CIFTable
A Library of Dynamic, Persistent, and Searchable Tables
Reference Guide

CIFTable Version 4.01

Version 1.0 Fall 1996
Version 3.01 August 1997
Version 4.01 August 1999

Steven Schirripa
John D. Westbrook
Nucleic Acid Database Project
Department of Chemistry
Rutgers, The State University of New Jersey

Please direct comments on this document to jwest@ndb.rutgers.edu.

CIFTable - A Library of Dynamic, Persistent, and Searchable Tables

Copyright © 1997 Rutgers,The State University of New Jersey

This software is provided WITHOUT WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR ANY OTHER WARRANTY, EXPRESS OR IMPLIED. RUTGERS MAKE NO REPRESENTATION OR WARRANTY THAT THE SOFTWARE WILL NOT INFRINGE ANY PATENT, COPYRIGHT OR OTHER PROPRIETARY RIGHT.

The user of this software shall indemnify, hold harmless and defend Rutgers, its governors, trustees, officers, employees, students, agents and the authors against any and all claims, suits, losses, liabilities, damages, costs, fees, and expenses including reasonable attorneys' fees resulting from or arising out of the use of this software. This indemnification shall include, but is not limited to, any and all claims alleging products liability.

This software may be used only for not-for-profit educational and research purposes.

Contents

1 Introduction

The CIFTable package contains several files that converge together in the SSTable class, a *s*earchable *s*tring *table*. The SSTable class has a number of characteristics that make it useful for a wide variety of applications:

- SSTable objects are general to the datatypes supported. Other types can be stored as strings and converted at a higher level.
- SSTable objects are extremely flexible. Data can be inserted and removed easily, by row, by column, or by individual cell.
- SSTable objects provide persistent storage management of themselves.
- SSTable objects have an efficient searching mechanism built internally. This mechanism allows for multiple column searching with logical conjunction and intersection.

This document describes the public and static interface to the SSTable class.

2 SSTable Public Method and Member Descriptions

The following section describes the public interface to the SSTable class. Example code will be provided along with each method or member.

2.1 Constructors

This subsection contains the constructors for the SSTable class.

2.1.1 SSTable

NAME

SSTable

PROTOTYPE

```
#include "SSTable.h"

SSTable::SSTable();
SSTable::SSTable(const char * label);
```

EXAMPLE

```
#include "SSTable.h"

SSTable Table;
SSTable * pTable = new SSTable();

SSTable Table2("table2);
SSTable * pTable2 = new SSTable("ptable2");
```

PURPOSE

SSTable() is the default constructor for the SSTable class.
*SSTable(const char *)* is a constructor for the SSTable class that also sets the table's name.

RECEIVES

SSTable()

No Arguments

SSTable(const char *)

`label`	The name of the table

RETURN VALUE

None

REMARKS

None

2.2 Column methods

This subsection contains methods for column creation, filling, and access.

2.2.1 AddColumn

NAME

AddColumn

PROTOTYPE

```
#include "SSTable.h"

int SSTable::AddColumn(const char * newColumnName,
                       int makeTree = 0,
                       char opts = DEFAULT_OPTIONS);
```

EXAMPLE

```
#include "SSTable.h"
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif

SSTable * pTable = new SSTable();
pTable->AddColumn("NextToLastColumn");
pTable->AddColumn("LastColumn", TRUE, SSTable::DT_STRING |
  SSTable::CASE_SENSE | SSTable::W_SPACE_SENSE);
```

PURPOSE

AddColumn appends a column to the table, but *does not* add data to the column yet. Use FillColumn to add data to the whole column.

RECEIVES

`newColumnName`	The name of the new column. The column name must be unique to the table.
`makeTree`	If zero, no index will be constructed on the column, which will not be searchable. Set to a positive number if the column is to be searchable. Has a default value of 0.
`opts`	Sets the options for the column. These include the datatype, case sensitivity, and white space sensitivity for comparisions. Has a default value indicating a column of strings, with case and white space sensitivity.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *FillColumn*
Column Option Constants

2.2.2 InsertColumn

NAME

InsertColumn

PROTOTYPE

```
#include "SSTable.h"
int SSTable::InsertColumn(const char * newColumnName,
                          int destination, int makeTree = 0,
                          char opts = DEFAULT_OPTIONS);

int SSTable::InsertColumn(ReVarCifArray<CifString> & theCol,
                          const char * newColumnName,
                          int destination, int makeTree = 0,
                          char opts = DEFAULT_OPTIONS);
```

EXAMPLE

```
#include "SSTable.h"
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif
...
SSTable * pTable = new SSTable();
ReVarCifArray<CifString> last_column;
CifString cs;

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  last_column.Add(cs);
  cs.Clear();
}

// put a column in 1st position ...
pTable->InsertColumn("FirstColumn", 0);
pTable->InsertColumn(last_column, "LastColumn",
  pTable->GetNumColumns(), TRUE);
pTable->InsertColumn("MiddleColumn", pTable->GetNumColumns()/2,
  TRUE, SSTable::DT_INTEGER);
```

PURPOSE

InsertColumn inserts a column into the table at a specified location. The second form of this method also adds data into the table. The first form of the method requires calling another method, such as FillColumn, to add data to the table.

RECEIVES

InsertColumn(const char *, int, int, char)

`newColumnName`	The name of the new column. The column name must be unique to the table.
`destination`	The index where the column is to be placed. Keep in mind that the first column is column 0.
`makeTree`	If zero, no index will be constructed on the column, which will not be searchable. Set to a positive number if the column is to be searchable. Has a default value of 0.
`opts`	Sets the options for the column. These include the datatype, case sensitivity, and white space sensitivity for comparisions. Has a default value indicating a column of strings, with case and white space sensitivity.

