Filter ("SEX == 'M'");
// now, we have JOBCODE 5 or 7 and SEX == 'M'
table.Filter (NULL);
// now, we have JOBCODE 5 or 7
table.Filter (NULL);
// now, we have no filter
table.Filter (NULL);
//   the last DB_Filter produced an error because there is
//   to filter to remove.
```

The Find Function and Filter Expressions

Filter expressions may be used with the Find member function to simplify searching. The Find function accepts one or more arguments - the filter expression and zero or more replaceable arguments (more about that later).

```
Employees.Find ("First == 'John' && Last == 'Smith'");
```

Replaceable Elements in Filter and Find

Filter expressions may contain replaceable elements resembling those in *printf* arguments. Both Find and Filter Table member functions accept variable-length argument lists. The filter expression itself may contain substrings that start with the percent sign (%) and followed by one of the following letters: R, I, S, B. The meaning of these is as follows:

%R	replace with a DBREAL argument from the argument list
%I	replace with a DBINTEGER argument from the argument list
%S	replace with a DBSTRING argument from the argument list
%B	replace with a DBBLOB argument from the argument list; a DBBLOB argument really has two values - a pointer to the blob value followed by a DBSIZE that has the size of the blob

Thus, the following:

```
char *lastname = "Smith", *firstname = "John";
Employees.Find ("Employees", "LastName == %S && FirstName == %S", lastname,
firstname);
```

would have the same effect as

```
Employees.Find ("Employees", "FirstName == 'John' && LastName == 'Smith'");
```

This feature makes the filter mechanism easier to use than it would be if you had to use the *sprintf* function to embed your arguments in the filter expression. Also, it is more efficient to use this form in a loop because the filter expression is parsed and partially optimized only the first time it is seen by the filter functions in RC21.

Complex Filter Expressions

Filter expressions may be arbitrarily complex. You may combine the relational operators, <, >, <= , >=, ==, and !=. You may use the wildcard match operator ? and the lexical operator ??. You may use the logical || && and !. You may use the arithmetic operators +, -, *, /, and %, and you may use the grouping operators (and). Thus, if you wanted to find all the employees whose first name was equal to their last name and whose salary in thousands was equal to their age, you might write something like:

```
Employees.Filter ("LastName == FirstName"
"&& Salary / 1000 == (90 - DOB)");
```

Note that although the parentheses are shown in this example, they are not necessary because the C rules of precedence apply.

Optimizations

When possible, RC21 will use existing index structures to reduce disk access and processing. Currently, RC21 does not use composite key indexes that have substring segments. Also, expressions that have ORs at the highest level in the parse tree produce inefficient search strategies. Improvements to search optimization strategy remain high on our list of priorities.

When the RC21 observes a recurrence of a filter expression, it uses the parse tree and optimization strategy it produced the last time it saw that filter expression. Thus, it is more efficient to use a filter expression that has replaceable parameters rather than to produce a new filter expression by generating a string using sprintf or something like that.

VIEWS

Along with filters comes the capability of *views*. Views are simply filters that are permanently associated with a table. The table-filter combination is called a view.

Let's say we have a table called employees that was created like this:

```
db.AddTable ("Employees");
Table Employees(db, "Employees");
Employees.AddColumn ("FirstName", String);
Employees.AddColumn ("LastName", String);
Employees.AddColumn ("Salary", Integer);
```

Now, we could create a VIEW into this table by selecting only those employees with salary < 15000 as follows:

```
db.AddTable ("Group1", "Group 1 employees",
"Employees: Salary < 15000");
```

Observe that the *view expression* is formed by concatenating the *base table* name with a colon and the filter expression. Now, when we access the table called "Group1", we will automatically select only those employees with salary < 15000.

Composite Views

Anything you may do with a table, you may do with a view. Therefore, you can base a view on another view:

```
Employees.AddTable ("Group1 Females", "Group 1 employees",
"Group1: Sex == 'F'");
```

.. which would be the same as if you had written:

```
Employees.AddTable ("Employees", "Group1 Females", "Group 1 employees",
"Employees: Salary < 15000 && Sex == 'F'");
```

Note the the binding occurs at the time the table is constructed, *not* when it is created with AddTable. Thus, you may create a view that fails to open properly (with appropriate error handling) when you try to use it if you were to, say, delete the base table.

Updatable VIEWS

As mentioned previously, anything you may do with a table you may do with a view. So, you may do an AddRow, DeleteRow, PutString, PutInteger, etc.

VIEWS and Filters and Find

You may apply a filter to a view just as you may apply to the base table. The result is as though you had and'd the selection conditions. You may also do a Find. Then, it's as though you had concatenated all three selection expressions with &&.

Immediate Views

A VIEW expression may be supplied in place of a TABLE name expression in the Table function. Here is an example:

```
DBTAB tab = new Table (db, "Employees : %I - YearOfHire > 10",
(DBINTEGER) getyear());
```

Joins

In case it's not obvious, the relational join is performed explicitly by nesting Filter and ForAllRows or using a Find. For example, consider a table called "people" that has columns "name", and "zip", "zip" being the zipcode of the place where the person lives. There is also a table called "cities" that has columns "name" and "zip". Now, suppose we want to find all the people who live in "Burlington". In SQL, we would use something like this:

```
SELECT people.name FROM people, cities
WHERE people.zip = cities.zip AND cities.name = 'Burlington'
```

The code that follows builds a database and performs the above query using the ForAllRows macro and the Find function.

```
// EXAMP063 - Join
#include <iostream.h>
#include <rc21.h>
#include <string.h>

struct {
   char *cityname;
   long zipcode;
   } citylist[9] =
   {
   {"Burlington", 5401},
   {"Bennington", 5201},
   {"Barre", 5641},
   {"Middlebury", 5753},
   {"Stowe", 5672},
   {"Morrisville", 5661},
   {"Waterbury", 5676},
   {"Waterbury Center", 5677},
   {"Moscow", 5662},
   };

struct {
   char *personname;
   long zipcode;
   } person[20] =
   {
   {"John Elkins", 5672},{"Ethan Allan", 5401},{"Joe Smith", 5201},
   {"Zeke", 5641},{"Aaron", 5753},{"Geoffrey", 5672},
   {"Thomas Jefferson", 5661},{"Miles Davis", 5676},{"Mark Spitz", 5677},
   {"Bill Koch", 5662},{"Lola Falana", 5401},{"Ross Perot", 5201},
   {"George Bush", 5641},{"Bill Clinton", 5753},{"Rush Limbaugh", 5672},
   {"John McLaughlin", 5672},{"Madonna", 5401},{"Frank Zappa", 5201},
   {"Bill Frizzle", 5641},{"Anita Hill", 5753},
   };

void errorfunc(void)
   {
   PrintError("OH NO");
   }

void main(void)
   {
   int i;

   cout << "\nBegin examp063." << endl;
   SetErrorFunc(errorfunc);
```

```
Remove ("test.db");
Create("test.db");
Database db("test.db");
db.AddTable ("cities");
Table cities(db, "cities");
cout << cities.IsTable() << endl;