InsertColumn(ReVarCifArray<CifString>&, const char *, int, int, char)

`theCol`	An array of CifStrings to add as the data of the column. Even if the datatype of the column is not of string type, it needs to be put into the table as CifStrings.
`newColumnName`	The name of the new column. The column name must be unique to the table.
`destination`	The index where the column is to be placed. Keep in mind that the first column is column 0.
`makeTree`	If zero, no index will be constructed on the column, which will not be searchable. Set to a positive number if the column is to be searchable. Has a default value of 0.
`opts`	Sets the options for the column. These include the datatype, case sensitivity, and white space sensitivity for comparisions. Has a default value indicating a column of strings, with case and white space sensitivity.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *FillColumn*
Column Option Constants

2.2.3 FillColumn

NAME

FillColumn

PROTOTYPE

```
#include "SSTable.h"

int SSTable::FillColumn(ReVarCifArray<CifString> & theCol,
                        int colIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();
ReVarCifArray<CifString> some_column;
CifString cs;
int errCode = 0;

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  some_column.Add(cs);
  cs.Clear();
}

if ( (errCode = pTable->AddColumn("LastColumn", 0)) >= 0)
  pTable->FillColumn(some_column, pTable->GetColumnIndex(
    "LastColumn"));
else
  log_error(SSTable::GetErrorMessage(errCode));
```

PURPOSE

FillColumn puts an array of CifStrings into a specified column as its data.

RECEIVES

`theCol`	An array of CifStrings to place into the column as data.
`colIndex`	The index of the column where the data is to be placed. Keep in mind that the first column is column 0.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

None

2.2.4 AppendToColumn

NAME

AppendToColumn

PROTOTYPE

```
#include "SSTable.h"

int SSTable::AppendToColumn(ReVarCifArray<CifString> & theCol,
                            int colIndex);
int SSTable::AppendToColumn(ReVarCifArray<CifString> & theCol,
                            CifString & colName);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();
ReVarCifArray<CifString> some_column;
CifString cs;
int errCode = 0;

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  some_column.Add(cs);
  cs.Clear();
}

cs = "LastColumn";

if ( (errCode = pTable->AddColumn(cs.Text(), 0)) >= 0) {
  pTable->FillColumn(some_column, pTable->GetColumnIndex(
    cs.Text()));
  // Double the data!!
  pTable->AppendToColumn(some_column, pTable->GetColumnIndex(
    cs.Text()));
  // Triple it!!!
  pTable->AppendToColumn(some_column, cs);
} else
  log_error(SSTable::GetErrorMessage(errCode));
```

PURPOSE

AppendToColumn appends an array of CifStrings to the data of the specified column. This is ok even if the data goes beyond the number of rows, but it can lead to non-rectangular tables. The second form of this method is a convenience method to avoid explicit column index lookup.

RECEIVES

AppendToColumn(ReVarCifArray<CifString> &, int)

`theCol`	An array of CifStrings to append to the column.
`colIndex`	The index of the column where the data is to be appended. Keep in mind that the first column is column 0.

AppendToColumn(ReVarCifArray<CifString> &, CifString &)

`theCol`	An array of CifStrings to append to the column.
`colName`	The name of the column where the data is to be appended.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

None

2.2.5 AddElementToColumn

NAME

AddElementToColumn

PROTOTYPE

```
#include "SSTable.h"

int SSTable::AddElementToColumn(CifString & theElement, int colIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();
ReVarCifArray<CifString> some_column;
CifString cs;
int errCode = 0, i, index;

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  some_column.Add(cs);
  cs.Clear();
}

cs = "LastColumn";
index = pTable->GetColumnIndex(cs.Text());

if ( (errCode = pTable->AddColumn(cs.Text(), 0)) >= 0) {
  for (i = 0; i < some_column.Length(); i++)
    pTable->AddElementToColumn(some_column[i], index);
} else
  log_error(SSTable::GetErrorMessage(errCode));
```

PURPOSE

AddElementToColumn appends a CifString to the data of the specified column. The string is placed in the first available row of the column.

RECEIVES

`theElement`	The CifString to add to the column.
`colName`	The name of the column where the data is to be added.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

None

2.2.6 ClearColumn

NAME

ClearColumn

PROTOTYPE

```
#include "SSTable.h"

int SSTable::ClearColumn(int colIndex);
```

EXAMPLE

```
#include "SSTable.h"
extern void processTableColumn(SSTable &, int);
...
SSTable * pTable = new SSTable();
ReVarCifArray<CifString> last_column;
CifString cs;

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  last_column.Add(cs);
  cs.Clear();
}

pTable->InsertColumn(last_column, "LastColumn",
  pTable->GetNumColumns(), TRUE);

// process the last column
processTableColumn(*pTable, pTable->GetColumnIndex(
  "LastColumn"));

// clear the last column
pTable->ClearColumn(pTable->GetColumnIndex("LastColumn"));

last_column.Clear(); // clean up array and restock with new data
while (tokenizer2.hasMoreTokens()) {
  cs = (char *) tokenizer2.GetNextToken();
  last_column.Add(cs);
  cs.Clear();
}
```

```
// fill the column again and reprocess it, since ClearColumn keeps
// the column there (ONLY USE FillColumn here!!! Having a column
// smaller than it was before may lead to undesireable results. See
// "Remarks", below)
pTable->FillColumn(last_column, pTable->GetColumnIndex(
  "LastColumn"));
processTableColumn(*pTable, pTable->GetColumnIndex(
  "LastColumn"));
```

PURPOSE

ClearColumn erases all of the data in a particular column. It does not remove the column from the table; rather, it keeps placeholders. Thus, the number of columns and rows will not change as a result of this operation.

RECEIVES

`colIndex`	The index of the column to be cleared.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

This method may be moved to the private section, since it is too easy to misuse. FillColumn is the only method that will refill a column properly after a call to ClearColumn. The proper method for refilling a column is to use DeleteColumn and then any method to add data to a column. The other method is to use RemoveColumn followed by InsertColumn.