DBTAB t;
ForAllTableObjects (&db,t)
   {
   cout << t->ID();
   cout << ' ' << t->Name() << endl;
   }


cities.AddColumn ("name", String, "name of a city");
Column cityname(cities, "name");
cities.AddColumn ("zip", Integer, "zip code of this city");
Column cityzip (cities, "zip");
for (i=0; i<9; ++i)  /* build the city table */
   {
   cities.AddRow ();
   cityname << citylist[i].cityname;
   cityzip << citylist[i].zipcode;
   }

cout << "\nCities:" << endl;
ForAllRows (cities)
   {
   cout << cityname << ' ';
   cout << cityzip << endl;
   }

db.AddTable ("people");
Table people (db, "people");
people.AddColumn ("name", String, "name of a person");
Column personname (people, "name");
people.AddColumn ("zip", Integer, "zip code of reference", "",
   cityzip);
Column personzip (people, "zip");

for (i=0; i<20; ++i)  /* build the person table */
   {
   people.AddRow ();
   personname << person[i].personname;
   personzip << person[i].zipcode;
   }

cout << "\nPeople:" << endl;
ForAllRows (people)
   {
   cout << personname << ' ';
   cout << personzip << endl;
   }

/*
now, join the list of people with the list of cities;
the outer loop will select all people;  the inner loop will
select city based on zip code.
*/

cout << "\nWhere people live:" << endl;
ForAllRows (people)
   {
   DBINTEGER zip = personzip.GetInteger ();
```

```
      cities.Find ("zip == %I", zip);
      cout << personname << " lives in " << cityname << endl;
      }

   db.Commit ();
}
```

Fundamentally, the C code from the program above that is equivalent to the SQL code is

```
ForAllRows (people)
   {
   DBINTEGER zip = personzip.GetInteger ();
   cities.Find ("zip == %I", zip);
   cout << personname << " lives in " << cityname << endl;
   }
```

Order Of Retrieval Of Rows

When traversing a table, the order of retrieval of rows in the table may depend upon a number of factors. If there is an OrderBy, the order will depend strictly upon the values in the OrderBy column. When there is a filter and there is no OrderBy, the order will be determined by RC21 based upon some optimization strategy.

WARNING: When there is an OrderBy and the program changes a value in the OrderBy column, the current row of the table is at the new position. This technique could lead to unexpected results and is therefore not recommended However, if you must change the value in the OrderBy column, try something like the following:

```
DBROWID count; DBTAB table; DBCOL column;
int i;

...

count = Table->CountRows();
DBROWID rowid_array[count];

// first, get all the pre-change rowid's
column->OrderBy ();
ForAllRows (table)
   {
   rowid_array[i] = table->CurrentRow();
   }

/* now, go back and change the values */
for (i=0; i<count; ++i)
   {
   table->GotoRow (rowid_array[i]);
   /* now, change the value */
   column->Put...
   }
```

Class BookMark:public RCObject

The **BookMark** class is used to remember a row in a table and go to it. This replaces the function of Table.CurrentRow and Table.GotoRow (which may not function properly on uncommited rows in client-server operation).

Public Members

Construction/Destruction

BookMark	Constructs a new BookMark object and sets it to the current row in a given table.
~Database	Destroys the BookMark object.

Operations

Operator=	Assigns the bookmark to a row in a table.
Go	Goes to the table and row last assigned.

BookMark::BookMark

Syntax

BookMark();

BookMark(DBTAB *table*);

Parameters

table
the DBTAB pointer to table.

Remarks

Creates a new bookmark object, optionally set to a row in a table.

Return Value

None.

Example

```
DBTAB t = new Table(db, "table");
BookMark x;
BookMark y(t);
```

BookMark::Go

Syntax **DBERROR Go();**

Parameters None

Remarks Goes to a row in a table.

Return Value A reference to the bookmark.

Example

```
DBTAB t = new Table(db, "table");
BookMark x;
.
.
.
t.Find("name == %s", name);
x = t;
.
.
.
x.Go();  //  something like t.GotoRow(...
```

BookMark::operator =

Syntax **BookMark& operator=(Table& *table*);**

Parameters *table*
a reference to a table.

Remarks Assigns to the bookmark the indicated table and its current row.

Return Value A reference to the bookmark.

Example

```
DBTAB t = new Table(db, "table");
BookMark x;
.
.
.
t.Find("name == %s", name);
X = t;
```

class Database:public RCObject

The **Database** class encapsulates all database operations.

Public Members

Construction/Destruction

Database	Constructs a new database object given a pathname to a database file.
~Database	Destroys the database object and any subsidiary objects like Table and Column objects.

Conversion

Operator DB	Converts a reference to the object to a pointer to the object

Operations

Changed	Tells whether the database has been changed by any operation since the last Commit operation.
CachePages	Tells the number of cache pages associated with the Database object. Cache pages are essentially an LRU buffer for reading and writing data from/to the permanent storage device (the disk).
Name	Returns the name of the Database file.
Release	Tells the release number of the release of RC21 with which the associated database file was created.
PageSize	Tells the size of a database page. This is a property of the database file specificed when it is **Create**d.
HashSize	Tells the number of entries in the database's hash table. This value was specified or defaulted when the database was **Create**d.
CountTables	Tells how many tables there are in the database.
AddTable	Adds a new table to the database.
DeleteTable	Deletes an existing table from the database.
TableExists	Tells whether a table with a given name exists

	within the database.
Commit	Commits all changes to the database to the permanent, public database file. In multi-user operations, makes all changes visible to other users of the databae.
AutoCommit	In single-user operations specifies whether updates shall be automatically written to the database file. Also, tells whether the Database object is in AutoCommit mode.
RollBack	Rolls back uncommited changes.
FirstTable	Prepares to loop through a list of tables for the database. Used with **NextTable**.
NextTable	With **FirstTable**, iterates through all the tables in a database.
CurrentTable	With **FirstTable**, **NextTable** returns a pointer to the current table during iteration..
Copy	Makes a logical copy of a database in a new file.
LockSchema	In multi-user operations, attempts to get write control of the database schema.
CommitStatus	In single-user operations, tells how much resources will be needed successfully to comple a **Commit** operation.
operator =	Copies contents of one database into another database.
operator []	A cute way to reference a table.

Database::AddTable

Syntax

DBERROR AddTable(CDBSTRING *name***,**
CDBSTRING *description***=(CDBSTRING)"no description",**
CDBSTRING *view* **= (CDBSTRING)"");**

Parameters

name
the name of the table.

description
the description of the table. This is for documentation purposes only.

view
if this string is supplied, it is a *view-expression*. When the view-expression is supplied, this table is actually a stored view. A stored view is equivalent to a selection of rows from another *base* table.

The syntax for a *view-expression* is <tablename>: <filter-expression>

Note that the view expression is not parsed until the construction of a Table object for this table.

Remarks

Adds a new table or view to the database.

Return Value

Returns a DBERROR value.

Example

```
Create ("test.db");
Database x("test.db");
x.AddTable("Employees");
x.AddTable("Males", "Male Employees",
   "Employees: Sex == 'M'");
x.AddTable("Females", "Female Employees",
   "Employees: Sex == 'F'");
```

Database::AutoCommit

Syntax

int AutoCommit(int *onoff***);**

int AutoCommit(void);

Parameters *onoff*

the new value for the *AutoCommit* state. Use 1 to cause the database file to be updated as the Database object deems necessary. Use 0 to cause the database to be updated only upon **Commit** operations. The default value is 0.

Remarks In single-user operations it is possible to have changes to the database be commited to the disk as necessary, given the size of the cache. Naturally, this could result in the loss of database integrity should it turn out that the database is not properly closed. The AutoCommit operation with an argument reset the variable. Without an argument, the operation is used simply to find the current value of the variable.

Return Value The value of the *AutoCommit* state prior to this operation.

Example