See also: *DeleteColumn*
RemoveColumn

2.2.7 DeleteColumn

NAME

DeleteColumn

PROTOTYPE

```
#include "SSTable.h"
int SSTable::DeleteColumn(int colIndex);
```

EXAMPLE

```
#include "SSTable.h"
...
SSTable * pTable = new SSTable();
ReVarCifArray<CifString> last_column;
CifString cs;

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  last_column.Add(cs);
  cs.Clear();
}

pTable->InsertColumn(last_column, "LastColumn",
  pTable->GetNumColumns(), TRUE);

// process the last column
processTableColumn(*pTable, pTable->GetColumnIndex(
  "LastColumn"));

// erase the last column
pTable->DeleteColumn(pTable->GetColumnIndex("LastColumn"));

last_column.Clear(); // clean up array and restock with new data
while (tokenizer2.hasMoreTokens()) {
  cs = (char *) tokenizer2.GetNextToken();
  last_column.Add(cs);
  cs.Clear();
}

// fill the column again and reprocess it, since DeleteColumn
```

```
// keeps the column there (any column filler can be used here)
pTable->FillColumn(last_column, pTable->GetColumnIndex(
  "LastColumn"));
processTableColumn(*pTable, pTable->GetColumnIndex(
  "LastColumn"));
```

PURPOSE

DeleteColumn will remove the data from a column, but not the column itself. This method is safer to use than ClearColumn unless the peculiar behavior of ClearColumn is desired.

RECEIVES

`colIndex`	The index of the column to be deleted.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

Using DeleteColumn ensures that filling or appending data to a column results in proper row-wise positioning. For example, if a 10 element column is erased with ClearColumn and then AppendElementToColumn is called, the element will be added to the 11th position with 10 "blank" placeholders ahead of it. If DeleteColumn is used, then there will only be 1 element in the column.

See also: *ClearColumn*
RemoveColumn

2.2.8 RemoveColumn

NAME

RemoveColumn

PROTOTYPE

```
#include "SSTable.h"
int SSTable::RemoveColumn(int colIndex);
```

EXAMPLE

```
#include "SSTable.h"
...
SSTable * pTable = new SSTable();
ReVarCifArray<CifString> last_column;
CifString cs;

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  last_column.Add(cs);
  cs.Clear();
}

pTable->InsertColumn(last_column, "LastColumn",
  pTable->GetNumColumns(), TRUE);

// process the last column
processTableColumn(*pTable, pTable->GetColumnIndex(
  "LastColumn"));

// erase the last column, then re-add it
pTable->RemoveColumn(pTable->GetColumnIndex("LastColumn"));
pTable->InsertColumn("LastColumn", pTable->GetNumColumns(),
  TRUE);

last_column.Clear(); // clean up array and restock with new data
while (tokenizer2.hasMoreTokens()) {
  cs = (char *) tokenizer2.GetNextToken();
  last_column.Add(cs);
  cs.Clear();
}
```

```
// fill the column again and reprocess it, since we reinserted it.
pTable->FillColumn(last_column, pTable->GetColumnIndex(
  "LastColumn"));
processTableColumn(*pTable, pTable->GetColumnIndex(
  "LastColumn"));
```

PURPOSE

RemoveColumn will remove the data from a column, and then remove the column itself.

RECEIVES

`colIndex`	The index of the column to be removed.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *ClearColumn*
DeleteColumn

2.2.9 GetColumn

NAME

GetColumn

PROTOTYPE

```
#include "SSTable.h"
ReVarCifArray<CifString> * SSTable::GetColumn(int colIndex);
```

EXAMPLE

```
#include "SSTable.h"
extern void processColumn(ReVarCifArray<CifString> *);

SSTable * pTable = new SSTable();
ReVarCifArray<CifString> * col;

... // fill up table

for (int i = 0; i < pTable->GetNumColumns(); i++) {
  col = pTable->GetColumn(i);
  if (col) {
    processColumn(col);
    pTable->FillColumn(*col, i);
  }
}
```

PURPOSE

GetColumn returns an array of strings of the data of a column.

RECEIVES

`colIndex`	The index of the column to be retrieved.

RETURN VALUE

A pointer to a ReVarCifArray<CifString> that contains the data of the specified column.

A NULL value will be returned if an error occurs.

REMARKS

See also: *GetSubColumn*

2.2.10 GetSubColumn

NAME

GetSubColumn

PROTOTYPE

```
#include "SSTable.h"
ReVarCifArray<CifString> * SSTable::GetSubColumn(int colIndex,
                                     int from, int to);
```

EXAMPLE

```
#include "SSTable.h"
extern void processColumn(ReVarCifArray<CifString> *);

SSTable * pTable = new SSTable();
SSTable * pTable2 = new SSTable();
ReVarCifArray<CifString> * subcol;
ReVarCifArray<CifString> * colNames;

... // fill up table pointed to by pTable, but leave pTable2 alone

colNames = pTable->GetColumnNames();
for (int i = 0; i < pTable->GetNumColumns(); i++) {
  // get upper half of the column
  subcol = pTable->GetSubColumn(i, 0, pTable->GetNumRows() / 2);
  if (subcol) {
    // process the subcolumn
    processColumn(subcol);
    // put into the new table
    pTable2->InsertColumn(*subcol, (*colNames[i]).Text(), i, 1);
  }
}
```

PURPOSE

GetSubColumn returns an array of strings representing a subcolumn of data.

RECEIVES

`colIndex`	The index of the column.
`from`	The index of the first row to be included in the subcolumn.
`to`	The index of the last row to be included in the subcolumn.

RETURN VALUE

A pointer to a ReVarCifArray<CifString> that contains the data of the specified subcolumn.
A NULL value will be returned if an error occurs.