```
Database x("test.db");
x.AutoCommit(1);  // no Commit operations necessary
.
.
.
cout << x.AutoCommit(0);  // outputs "1"
```

Database::CachePages

Syntax **int CachePages(void);**

Remarks Returns the number of cache pages allocated when the Database was constructed.

Return Value The number of cache pages associated with this Database object.

Example

```
Database x("test.db", 500);
cout << x.CachePages() << endl;  // outputs 500
```

Database::Changed

Syntax **int Changed(void);**

Remarks Returns TRUE if the database has changed since the last **Commit**, otherwise

False.

Return Value True if the database has changed since the last **Commit** operation, otherwise, FALSE.

Example

```
Database x("test.db");
x.AddTable("newtab", "new table");
cout << x.Changed() << endl;  // outputs 1
x.Commit();
cout << x.Changed() << endl;  // outputs 0
```

Database::CountTables

Syntax **DBROWID CountTables(void);**

Remarks Returns the number of tables in the database. This number includes the two "internal" tables, which are used to store the database schema. Note: DBROWID is a long.

Return Value Returns a DBROWID (long) value.

Example

```
Create ("x.db");
Database x("test.db");
cout << x.CountTables() << endl;  // outputs "2"
x.AddTable("newtab");
cout << x.CountTables() << endl;  // outputs "3"
```

Database::Commit

Syntax **DBERROR Commit(void);**

Remarks Commits all database changes to the database file. There are several steps to this process:

1. Copy all changes from buffers to a secondary file.
2. Flush secondary file to the disk.
3. Mark the database file with the name of the secondary file.
4. Copy all changes from the secondary file to the primary database file.
5. Flush the primary database file to the disk.

6. Unmark the database file with the name of the secondary file.

This way, if some unexpected interuption occurs, such as the machine losing power, it should be possible to restore the integrity of the database file. Such a restoration operation would occur automatically if the state of the database file necessitates it. The database would be restored to its state before or after the Commit, depending on how much of the operation had been completed.

Example

```
Database x("test.db");
.
.
.
x.Commit();
```

Database::CommitStatus

Syntax

DBERROR CommitStatus(unsigned long &*dbsize***, unsigned long &***tempsize***);**

Parameters

dbsize

variable into which will be stored the amount of additional memory that will be required successfully to complete a Commit operation.

tempsize

variable into which will be stored the amount of additional disk storage that will be required successfully to complete a Commit operation.

Remarks

In single-user operations use of this function will enable you to take action to ensure the successful completion of a Commit operation. You may compare the values returned with available memory and disk space. If a problem looms, you may be able to free resources so that the Commit will complete successfully.

Return Value

A DBERROR code.

Example

```
Database x("test.db");
.
.
unsigned long msize, tempsize;
x.CommitStatus(msize, dsize);
.
// make sure there's enough memory and disk space
.
x.Commit();
```

Database::Copy

Syntax **DBERROR Copy(DB *to*);**

Remarks Copies the contents of a database into another database.

Example

```
Database x("test.db");
Create ("newtest.db");
Database y("newtest.db");
x.Copy(y);  // contest of test.db copied to newtest.db
```

Database::CurrentTable

Syntax **DBTAB CurrentTable (void);**

Remarks With **FirstTable**, **NextTable**, selects the current table in an iteration through all the tables in a database.

Return Value Returns a DBTAB pointer to a Table object. This object was created with the **new** operator, so the Table must be deleted when it is no longer needed.

See Also **FirstTable**, **CurrentTable**, **ForAllTables** (macro)

Database::Database

Syntax **Database (CDBSTRING** *dbname***, CDBSTRING** *access***= "rw",**
int *npages***=20);**

Database (CDBSTRING *dbname***, int** *npages***=20);**

Parameters *dbname*
the name of the file containing the database or, in the case of the RC21 Server, the alias.

access
a string containing either "r" for read, "w" for write, or "rw" for read and write.

Npages
the number of pages to be allocated for the database cache. A larger number will generally result in better performance. In the client-server version, this parameter is ignored.

Remarks Constructs a new database object and associates it with the named database file. If access is not specified, "rw" is assumed.

Example

```
DB x = new Database("x.db");
Database y("x.db", "rw", 1000);
Database z("x.db", 1000);
```

Database::~Database

Syntax **~Database(void);**

Remarks Destroys the database object. Any Table objects associated with the database through the Table constructor are destroyed as well. Any Column objects associated are also destroyed.

Database::DeleteTable

Syntax **DBERROR DeleteTable(CDBSTRING *name*);**

Parameters *name*
the name of the table to be deleted.

Remarks Deletes the named table from the database. All objects associated with this table are deleted or marked invalid

Return Value DBERROR.

Example

```
Database x("test.db");
x.DeleteTable("newtab");
```

Database::FirstTable

Syntax **void FirstTable (void);**

Remarks With **NextTable**, **CurrentTable**, prepares to iterate through all the tables in a database.

See Also **NextTable**, **CurrentTable**, **ForAllTables** (macro)

ForAllTables

Syntax **ForAllTables (DBTAB** *db***) …**

Remarks This is a macro that combines the Database member functions **FirstTable**, **NextTable** to iterate through all the rows in a table.

See Also **FirstTable**, **CurrentTable**, **ForAllTables** (macro)

Example

```
Database x("test.db");
ForAllTables(x) // object x converted to &x automatically
   {
   DBTAB y = x.CurrentTable();
   cout << y.Name() << endl;
   delete y;
   }

output:
_tables
_columns
.
.
any other table names
.
.
```

Database::HashSize

Syntax **unsigned long HashSize(void);**

Remarks Returns the number of entries in the hash table of the database. This is the number specified when the database was **Create**d, or the default value if it was not specified.

Return Value Returns the number of entries in the hash table.

Example

```
Create ("test.db");
Database x("test.db");
cout << x.HashSize();  // outputs "1024"
Create ("test2.db", 2048, 1536);
Database y("test2.db");
cout << y.HashSize();  // outputs "1536"
```

Database::IsUsable

Syntax **IsUsable (void);**

Remarks Returns FALSE if the object was not properly constructed. Typically, would return FALSE if the database name were improperly specified.

Database::LockSchema

Syntax **LockReturn LockSchema (LockMode *m*);**

Parameters *m*

the enum value LockWait or LockNoWait. If LockWait is used, the operation will block until a lock can be obtained. If LockNoWait is used, the operation will not block and the return value will indicate whether the lock was obtained or not.

Remarks For multi-user operations, locks the schema, enabling changes to the schema. Lock persists until Commit.

Example

```
Database x("test.db");
x.LockSchema(LockWait);
x.AddTable("newtab", "new table");
x["newtab"].AddColumn("newcol", Integer);
x.Commit();
```

Database::Name

Syntax **CDBSTRING Name(void);**

Remarks Returns the filename of the database.

Return Value Pointer to a string containing the filename of the database.

Example

```
Database x("test.db");
cout << x.Name();  // outputs "test.db"
```

Database::NextTable

Syntax **int NextTable (void);**

Remarks With **FirstTable**, **CurrentTable**, moves to the next table in a database. This is used to iterate through all the tables in a database. Typically, the **ForAllTables** macro is used.

Return Value Returns TRUE if there are any more tables to iterate, otherwise, returns FALSE.

See Also **FirstTable**, **CurrentTable**, **ForAllTables** (macro)

Database::operator DB

Syntax **operator DB(void);**

Remarks Converts a Database object to a pointer to the object (returns this). This is provided so that the compiler will be able to perform automatic conversions between types.

Database::operator =

Syntax **Database& operator =(Database&** *fromdb***);**

Parameters *fromdb*
the Database object from which the contents will be copied.

Remarks Copies the contents of the right-hand Database operand to the left-hand Database operand.

Return Value A reference to the left-hand Database.

Example

```
Database x("test1.db");
Create ("test2.db");
Database y("test2.db");
y = x;  // copies contents of x into y
```

Database::operator []

Syntax **class Table& operator[] (CDBSTRING** *tablename***);**

Parameters *tablename*
the name of the table you wish to reference.

Remarks Basically, this is syntactic fluff - a cool way to reference a table without constructing a Table object. Fundamentally, a reference is returned to a preconstructed object. All references to the preconstructed object have the same effect on the database as if they were references to an object constructed by you.

Example

```
Database x("test.db");
x.AddTable("newtab", "new table");
cout << x["newtab"].Name() << endl;  // outputs "newtab"
```

Database::PageSize

Syntax **unsigned PageSize(void);**

Remarks Returns the size in bytes of one logical page in the database. This is the number specified when the database was **Create**d, or the default value if it was not specified.

Return Value Returns the logical pagesize for the database.

Example

```
Create ("test.db");
Database x("test.db");
cout << x.PageSize();  // outputs "1024"
Create ("test2.db", 2048);
Database y("test2.db");
cout << y.PageSize();  // outputs "2048"
```

Database::Release

Syntax **CDBSTRING Release(void);**

Remarks Returns a string that contains the release number of the release of RC21 with which the database was originally **Create**d.

Example

```
Create ("test.db");
Database x("test.db");
cout << x.Release() << endl;  // outputs "7.1.B1"
```

Database::RollBack

Syntax **DBERROR RollBack (void);**

Remarks Rolls back all the changes to the database since the last Commit. In other words, all changes are discarded.

Return Value A DBERROR code.

Example

```
Database x("test.db");
x.AddTable("newtab", "new table");
cout << x.Changed() << endl;  // outputs 1
x.RollBack();
cout << x.Changed() << endl;  // outputs 0
```

Database::TableExists

Syntax **int TableExists(CDBSTRING** *name***);**

Parameters *Name*
the name of the table.

Remarks Looks in the list of tables for this database to determine if a table exists.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Example

```
Create ("test.db");
Database x("test.db");
x.AddTable("newtab");
cout << x.TableExists("newtab") << endl;  // outputs 1
cout << x.TableExists("anothertab") << endl;  // outputs 0
```

class Table:public RCObject

The **Table** class encapsulates all operations on a table.

Public Members

Construction/Destruction

Table	Constructs a new Table object given a Database object and the name of a table.
~Table	Destroys the Table object and any subsidiary objects like Column objects.

Conversion

Operator DB	Converts a reference to the Table object to a pointer to its Database object.
Operator DBTAB	Converts a reference to the Table object to a pointer to itself.

Classifiers

IsTable	Returns TRUE for a Table object. Returns FALSE for any other kind of object. This is useful for determining the type of a generic RCObject. Truth be told, RTTI could be used, but this is easier.

Properties

Name	Returns the name of a table or changes the name of a table.
Description	Returns the description of the table or changes the description.
View	Returns the view expression for a table or changes the view expression.
ID	Returns the unique ID for this table in the database.
CountColumns	Returns the number of columns in the table.
CountRows	Returns the number of rows in the table.

Operations

AddColumn	Adds a new column to the table.
AddKey	Adds a new key to the table. A key is a

	combination of the contents of one or more columns with potentially ascending and descending segments
DeleteColumn	Deletes a column or a key.
Copy	Copies the table to another table.
AddRow	Add a new, empty row to the table.
DeleteRow	Delete the current row.
CopyRow	Copy the contents of a row to another (existing) row.
DuplicateRow	Add a row and copy in the contents of the current row.
IsaRow	Returns TRUE if the Table cursor is positioned at a valid row.
FinishRow	Checks constraints and returns errors if the current row is not "complete and correct".
ResetRow	Positions the Table cursor prior to iterating through the rows of a table.
NextRow	With **ResetRow**, iterates to the next row in the table.
PrevRow	With **ResetRow**, iterates to the previous row in the table.
CurrentRow	Returns the ROWID of the current row. (Not recommended in multiuser operations.)
GotoRow	Moves the Table cursor to the row with the given ROWID. (Not recommended in multi-user operations.)
GetRow	Fetches all the contents of a row into a string (for PERL).
Erase	Erase all the rows in a table.
OrderBy	Sets the order in which rows will be retrieved from the Table.
Unordered	Sets the order in which rows will be retrieved to be the chronological order or any other order deemed convenient by the retrieval function.
Filter	Places constraints on which rows will be retrieved from the Table. These *filter*s are cumulative.
vFilter	Same as **Filter**, but with a va_list pointer.
ClearFilters	Removes all filters from the table.
Find	Moves the Table cursor to a row that matches

	the given filter expression and returns TRUE or returns FALSE if no row matches the filter expression.
vFind	Same as **Find**, but with a va_list argument.
RowExists	Determines if *any* row in the table matches the given filter expression.
vRowExists	Same as **RowExists** but with a va_list argument.
PassesFilter	Determines if the current row passes the given filter expression.
vPassesFilter	Same as **PassesFilter** but with a va_list argument.
FirstColumn	Prepares to iterate through the columns in a table.
NextColumn	With **FirstColumn**, iterates to the next column in the table.
CurrentColumn	With **FirstColumn**, **NextColumn**, returns a pointer to the current column in an iteration of all columns in a table.
Operator ++	Does an **AddRow**.
Operator []	A cute way to reference a column.
Serno	Delivers a unique serial number for this this table.

Table::AddColumn

Syntax

DBERROR AddColumn(CDBSTRING *name***,**
COLATTRIBS *attr***=NOATTRIBS,**
CDBSTRING *description***=(CDBSTRING)"no description",**
CDBSTRING *format***=(CDBSTRING)"",**
const DBCOL *reference***=NULL);**

Parameters

name
the name of the new column.

attr
a **COLATTRIBS** word that contains the attributes for the new column. If this is omitted, the new column will be untyped, unindexed, non-shared.

description
a descriptive string for the new column. This is for documentation purpose only.

format
a C format code like "%s", "%ld", that can be used to help format output for a generalized reporting program.

reference
if this value is supplied, RC21 will allow only values that occur in the reference column to be placed in this column. This ensures that only certain acceptable values will be stored.

Remarks

Adds a new column to the table.

Return Value

Returns a DBERROR code.

Example

```
Create ("company.db");
Database company("company.db");
company.AddTable("state");
Table state(company, "state");
company.AddTable("employee");
Table employee(x, "employee");
employee.AddColumn("FirstName", String, NULL, "%s%");
employee.AddColumn("LastName", String, NULL, "%s%");
employee.AddColumn("StreetAddress", String, NULL, "%s%");
employee.AddColumn("City", String, NULL, "%s%");
employee.AddColumn("State", String, "", "%s", State);
```

Table::AddKey

Syntax **DBERROR AddKey (CDBSTRING** *name***, CDBSTRING** *keys***, COLATTRIBS** *attr***=NOATTRIBS);**

Parameters

Name
the name of the key.

Keys
a string containing the key description. The key description is a list of column names separated by "+" or "-" to indicate ascending or descending key segments. The names of string or blob columns may optionally be followed by [*begin*:*length*] to designate *substring keys*.

Attr
the attribute Unique may be specified. NonUnique is assumed.

Remarks Adds a new key to the table. A key is a combination of the contents of one or more columns with potentially ascending and descending segments.

Return Value A DBERROR code

Example