REMARKS

See also: *GetColumn*

2.3 Row methods

This subsection contains methods for row creation, filling, and access.

2.3.1 AddRow

NAME

AddRow

PROTOTYPE

```
#include "SSTable.h"

int SSTable::AddRow();
```

EXAMPLE

```
#include "SSTable.h"
#include "FileNavigator.h"

FileNavigator * pFileNavigator = NULL;
int objectIndex;

... // FileNavigator initialization

SSTable * pTable = new SSTable();
pTable->GetObject(objectIndex, pFileNavigator);
pTable->AddRow();
...
```

PURPOSE

AddRow appends a row to the table, but *does not* add data to the row yet. Use FillRow or a similar method to add data to the row.

RECEIVES

```
No Arguments
```

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *InsertRow*

2.3.2 InsertRow

NAME

InsertRow

PROTOTYPE

```
#include "SSTable.h"
int SSTable::InsertRow(int rowIndex);
int SSTable::InsertRow(ReVarCifArray<CifString> & theRow,
                       int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"
#include "FileNavigator.h"

FileNavigator * pFileNavigator = NULL;

... // FileNavigator and tokenizer initialization

SSTable * pTable = new SSTable();
ReVarCifArray<CifString> some_row;
CifString cs;
int errCode = 0;

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  some_row.Add(cs);
  cs.Clear();
}

pTable->GetObject(objectIndex, pFileNavigator);
pTable->InsertRow(0); // Insert an empty 1st row
 // Insert a filled up middle row
pTable->InsertRow(some_row, pTable->GetNumRows()/2);
...
```

PURPOSE

InsertRow inserts a row into the table at a specified location. The second form of this method adds data into this row. The first form of the method requires calling another method, such as FillColumn, to add data to the table.

RECEIVES

InsertRow(int)

`rowIndex`	The index where the row is to be inserted. Keep in mind that the first row is row 0.

InsertRow(ReVarCifArray<CifString>&, int)

`theRow`	An array of CifStrings to add as the data of the row.
`rowIndex`	The index where the row is to be inserted. Keep in mind that the first row is row 0.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *AddRow*

2.3.3 FillRow

NAME

FillRow

PROTOTYPE

```
#include "SSTable.h"

int SSTable::FillRow(ReVarCifArray<CifString> & theRow,
                     int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();
ReVarCifArray<CifString> some_row;
CifString cs;
int errCode = 0;

...//table initialization

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  some_row.Add(cs);
  cs.Clear();
}

if ( (errCode = pTable->AddRow()) >= 0)
  pTable->FillRow(some_row, pTable->GetNumRows() - 1);
else
  log_error(SSTable::GetErrorMessage(errCode));
```

PURPOSE

FillRow puts an array of CifStrings into a specified row as its data.

RECEIVES

`theRow`	An array of CifStrings to place into the row as data.
`rowIndex`	The index of the row where the data is to be placed. Keep in mind that the first row is row 0.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

None

2.3.4 AppendToRow

NAME

AppendToRow

PROTOTYPE

```
#include "SSTable.h"

int SSTable::AppendToRow(ReVarCifArray<CifString> & theRow,
                         int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();
ReVarCifArray<CifString> some_row;
CifString cs;
int errCode = 0;

...//table initialization

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  some_row.Add(cs);
  cs.Clear();
}

if ( (errCode = pTable->AddRow()) >= 0)
  pTable->AppendToRow(some_row, pTable->GetNumRows() - 1);
else
  log_error(SSTable::GetErrorMessage(errCode));
```

PURPOSE

AppendToRow appends an array of CifStrings to the data of the specified row. This method is not recommended. See "Remarks" below.

RECEIVES

`theRow`	An array of CifStrings to append to the row.
`rowIndex`	The index of the row where the data is to be appended. Keep in mind that the first row is row 0.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

It is not recommended to use this method. It may be unpredictable to use this method on a row with empty strings, if some of the columns are searchable. The tables destinguish "" from placeholders only if the column is searchable (since empty strings may be desired as a data element). To properly handle with this, use UpdateCell on "" and AddElementToRow if a "" does not appear (i.e. iterate over the cells in the row).

2.3.5 AddElementToRow

NAME

AddElementToRow

PROTOTYPE

```
#include "SSTable.h"

int SSTable::AddElementToRow(CifString & theElement, int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();
CifString cs;
int errCode = 0, currentRow;

... // table initialization

while (tokenizer.hasMoreTokens()) {
  cs = (char *) tokenizer.GetNextToken();
  pTable->AddElementToRow(cs, currentRow);
  cs.Clear();
}
```

PURPOSE

AddElementToRow adds one CifString to a specified row. The column where the string is placed is determined by looking for the last non-null string in an unsearchable column or last null string in a searchable column and then placing the new element to the right.

RECEIVES

`theElement`	The CifString to add to the row.
`rowIndex`	The index of the row where the CifString is to be added. Keep in mind that the first row is row 0.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

None

2.3.6 ClearRow

NAME

ClearRow

PROTOTYPE

```
#include "SSTable.h"

int SSTable::ClearRow(int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"
...
SSTable * pTable = new SSTable();

... // do table maniputation

// clear the first 1/2 rows, but keep them as placeholders
for (int i = 0; i < (pTable->GetNumRows() / 2); i++) {
  pTable->ClearRow(i);
}
```

PURPOSE

ClearRow erases all of the data in a particular row. It does not remove the row from the table; rather, it keeps placeholders. Thus, the number of columns and rows will not change as a result of this operation.

RECEIVES

`rowIndex`	The index of the row to be cleared.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *DeleteRow*

2.3.7 DeleteRow

NAME

DeleteRow

PROTOTYPE

```
#include "SSTable.h"
int SSTable::DeleteRow(int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"
...
SSTable * pTable = new SSTable();

... // do table maniputation

// delete the first 1/2 rows
for (int i = 0; i < (pTable->GetNumRows() / 2); i++) {
  pTable->DeleteRow(i);
}
```

PURPOSE

DeleteRow will remove the data from a row, and then the row itself. This is safest method to get rid of a whole row of data. The row can always be reinserted later.