```
x.AddKey ("LastFirst", "+LastName+FirstName");
```

Table::AddRow

Syntax **DBERROR AddRow (void);**

Remarks Adds a new, empty row to the table and positions the table cursor at that row.

Return Value A DBERROR code

Table::ClearFilters

Syntax **Void ClearFilters (void);**

Remarks Removes all filters from the table. In the case of a *view* Table, does not remove the base view expression.

Table::Copy

Syntax **DBERROR Copy (CDBSTRING** *newtab***);**

Parameters *Newtab*
the name of a new table that will be added to the database and into which the contents of this table will be copied.

Remarks Creates a new table and copies the contents there.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Table::CopyRow

Syntax **DBERROR CopyRow (DBROWID** *row***);**

DBERROR CopyRow (DBTAB *table***);**

Parameters *row*
a rowid in the same table.

table
a pointer to another table that shares column names with this table.

Remarks Copy the contents of a row to another (existing) row. In the case where the argument is a rowid, the contents of a row will be copied to another row in the same table. In the case where the argument is a pointer to Table, the contents of this row will be copied to the current row in the *to* table. Values will be copied from each column to its namesake in the *to* table.

Return Value A DBERROR code.

Table::CountColumns

Syntax **DBROWID CountColumns (void);**

Remarks Counts the columns in the table.

Return Value Returns the number of columns in the table, include the internal columns used to store the schema values and other internal values.

Example

```
Create ("test.db");
Database x("test.db");
x.AddTable("newtab");
cout << x.CountColumns() << endl;  // outputs "1"
```

Table::CountRows

Syntax **DBROWID CountRows (void);**

Remarks Counts the (nondeleted) rows in the table.

Return Value Returns the number of rows in the table.

Table::CurrentColumn

Syntax **DBCOL CurrentColumn (void);**

Remarks With **FirstColumn**, **NextColumn**, returns a pointer to the current column in an iteration of all columns in a table.

Return Value Returns the DBCOL pointer to the next column in the iteration

See Also **ForAllColumns** macro.

Table::CurrentRow

Syntax **int TableExists(CDBSTRING** *name***);**

Parameters *Name*
the name of the table.

Remarks Determines the ROWID of the current row.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Example

```
Create ("test.db");
Database x("test.db");
x.AddTable("newtab");
cout << x.TableExists("newtab") << endl;  // outputs 1
cout << x.TableExists("anothertab") << endl;  // outputs 0
```

Table::DeleteColumn

Syntax **DBERROR DeleteColumn(CDBSTRING** *name***);**

Parameters *Name*
the name of the column to delete.

Remarks Deletes a column or a key from the table.

Return Value A DBERROR code.

Table::DeleteRow

Syntax **DBERROR DeleteRow(void);**

Remarks Deletes the current row.

Return Value Returns a DBERROR code..

Table::Description

Syntax **CDBSTRING Description (void);**

DBERROR Description (CDBSTRING *description***);**

Remarks Returns the description of the table or changes the description.

Return Value For the zero-argument form, returns a string containing the current value for the column description. For the one-argument form, returns a DBERROR code.

Table::DuplicateRow

Syntax **DBERROR DuplicateRow (void);**

Remarks Add a row and copy in the contents of the current row.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Table::Erase

Syntax **DBERROR Erase (void);**

Remarks Erase all the rows in a table.

Return Value Returns a DBERROR code.

Table::Filter

Syntax **DBERROR Filter (CDBSTRING** *filter* **...);**

Parameters *filter*
a string containing a filter expression.

…
additional arguments if the filter string contains replaceable parameters (%I, %R, %S, etc.)

Remarks Places constraints on which rows will be retrieved from the Table. These *filter*s are cumulative.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Example

```
Database x("company.db");
Table y(x, "Employees");
y.Filter("Salary > 50000 && JobCode == 56");
```

Table::Find

Syntax **int Find (CDBSTRING** *filter* **...);**

Parameters *filter*
a string containing a filter expression.

…
additional arguments if the filter string contains replaceable parameters

(%I, %R, %S, etc.)

Remarks Moves the Table cursor to a row that matches the given filter expression and returns TRUE, or returns FALSE if no row matches the filter expression.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Example

```
Database x("company.db");
Table y(x, "Employees");
y.Find("LastName == '%S' && FirstName == '%S'", lastname,
firstname);
```

Table::FinishRow

Syntax **DBERROR FinishRow (void);**

Remarks Checks constraints and returns errors if the current row is not "complete and correct".

Return Value Returns a DBERROR code. It is important to check this code to make sure the value is DB_OK. Otherwise, there would be no point in performing the operation.

Table::FirstColumn

Syntax **void FirstColumn (void);**

Parameters *name*
the name of the table.

Remarks With **NextColumn, Current Column**, prepares to iterate through the columns in a table.

See Also **ForAllColumns** macro.

Table::ResetRow

Syntax **DBERROR ResetRow(void);**

Remarks Positions the Table cursor prior to iterating through the rows of a table.

Return Value Returns a DBERROR code.

ForAllColumns (macro)

Syntax **ForAllColumns(DBTAB** *tab***) …**

Parameters *tab*
pointer to a Table object whose columns we will iterate.

Remarks This macro composes **FirstColumn**, **NextColumn**, to move through all the columns in a table. Use **CurrentColumn** within the loop to access each column. Note that **CurrentColumn** returns a pointer (DBCOL) to an existing column. Do not **delete** it.

Example

```
Create ("test.db");
Database x("test.db");
ForAllColumns(x["_tables"])
   {
   DBCOL y = x.CurrentColumn();
   cout << y->Name() << endl;

output (these are the "internal" columns):

_status
_name
_description
_view
```

Table::GetRow

Syntax **CDBSTRING GetRow (CDBSTRING** *column_list***, char** *sep***);**

Parameters *column_list*
a list of column names separated by spaces.

sep
the character that will be used to separate the values placed in the output string.

Remarks Fetches all the contents of a row into a string (for PERL).

Return Value Returns a string containing the string representation of the values in the columns.

Table::GotoRow

Syntax **void GotoRow(DBROWID** *newrow***);**

Parameters *newrow*
the rowid of the row to move to.

Remarks Moves the Table cursor to the row with the given ROWID.

Table::ID

Syntax **FIELDID ID(void);**

Remarks Returns the unique ID for this table in the database.

Return Value The value returned is a unique indentifier for the table in the database. This function is provided primarily for internal purposes.

Table::IsaRow

Syntax **int IsaRow(void);**

Remarks Returns TRUE if the Table cursor is positioned at a valid row. In an object-oriented program it is possible for an even to cause a row to be deleted. Meanwhile the Table is still positioned at the row - it has not moved to another row yet. This function can be used to make sure that the row still exists - that it has not been deleted by some other function

Return Value Returns TRUE if the Table cursor is positioned at a valid row.

Table::IsTable

Syntax **int IsTable(void);**

Remarks Returns TRUE for a Table object. Returns FALSE for any other kind of object. This is useful for determining the type of a generic RCObject. Truth

be told, RTTI could be used, but this is easier.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Table::Name

Syntax **CDBSTRING Name (void);**

DBERROR Name (CDBSTRING name);

Parameters *name*
the new name for the table.

Remarks Returns the name of a table or changes the name of a table.

Return Value In the case of the 0-argument form, returns the name of the table. In the case of the 1-argument form, returns a DBERROR code.

Example

```
Create ("test.db");
Database x("test.db");
x.AddTable("newtab");
Table y(x, "newtab");
cout << y.Name() << endl;  // outputs "newtab"
y.Name("renamed");
cout << y.Name() << endl;  // outputs "renamed"
```

Table::NextColumn

Syntax **int NextColumn (void);**

Remarks With **FirstColumn**, iterates to the next column in the table.

Return Value Returns TRUE if there is another column. Returns FALSE at the end of the list.

See Also **FirstColumn**, **CurrentColumn**, **ForAllColumns**

Table::NextRow

Syntax **int NextRow (void);**

Remarks With **ResetRow**, iterates to the next row in the table.

Return Value Returns TRUE if there are more rows. Returns FALSE at the end of the table.

See Also **ResetRow**, **ForAllRows**

Table::operator DB

Syntax **operator DB(void);**

Remarks Converts a reference to the Table object to a pointer to its Database object.

Return Value Returns a DB (pointer to a Database object).

Example

```
Create ("test.db");
Database x("test.db");
x.AddTable("newtab");
Table y(x, "newtab");
DB z = DB(y);
cout << z->Name() << endl;  // outputs "test.db"
```

Table::operator DBTAB

Syntax **operator DBTAB (void);**

Remarks Converts a reference to a Table object to a pointer to the object.

Return Value Returns a DBTAB (pointer to a.Table object). This operator is provided primarily to allow Table references to be used as though they were DBTAB pointers in function calls requiring DBTAB arguments.

Example

```
Create ("test.db");
Database x("test.db");
x.AddTable("newtab");
Table y(x, "newtab");  // x converted automatically to DB
y.AddColumn("newcol");
Column z(y, "newcol");  /* y converted automatically to
   DBTAB */
```

Table::operator ++

Syntax **void operator ++(void);**

Remarks Does an **AddRow**.

Table::Operator[]

Syntax **Column& operator [] (CDBSTRING** *colname***);**

Parameters *colname*
the name of the column to access.

Remarks A cute way to reference a column.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Example

```
Table employees(db, "Employees");
cout << employees["LastName"].Name() << endl;

output: LastName
```

Table::OrderBy

Syntax **DBERROR OrderBy (CDBSTRING** *colname***);**

Parameters *colname*
the name of the column that will be used for ordering rows.

Remarks Sets the order in which rows will be retrieved from the Table.

Return Value Returns a DBERROR code.

Example

```
Table employees(db, "employees");
employees.OrderBy("LastFirst");
ForAllRows(employees)
   {
   cout << employees["LastName"].GetString();
   cout << ", " << employees["FirstName"].GetString() <<
      endl;
   }