RECEIVES

`rowIndex`	The index of the row to be deleted.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *ClearRow*

2.3.8 GetRow

NAME

GetRow

PROTOTYPE

```
#include "SSTable.h"
ReVarCifArray<CifString> * SSTable::GetRow(int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"
extern void processRow(ReVarCifArray<CifString> *);

SSTable * pTable = new SSTable();
ReVarCifArray<CifString> * row;

... // fill up table

for (int i = 0; i < pTable->GetNumRows(); i++) {
  row = pTable->GetRow(i);
  if (row) {
    processRow(row);
    pTable->FillRow(*row, i);
  }
}
```

PURPOSE

GetRow returns the data of a row as an array of strings.

RECEIVES

`rowIndex`	The index of the row to be retrieved.

RETURN VALUE

A pointer to a ReVarCifArray<CifString> that contains the data of the specified row.

A NULL value will be returned if an error occurs.

REMARKS

See also: *GetSubRow*

2.3.9 GetSubRow

NAME

GetSubRow

PROTOTYPE

```
#include "SSTable.h"
ReVarCifArray<CifString> * SSTable::GetSubRow(int rowIndex,
                                      int from, int to);
```

EXAMPLE

```
#include "SSTable.h"
extern void processRow(ReVarCifArray<CifString> *);

SSTable * pTable = new SSTable();
SSTable * pTable2 = new SSTable();
ReVarCifArray<CifString> * subrow;

... // fill up table pointed to by pTable, but leave pTable2 alone

for (int i = 0; i < pTable->GetNumRows(); i++) {
  // get left half of the rows
  subrow = pTable->GetSubRow(i, 0, pTable->GetNumColumns() / 2);
  if (subrow) {
    // process the subrow
    processRow(subrow);
    // put into the new table
    pTable2->InsertRow(*subrow, i);
  }
}
```

PURPOSE

GetSubRow returns as the data of a subrow an array of strings.

RECEIVES

`rowIndex`	The index of the row.
`from`	The index of the first column to be included in the subrow.
`to`	The index of the last column to be included in the subrow.

RETURN VALUE

A pointer to a ReVarCifArray<CifString> that contains the data of the specified subrow.
A NULL value will be returned if an error occurs.

REMARKS

See also: *GetRow*

2.4 Cell methods

This subsection contains methods for cell filling, update, and access.

2.4.1 ClearAt

NAME

ClearAt

PROTOTYPE

```
#include "SSTable.h"

int SSTable::ClearAt(int colIndex, int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();

... // do table maniputation

// clear all non-diagonal data
for (int i = 0; i < pTable->GetNumColumns(); i++) {
  for (int j = 0; j < pTable->GetNumRows(); j++) {
    if (i != j)
      pTable->ClearAt(i, j);
  }
}
```

PURPOSE

ClearAt erases all the data of a particular cell. It does not remove the cell from the table; rather, it keeps a placeholder. Thus, the overall "shape" of the table will not change.

RECEIVES

`colIndex`	The index of the column of the cell to be cleared.
`rowIndex`	The index of the row of the cell to be cleared.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *DeleteAt*

2.4.2 DeleteAt

NAME

DeleteAt

PROTOTYPE

```
#include "SSTable.h"

int SSTable::DeleteAt(int colIndex, int rowIndex,
                      int updateNumRows = 1);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();

... // do table maniputation

// Prune all bad values from table
for (int i = pTable->GetNumColumns() - 1; i >= 0; i--) {
  for (int j = pTable->GetNumRows() - 1; j >= 0; j--) {
    CifString cs;
    pTable->GetCell(cs, i, j);
    if (!cs.Compare(BAD_VALUE))
      pTable->DeleteAt(i, j);
  }
}
```

PURPOSE

DeleteAt erases all the data of a particular cell. It also removes the cell from the table. WARNING: the shape of the table may change.

RECEIVES

`colIndex`	The index of the column of the cell to be deleted.
`rowIndex`	The index of the row of the cell to be deleted.
`updateNumRows`	Used internally to avoid recompution of the number of rows. This parameter does not need to be bothered with. The default is to recompute. This is the safest thing anyway, so ignore this parameter.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *ClearAt*

2.4.3 UpdateCell

NAME

UpdateCell

PROTOTYPE

```
#include "SSTable.h"

int SSTable::UpdateCell(CifString & theElement, int colIndex,
                        int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();
CifString zero("0");

... // do table maniputation

// change all non-diagonals to "0"
for (int i = 0; i < pTable->GetNumColumns(); i++) {
  for (int j = 0; j < pTable->GetNumRows(); j++) {
    if (i != j)
      pTable->UpdateCell(zero, i, j);
  }
}
```

PURPOSE

UpdateCell changes the data of a particular cell.

RECEIVES

`theElement`	The CifString to update the cell with.
`colIndex`	The index of the column of the cell to be updated.
`rowIndex`	The index of the row of the cell to be updated.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

None

2.4.4 GetCell

NAME

GetCell

PROTOTYPE

```
#include "SSTable.h"

int SSTable::GetCell(CifString & theCell, int colIndex,
                     int rowIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();

... // do table maniputation

// Clear all bad values in table
for (int i = 0; i < pTable->GetNumColumns(); i++) {
  for (int j = 0; j < pTable->GetNumRows(); j++) {
    CifString cs;
    pTable->GetCell(cs, i, j);
    if (!cs.Compare(BAD_VALUE))
      pTable->ClearAt(i, j);
  }
}
```

PURPOSE

GetCell gets a copy of the CifString in a particular cell.