Results:  list of employees ordered by last name, first name.
```

Table::PassesFilter

Syntax **int PassesFilter (CDBSTRING *filter* ...);**

Parameters *Filter*
a string containing a filter expression.

…
additional arguments if the filter string contains replaceable parameters (%I, %R, %S, etc.)

Remarks Determines if the current row passes the given filter expression.

Return Value Returns TRUE if the current row in the Table passes the filter expression.

Table::PrevRow

Syntax **int PrevRow (){return NextRow(-1);**

Remarks With **ResetRow**, iterates to the previous row in the table. This is used to move through the rows in a table backwards. You may, instead use the macro **ForAllRowsBackwards**.

Return Value Returns TRUE if there is another row to access, otherwise returns FALSE.

Table::RowExists

Syntax **int RowExists (CDBSTRING filter ...);**

Parameters *Filter*
a string containing a filter expression.

…
additional arguments if the filter string contains replaceable parameters (%I, %R, %S, etc.)

Remarks Determines if *any* row in the table matches the given filter expression. It does *not* move to a matching row. It simply returns TRUE or FALSE.

Return Value Returns TRUE if a row exists in the Table that matches the given filter expression. Returns FALSE otherwise.

Table::SerNo

Syntax **DBINTEGER SerNo();**

Remarks Each table has a serial-number-dispenser. Each time the serial number is accessed, it is incremented by one. To get the current serial number for the table, use this function. Serial numbers range in value from 1 to MAX_UNSIGNED_INTEGER.

NOTE: In the server version, the serial number isnot reused – even if the transaction containing this function call is rolled back.

Return Value Returns the current serial number.

Table::Table

Syntax **Table (DB** *db***, CDBSTRING** *tname***);**
Table (Table& *table***);**

Parameters *Db*

a DB pointer to a Database object with which this new Table object will be associated. If this argument is not supplied, the current Database is assumed. The current Database is the last Database object that has been accesses either directly or indirectly by database operations.

Tname

the name of the table.

Remarks Constructs a new Table object given a Database object and the name of a table.

Or .. construct a new Table given an existing Table object. The new Table will be an exact copy of the existing Table, including current row, filters, etc.

Table::~Table

Syntax **~Table (void);**

Remarks Destroys the Table object and any subsidiary objects like Column objects.

Table::Unordered

Syntax **DBERROR Unordered(void);**

Remarks Sets the order in which rows will be retrieved to be the chronological order or any other order deemed convenient by the retrieval function. Cancels any OrderBy specification.

Return Value Returns a DBERROR code.

Table::vFilter

Syntax **DBERROR vFilter (CDBSTRING** *filter***, va_list** *ap***);**

Parameters *Filter*
a filter expression.

Ap
a va_list pointer to an argument list.

Remarks Same as **Filter**, but with a va_list pointer. This is a way for a function to pass on its argument list to the RC21 filter mechanism.

Return Value Returns a DBERROR code.

Table::vFind

Syntax **int vFind (CDBSTRING** *filter***, va_list** *ap***);**

Parameters *Filter*
a filter expression.

Ap
a va_list pointer to an argument list.

Remarks Same as **Find**, but with a va_list argument. This is a way for a function to pass on its argument list to the RC21 filter mechanism.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Table::vPassesFilter

Syntax **int vPassesFilter (CDBSTRING** *filter***, va_list** *ap***);**

Parameters *Filter*
a filter expression.

Ap
a va_list pointer to an argument list.

Remarks Same as **PassesFilter** but with a va_list argument. This is a way for a function to pass on its argument list to the RC21 filter mechanism.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Table::vRowExists

Syntax **int vRowExists (CDBSTRING** *filter***, va_list** *ap***);**

Parameters *Filter*
a filter expression.

Ap
a va_list pointer to an argument list.

Remarks Same as **RowExists** but with a va_list argument. This is a way for a function to pass on its argument list to the RC21 filter mechanism.

Return Value Returns TRUE if a row exists in the table that satisfies the filter expression. Otherwise, returns FALSE. Does not modify the Table cursor.

Table::View

Syntax **CDBSTRING View (void);**

DBERROR View (CDBSTRING *newview***);**

Parameters *Newview*
the new view expression.

Remarks Returns the view expression for a table or changes the view expression to the new value in *newview*.

Return Value Returns either a string or a DBERROR code, depending on the form used.

class Column:public RCObject

The **Column** class encapsulates all operations on a column.

Public Members

Construction/Destruction

Column	Constructs a new Column object given a Table object and the name of a colum
~Column	Destroys the Column object.

Conversion

operator DBTAB	Converts a reference to the Column object to a pointer to its Table object.
operator DBCOL	Converts a reference to the Column object to a pointer to itself.

Classifiers

IsColumn	Returns TRUE for a Column object. Returns FALSE for any other kind of object. This is useful for determining the type of a generic RCObject.

Properties

Name	Returns the name of a table or changes the name of a table.
Description	Returns the description of the table or changes the description.
Format	Returns the C format of the column or changes it.
Attributes	Returns attributes of the column (data type, etc) or changes them.
Index	Sets the column to the “indexed” state and reindexes it, if necessary.
Unindex	Sets the column to the “unindexed” state and discards any indexing structures, if necessary.
Reference	Returns a pointer to the associated *reference* object, if any, or NULL.

Tab	Returns a pointer to the associated Table object for this Column.
GetSQLAttributes	Does what it says.
PutSQLAttributes	Does what it says.
ID	Returns the unique ID for this column in the database.

Operations

Erase	Erase all the values in the column.
HasAnyValues	Tells you if there are any non-null values in the column.
KeyExists	Tells if a given value occurs in the column.
Copy	Copies contents of a column to a new column with a different name.

Storing Values

PutInteger	Store an integer value in the column at the current row (the cell).
PutReal	Store a real (double or float) value in the cell.
PutString	Store a string value in the cell.
PutSTRING	Store a STRING (case-insensitive) value in the cell.
PutBLOB	Store a BLOB (counted byte array) value in the cell.
PutNull	Store a Null value in the cell (in other words, erase the value).
PutNULL	Same as PutNull if you can't remember how it's spelled.
PutUser	Store a User value (user-defined sorting algorithm) in the cell.
Put	Polymorphic function to store a value or arbitrary type in the cell.
operator <<	Polymorphic function to store a value or arbitrary type in the cell.

AddRow	Adds a new row to this column's table.
Inc	Add a value to the current value in the column.

Fetching Values

GetInteger	Fetches an integer value from the cell.
GetReal	Fetches a double value from the cell.
GetString	Fetches a string value from the cell.
GetValue	Fetches a DBVALUE from the cell. Use this for fetching BLOBs.
GetType	Find out the type of the value in the cell.
GetSize	Find out how many bytes would be required to store the value in the cell.
Size	Same as **GetSize**.
Get	Same as **GetValue**.
operator int	Same as **GetInteger**.
operator double	Same as **GetReal**.
operator const char*	Same as **GetString**.
operator float	Same as **GetReal**.

Column::AddRow

Syntax **DBERROR AddRow(void);**

Remarks Adds a new row to this column's table.

Return Value Returns a DBERROR code.

Column::Attributes

Syntax **COLATTRIBS Attributes (void);**

DBERROR Attributes(COLATTRIBS *newattribs***);**

Parameters *Newattribs*
a COLATTRIBS value representing the new attributes for the column.

Remarks Returns attributes of the column (data type, etc) or changes them to the values in *newattribs*. The only attribute that may be changed is the Indexed/Unindexed attribute. This may also be accomplished by using the **Index** or **Unindex** operations. However, this operation is provided for symmetry.

Return Value Returns a DBERROR code if changing attributes. Returns a COLATTRIBS value if querying attributes.

Column::Column

Syntax **Column (DBTAB** *table***, CDBSTRING** *cname***);**

Parameters *Table*
a pointer to a Table with which this column will be associated. The database table associated with the Table must contain a column with the given name.

Cname
the name of the column with which to associate this Column object.

Remarks Constructs a new Column object given a Table object and the name of a column.

Column::~Column

Syntax **~Column(void);**

Remarks Destroys the Column object.

Column::Copy

Syntax **DBERROR Copy (CDBSTRING** *newcol***);**

DBERROR Copy (DBCOL *destination***);**

Parameters *Newcol*
the name of the new column to create and copy this Column's contents to.

Destination
a pointer to a Column object for an existing column in the same table. The contents of *this* column will be copied to the destination column.

Remarks Copies contents of a column to a new column with a different name. In other words, this operation duplicates a column and gives the duplicated column the new name.

The alternate form copies the contents of one column in the Table to another column in the same Table.

Return Value Returns a DBERROR code.

Column::IsColumn

Syntax **int IsColumn(void);**

Remarks Returns TRUE for a Column object. Returns FALSE for any other kind of object. This is useful for determining the type of a generic RCObject.

Return Value Returns TRUE for a Column object.

Column::Description

Syntax **CDBSTRING Description (void);**

DBERROR Description (CDBSTRING *newdescription***);**

Parameters *Newdescription*