RECEIVES

`theCell`	A CifString reference that is to contain the copied cell CifString.
`colIndex`	The index of the column of the cell.
`rowIndex`	The index of the row of the cell.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

None

2.5 Searching Methods

This subsection has methods used for searching and constraining the table. See the SetIntersect and SetUnion methods in the "Static Methods" section.

2.5.1 SearchColumn

NAME

SearchColumn

PROTOTYPE

```
#include "SSTable.h"

ReVarPCifArray<int> * SSTable::SearchColumn(CifString & target,
                                           CifString & colName,
                                           int & errCode);
ReVarPCifArray<int> * SSTable::SearchColumn(CifString & target,
                                           int colIndex,
                                           int & errCode);
```

EXAMPLE

```
#include "SSTable.h"

SSTable s("MyTable");
...
CifString cs("Target"), col1("column1");
CifString newCS("New Value");
int errCode = 0;

ReVarPCifArray<int> * iArray;
iArray = s.SearchColumn(cs, col1, errCode);
if (iArray) {
  // update each instance of "Target" with "New Value"
  int colIndex = s.GetColumnIndex(col1.Text());
  for (int i = 0; i < iArray->Length(); i++) {
    s.UpdateCell(newCS, colIndex, *iArray[i]);
  }
}
```

PURPOSE

SearchColumn returns the row indices of cells in a column that are equal to a target. You can only search a column that was constructed with a tree index.

RECEIVES

SearchColumn(CifString &, CifString &, int &)

`target`	The target string to search for.
`colName`	The name of the column to search.
`errCode`	A reference to an integer holding the error code resulting from this operation.

SearchColumn(CifString &, int, int &)

`target`	The target string to search for.
`colIndex`	The index of the column to search.
`errCode`	A reference to an integer holding the error code resulting from this operation.

RETURN VALUE

A pointer to a ReVarPCifArray<int> holding the row indices of the cells containing the target string.

A NULL value indicates a possible error or an unsuccessful search.

A negative value in errCode indicates an error or warning.

REMARKS

See also: *SearchColumns*

2.5.2 SearchColumns

NAME

SearchColumns

PROTOTYPE

```
#include "SSTable.h"

ReVarPCifArray<int> * SSTable::SearchColumns(
                        ReVarCifArray<CifString> & targets,
                        ReVarCifArray<CifString> & colNames,
                        int & errCode);
ReVarPCifArray<int> * SSTable::SearchColumns(
                        ReVarCifArray<CifString> & targets,
                        ReVarPCifArray<int> & colIds,
                        int & errCode);
```

EXAMPLE

```
#include "SSTable.h"

SSTable s("MyTable");
...
CifString cs("Hello"), cs2("World"), col1("A"), col2("B"), col3("C");
CifString newCS("!");
int errCode = 0;

ReVarCifArray<CifString> vals, names;
vals.Add(cs); vals.Add(cs2);
names.Add(col1); names.Add(col2);

ReVarPCifArray<int> * iArray;

// Search column "A" for "Hello" and column "B" for "World"
iArray = s.SearchColumns(vals, names, errCode);
if (iArray) {
  int colIndex = s.GetColumnIndex(col3.Text());
  for (int i = 0; i < iArray->Length(); i++) {
    s.UpdateCell(newCS, colIndex, *iArray[i]);
  }
}
```

PURPOSE

SearchColumns returns the row indices where all of the target strings match in their respective columns. This method is the equalivalent of searching multiple columns with SearchColumn and then doing a SetIntersect on them. It may be more efficient though. You can only search columns that were constructed with a tree index.

RECEIVES

SearchColumns(ReVarCifArray<CifString>&, ReVarCifArray<CifString>&, int &)

`targets`	An array of target strings to search for.
`colName`	An array of column names indicating which columns should be searched.
`errCode`	A reference to an integer holding the error code resulting from this operation.

SearchColumns(ReVarCifArray<CifString>&, ReVarPCifArray<int>&, int &)

`targets`	An array of target strings to search for.
`colIds`	An array of column indices indicating which columns should be searched.
`errCode`	A reference to an integer holding the error code resulting from this operation.

RETURN VALUE

A pointer to a ReVarPCifArray<int> holding the row indices having matches in all of the columns specified.
A NULL value indicates a possible error or an unsuccessful search.
A negative value in errCode indicates an error or warning.

REMARKS

See also: *SearchColumn*

2.5.3 ConstrainColumnLessThan

NAME

ConstrainColumnLessThan

PROTOTYPE

```
#include "SSTable.h"

ReVarPCifArray<int> * SSTable::ConstrainColumnLessThan(
                                  CifString & target,
                                  CifString & colName,
                                  int & errCode);
```

EXAMPLE

```
#include "SSTable.h"

CifString cs("30.00"), col1("twist");
int errCode = 0;
SSTable s("MyTable");
...
s.SetFlags(SSTable::DT_DOUBLE, s.GetColumnIndex(col1.Text()));
s.SetPrecision(2, s.GetColumnIndex(col1.Text()));

ReVarPCifArray<int> * iArray;
iArray = s.ConstrainColumnLessThan(cs, col1, errCode);
if (iArray) {
  for (int i = 0; i < iArray->Length(); i++) {
    // do something to each row with twist < 30.00
  }
}
```

PURPOSE

ConstrainColumnLessThan returns the row indices of cells in a column that are less than the target value. You can only search a column that was constructed with a tree index. The search is dictated by the datatype of the column, since, strings, integers, and doubles get compared differently.

RECEIVES

`target`	The target value to compare to.
`colName`	The name of the column to search.
`errCode`	A reference to an integer holding the error code resulting from this operation.

RETURN VALUE

A pointer to a ReVarPCifArray<int> holding the row indices of the cells containing a value less than the target.
A NULL value indicates a possible error or an unsuccessful search.
A negative value in errCode indicates an error or warning.