the new description to assign to the column.

Remarks Returns the description of the column or changes the description.

Return Value Returns the existing value for the column description, or a DBERROR code, depending on which form is used.

Column::Erase

Syntax **Void Erase (void);**

Remarks Erase all the values in the column.

Column::Format

Syntax **CDBSTRING Format (void);**

DBERROR Format (CDBSTRING *newformat***);**

Parameters *Newformat*
the new format value for the column.

Remarks Returns the C format of the column or changes it.

Return Value Returns the current value for *format* or a DBERROR code.

Column::Get

Syntax **DBVALUE Get (void)**

Remarks Same as **GetValue**.

Return Value A DBVALUE object containing type, size, value, etc.

Column::GetInteger

Syntax **DBINTEGER GetInteger(void);**

Remarks Fetches an integer value from the cell.

Return Value Returns a DBINTEGER value.

Column::GetReal

Syntax **DBREAL GetReal(void);**

Remarks Fetches a double value from the cell.

Return Value Returns a DBREAL value.

Column::GetSize

Syntax **DBSIZE GetSize ();**

Remarks Find out how many bytes would be required to store the value in the cell.

Return Value A count of bytes.

Column::GetSQLAttributes

Syntax **int GetSQLAttributes (DBINTEGER *type,**

DBINTEGER *precision,
DBINTEGER *scale);

Parameters

Type
SQL type.

Precision
SQL precision.

Scale
SQL scale.

Remarks Fetches ODBC attributes

Return Value Returns TRUE if SQL attributes are present, otherwise FALSE.

Column::GetString

Syntax **CDBSTRING GetString(void);**

Remarks Fetches a string value from the cell.

Return Value Returns a (pointer to a) string. Since the value returned is a pointer, the data pointed to is in a static area. This means that the pointer is only valid until the next call to RC21. Make sure to copy the value if persistence is necessary.

Column::GetType

Syntax **COLATTRIBS GetType (void);**

Remarks Find out the type of the value in the cell.

Return Value Returns COLATTRIBS value containing the type of data stored in the cell. All non-type data in this value have been masked out, so it is possible to compare the value with the manifest constants Real, Integer, String (DBString), Nbytes, Blob, User, Null, Key, STRING, Eblob.

Column::GetValue

Syntax **DBVALUE& GetValue(void);**

Remarks Fetches a DBVALUE from the cell. Use this for fetching BLOBs.

Return Value Returns a reference to a DBVALUE value. Since this a reference to data in the RC21 data area, this reference is valid only until the next call to RC21.

Column::HasAnyValues

Syntax **int HasAnyValues(void);**

Remarks Tells you if there are any non-null values in the column.

Return Value Returns TRUE if there are values in the column, otherwise FALSE.

Column::ID

Syntax **FIELDID ID(void);**

Remarks Returns the unique ID for this column in the database. This is provided primarily for internal purposes.

Return Value The column ID.

Column::Inc

Syntax **DBINTEGER Inc(DBINTEGER _i);**
DBREAL Inc(DBREAL _i);

Remarks Adds *_i* to the column value and returns the old value.

Return Value The old value of the column.

Column::Index

Syntax **DBERROR Index(void);**

Remarks Sets the column to the "indexed" state and reindexes it, if necessary.

Return Value Returns a DBERROR code.

Column::Name

Syntax **CDBSTRING Name (void);**

DBERROR Name (CDBSTRING *name***);**

Parameters *Name*
the new name for the column.

Remarks Returns the name of a table or changes the name of a table.

Return Value Returns a string containing the name of the column in the first case or a DBERROR code in the second case.

Column::operator const char*

Syntax **Operator const char*(void);**

Remarks Same as **GetString**.

Return Value Returns the string value of the cell.

Column::operator DBCOL

Syntax **Operator DBCOL(void);**

Remarks Converts a reference to the Column object to a pointer to itself.

Return Value Returns *this*. Enables the compiler automatically to convert references to a Column to DBCOL, where required for function calls.

Column::operator DBTAB

Syntax **Operator DBTAB(void);**

Remarks Converts a reference to the Column object to a pointer to its Table object.

Return Value Returns a DBTAB pointer.

Column::operator double

Syntax **Operator double(void);**

Remarks Same as **GetReal**.

Return Value Returns the double value in the cell. If the data type stored in the cell cannot be converted to double, an error is set and (double)0 is returned.

Column::operator float

Syntax **Operator float(void);**

Remarks Same as (float)**GetReal()**. If the data type stored in the cell cannot be converted to float, an error is set and (float)0 is returned.

Return Value Returns a float value from the cell.

Column::operator int

Syntax **Operator int(void);**

Remarks Same as **GetInteger**.

Return Value Returns an integer value from the cell. If the data type stored in the cell cannot be converted to int, an error is set and (int)0 is returned.

Column::operator <<

Syntax

DBINTEGER operator << (DBINTEGER *x***);**

int operator << (int *x***);**

DBREAL operator << (DBREAL *x***);**

CDBSTRING operator << (CDBSTRING *x***);**

Parameters

X
an integer, real, or string value.

Remarks

Polymorphic function to store a value or arbitrary type in the cell. Syntactic candy, these operators simply invoke **PutInteger**, **PutReal**, or **PutString**.

Return Value

Returns the value stored.

Column::Put

Syntax

DBERROR Put(DBINTEGER *x***);**
DBERROR Put(DBREAL *x***);**
DBERROR Put(CDBSTRING *x***);**
DBERROR Put(CDBBLOB *x***, DBSIZE** *n***);**
DBERROR Put(const DBUSER *x***, DBSIZE** *n***);**
DBERROR Put(const DBVALUE& *x***);**

Parameters

X
the value to store.

Remarks

Polymorphic function to store a value of arbitrary type in the cell.

Return Value

Returns a DBERROR code.

Column::PutBLOB

Syntax

DBERROR PutBLOB(CDBBLOB *blobptr***,DBSIZE** *blobsize***)**

Parameters

Blobptr
pointer to the BLOB to store.

Blobptr
size of the BLOB to store.

Remarks Store a BLOB (counted byte array) value in the cell.

Return Value Returns a DBERROR code

Column::PutInteger

Syntax **DBERROR PutInteger(DBINTEGER** *intvalue***);**

Parameters *intvalue*
the integer value to store.

Remarks Store an integer value in the column at the current row (the cell).

Return Value Returns a DBERROR code.

Column::PutNull
Column::PutNULL

Syntax **DBERROR PutNull(void);**

DBERROR PutNULL(void);

Remarks These functions are equivalent. No matter which way we spelled it, we couldn't remember how it was spelled. They erase the value in the cell, returning it to the Null value.

Return Value Returns a DBERROR code.

Column::PutReal

Syntax **DBERROR PutReal(DBREAL** *realvalue***);**

Parameters *realvalue*
the value to store.

Remarks Store a real (double or float) value in the cell.

Return Value Returns a DBERROR code.

Column::PutSQLAttributes

Syntax **DBERROR PutSQLAttributes (DBINTEGER** *type***,**
DBINTEGER *precision***,**
DBINTEGER *scale***);**

Parameters *type*
ODBC type.
precision
ODBC precision.

scale
ODBC scale.

Remarks Does what it says.

Return Value Returns a DBERROR code.

Column::PutSTRING

Syntax **DBERROR PutSTRING(CDBSTRING** *case_insensitive_string***);**

Parameters *case_insensitive_string*
the string value to store.

Remarks Store a STRING (case-insensitive) value in the cell. If the column is Indexed, the value will be indexed without respect to upper/lower case.

Return Value Returns TRUE if the named table is a table in this database, otherwise false.

Column::PutString

Syntax **DBERROR PutString(CDBSTRING** *string***);**

Parameters *string*
the string value to store.

Remarks Store a string value in the cell.

Return Value Returns a DBERROR code.

Column::PutUser

Syntax **DBERROR PutUser(const DBUSER** *blob***,DBSIZE** *blobsize***);**

Parameters *blob*
the blob value to store.
blobsize
the size of the blob to store.

Remarks Store a User value (user-defined sorting algorithm) in the cell.

Return Value Returns a DBERROR code.

Column::Reference

Syntax **DBCOL Reference (void);**

Remarks Returns a pointer to the associated *reference* object, if any, or NULL.

Return Value Returns a DBCOL pointer to the object associated with the reference column for this column.

Column::Size

Syntax **DBSIZE Size (void);**

Remarks Same as **GetSize**.

Return Value Returns the number of bytes required to store the value in the cell.

Column::Tab

Syntax **DBTAB Tab (void);**

Remarks Returns a pointer to the associated Table object for this Column.

Return Value Returns a DBTAB pointer to a Table object.

Column::Unindex

Syntax **DBERROR Unindex(void);**

Remarks Sets the column to the "unindexed" state and discards any indexing structures, if necessary.

Return Value Returns a DBERROR code.

Appendix

Compiling and Linking RC21 Applications

The header file needed to build RC21 applications is rc21.h. If you are on a Unix system make sure your #include gets the case of the file correct.

The header file in in the \rc21\include directory (or /rc21/include).

Rc21.h will include the file rc21_config.h. You must construct an rc21_config.h that corresponds to your environment and contains the following definitions:

P_os
P_compiler
P_linkage
P_service

The acceptable values for these variables are as follows:

P_os:
P_win32 = WINDOWS
P_vms = VMS
P_mac = MAC
P_dgux = DGUX

P_hpux = HPUX
P_linux = Linux

P_compiler:
P_msvc4 = Visual C++ 4.0 and later versions
P_bcb6 = Borland C++ Builder 6.0 and later versions
P_wat10 = WATCOM C 10.0
P_symantec = Symantec C++ Compiler // Nance 5/20/96
P_metrowerks = Metrowerks
P_ansi = ANY OLD ANSI C++ COMPILER
P_sas = SAS C++ Compiler (IBM Mainframe)
P_gnu = GNU GCC

P_linkage:
P_export = generate code for placement in a DLL (used when rebuilding RC21 only
P_import = generate code to import RC21 functions from a DLL
P_static = link with static library

P_service:
P_dedicated = single-user (embedded)
P_client = your code talks to an RC21 multi-user server
P_hybrid = combination of dedicated and client