REMARKS

See also: *ConstrainColumnLessThanEqual*

2.5.4 ConstrainColumnLessThanEqual

NAME

ConstrainColumnLessThanEqual

PROTOTYPE

```
#include "SSTable.h"

ReVarPCifArray<int> * SSTable::ConstrainColumnLessThanEqual(
                                  CifString & target,
                                  CifString & colName,
                                  int & errCode);
```

EXAMPLE

```
#include "SSTable.h"

CifString cs("30.00"), col1("twist");
int errCode = 0;
SSTable s("MyTable");
...
s.SetFlags(SSTable::DT_DOUBLE, s.GetColumnIndex(col1.Text()));
s.SetPrecision(2, s.GetColumnIndex(col1.Text()));

ReVarPCifArray<int> * iArray;
iArray = s.ConstrainColumnLessThanEqual(cs, col1, errCode);
if (iArray) {
  for (int i = 0; i < iArray->Length(); i++) {
    // do something to each row with twist <= 30.00
  }
}
```

PURPOSE

ConstrainColumnLessThanEqual returns the row indices of cells in a column that are less than or equal to the target value. You can only search a column that was constructed with a tree index. The search is dictated by the datatype of the column, since, strings, integers, and doubles get compared differently.

RECEIVES

`target`	The target value to compare to.
`colName`	The name of the column to search.
`errCode`	A reference to an integer holding the error code resulting from this operation.

RETURN VALUE

A pointer to a ReVarPCifArray<int> holding the row indices of the cells containing a value less than or equal to the target.
A NULL value indicates a possible error or an unsuccessful search.
A negative value in errCode indicates an error or warning.

REMARKS

See also: *ConstrainColumnLessThan*

2.5.5 ConstrainColumnGreaterThan

NAME

ConstrainColumnGreaterThan

PROTOTYPE

```
#include "SSTable.h"

ReVarPCifArray<int> * SSTable::ConstrainColumnGreaterThan(
                                  CifString & target,
                                  CifString & colName,
                                  int & errCode);
```

EXAMPLE

```
#include "SSTable.h"

CifString cs("0.00"), col1("twist");
int errCode = 0;
SSTable s("MyTable");
...
s.SetFlags(SSTable::DT_DOUBLE, s.GetColumnIndex(col1.Text()));
s.SetPrecision(2, s.GetColumnIndex(col1.Text()));

ReVarPCifArray<int> * iArray;
iArray = s.ConstrainColumnGreaterThan(cs, col1, errCode);
if (iArray) {
  for (int i = 0; i < iArray->Length(); i++) {
    // do something to each row with twist > 0.00
  }
}
```

PURPOSE

ConstrainColumnGreaterThan returns the row indices of cells in a column that are greater than the target value. You can only search a column that was constructed with a tree index. The search is dictated by the datatype of the column, since, strings, integers, and doubles get compared differently.

RECEIVES

`target`	The target value to compare to.
`colName`	The name of the column to search.
`errCode`	A reference to an integer holding the error code resulting from this operation.

RETURN VALUE

A pointer to a ReVarPCifArray<int> holding the row indices of the cells containing a value greater than the target.

A NULL value indicates a possible error or an unsuccessful search.

A negative value in errCode indicates an error or warning.

REMARKS

See also: *ConstrainColumnGreaterThanEqual*

2.5.6 ConstrainColumnGreaterThanEqual

NAME

ConstrainColumnGreaterThanEqual

PROTOTYPE

```
#include "SSTable.h"

ReVarPCifArray<int> * SSTable::ConstrainColumnGreaterThanEqual(
                                 CifString & target,
                                 CifString & colName,
                                 int & errCode);
```

EXAMPLE

```
#include "SSTable.h"

CifString cs("0.00"), col1("twist");
int errCode = 0;
SSTable s("MyTable");
...
s.SetFlags(SSTable::DT_DOUBLE, s.GetColumnIndex(col1.Text()));
s.SetPrecision(2, s.GetColumnIndex(col1.Text()));

ReVarPCifArray<int> * iArray;
iArray = s.ConstrainColumnGreaterThanEqual(cs, col1, errCode);
if (iArray) {
  for (int i = 0; i < iArray->Length(); i++) {
    // do something to each row with twist >= 0.00
  }
}
```

PURPOSE

ConstrainColumnGreaterThanEqual returns the row indices of cells in a column that are greater than or equal to the target value. You can only search a column that was constructed with a tree index. The search is dictated by the datatype of the column, since, strings, integers, and doubles get compared differently.

RECEIVES

`target`	The target value to compare to.
`colName`	The name of the column to search.
`errCode`	A reference to an integer holding the error code resulting from this operation.

RETURN VALUE

A pointer to a ReVarPCifArray<int> holding the row indices of the cells containing a value greater than or equal to the target.
A NULL value indicates a possible error or an unsuccessful search.
A negative value in errCode indicates an error or warning.

REMARKS

See also: *ConstrainColumnGreaterThan*

2.6 Persistence Methods

This subsection describes the methods used for the persistent storage of the table object.

2.6.1 WriteObject

NAME

WriteObject

PROTOTYPE

```
#include "SSTable.h"
#include "FileNavigator.h"

int SSTable::WriteObject(FileNavigator * fnav);
```

EXAMPLE

```
#include "SSTable.h"
#include "FileNavigator.h"

const int MAX_TABLES = 50;
const char * NAV_FILE = "table.odb";

SSTable * pTable = new SSTable[MAX_TABLES];
int * table_loc = new int[MAX_TABLES];
FileNavigator * f = new FileNavigator();
f->OpenFile(NAV_FILE, WRITE_MODE);
f->ReadFileHeader();

... // do table construction/maniputation

// Write out all the tables...
Word list_loc = 0, place = 0;
f->WriteWord(0, list_loc);

for (int i = 0; i < MAX_TABLES; i++) {
  table_loc[i] = pTable[i].WriteObject(f);
}
```

```
f->WriteWords((Word *) table_locs, MAX_TABLES, place);
f->UpdateWord(place, list_loc, list_loc);
f->CloseFile();
```

PURPOSE

WriteObject dumps out the object to a persistent storage file.

RECEIVES

`fnav`	A pointer to the FileNavigator object responsible for writing the persistent storage file.

RETURN VALUE

A negative value indicates an error or warning.

A non-negative value indicates the index of the object, needed for reconstruction.

REMARKS

See also: *GetObject*

2.6.2 GetObject

NAME

GetObject

PROTOTYPE

```
#include "SSTable.h"
#include "FileNavigator.h" // defines Word (4-byte integer)

int SSTable::GetObject(Word index, FileNavigator * fnav);
```

EXAMPLE

```
#include "SSTable.h"
#include "FileNavigator.h"

const char * NAV_FILE = "table.odb";
FileNavigator * f = new FileNavigator();
f->OpenFile(NAV_FILE, READ_MODE);
f->ReadFileHeader();
SSTable * pTable = NULL;
Word where = 0, errCode = 0;

where = f->GetWord(0, errCode); // the location of the list
if (errCode < 0) f->PrintError(errCode);
uWord numT = 0;
int * table_loc = (int *) f->GetWords(where, numT, errCode);
pTable = new SSTable[numT];

for (int i = 0; i < numT; i++) {
  pTable[i].GetObject(table_loc[i], f);
}

f->CloseFile();
```

PURPOSE

GetObject loads the table object from a persistent storage file.

RECEIVES

`index`	The index of the object (i.e. its location in the file).
`fnav`	A pointer to the FileNavigator object responsible for reading the persistent storage file.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *WriteObject*

2.7 General Methods

This subsection contains miscellaneous methods and members, including those for general table access and manipulation.

2.7.1 isIndexed

NAME

isIndexed

PROTOTYPE

```
#include "SSTable.h"

int SSTable::isIndexed(int colIndex);
```

PURPOSE

isIndexed tells whether a given column is searchable or not.

RECEIVES

`colIndex`	The index of the column

RETURN VALUE

Returns 0 if the column is not searchable or the index is out of range. Otherwise, the column is searchable and a value of 1 is returned.

REMARKS

None

2.7.2 Column Option Constants

PROTOTYPE

```
#include "SSTable.h"

const char SSTable::CASE_SENSE;
const char SSTable::CASE_INSENSE;
const char SSTable::W_SPACE_SENSE;
const char SSTable::W_SPACE_SENSE;
const char SSTable::DT_STRING;
const char SSTable::DT_INTEGER;
const char SSTable::DT_DOUBLE;
```

PURPOSE

CASE_SENSE is the bit field for setting comparisons to be case sensitive.
CASE_INSENSE is the bit field for setting comparisons to be case insensitive.
W_SPACE_SENSE is the bit field for setting comparisons to be white space sensitive.
W_SPACE_INSENSE is the bit field for setting comparisons to be white space insensitive.
DT_STRING is the value for a string datatype.
DT_INTEGER is the value for an integer datatype.
DT_DOUBLE is the value for a double datatype.

REMARKS

The first four constants are bitwise ORed together and passed as the "options" parameter to the table. They determine how string comparisions are done on a particular column.

The other three values are the values for the three supported primitive datatypes. You can bitwise OR these with the "options". They determine how values in a column get compared. The reason for this is illustrated by a simple example. Suppose the numbers 9.9 and 10.1 are in a column. If the datatype of the column is DT_STRING, 9.9 comes after 10.1. This is because 1 comes before 9. If the datatype was correctly assigned to DT_DOUBLE, 9.9 would come before 10.1.

2.7.3 SetFlags

NAME

SetFlags

PROTOTYPE

```
#include "SSTable.h"

int SSTable::SetFlags(char newOpts, int colIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();

... // do table construction

// Set all columns to case insensitive strings
for (int i = 0; i < pTable->GetNumColumns(); i++) {
  pTable->SetFlags(SSTable::DT_STRING |
                   SSTable::CASE_INSENSE, i);
}
```

PURPOSE

SetFlags set the flags for a particular column. These flags dictate the datatype of the column and the behavior of searches done on the column.

RECEIVES

`newOpts`	A bit field comprised of the bitwise ORing of the desired options.
`colIndex`	The index of the column to have flags set.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

Note that setting one of case (in)sensitivity, white space (in)sensitivity, or datatype does not interfere with the current setting of the other two.

See also: *Column Option Constants*

2.7.4 GetDataType

NAME

GetDataType

PROTOTYPE

```
#include "SSTable.h"

int SSTable::GetDataType(int colIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();

... // do table maniputation

for (int i = 0; i < pTable->GetNumColumns(); i++) {
  int dt = pTable->GetDataType(i);
  switch (dt) {
    case SSTable::DT_STRING: ...; break;
    case SSTable::DT_INTEGER: ...; break;
    case SSTable::DT_DOUBLE: ...; break;
    default: ...; break;
  }
}
```

PURPOSE

GetDataType returns the datatype code for a column.

RECEIVES

`colIndex`	The index of the column in question

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *Column Option Constants*

2.7.5 SetPrecision

NAME

SetPrecision

PROTOTYPE

```
#include "SSTable.h"

int SSTable::SetPrecision(unsigned int newP, int colIndex);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();

... // do table construction

// Set all columns to doubles with 3 decimals precision
for (int i = 0; i < pTable->GetNumColumns(); i++) {
  pTable->SetFlags(SSTable::DT_DOUBLE, i);
  pTable->SetPrecision(3, i);
}
```

PURPOSE

SetPrecision sets the number of significant decimal places for the data in a column. This precision only has meaning for columns with a datatype of DT_DOUBLE.

RECEIVES

`newP`	The new precision, in decimal places.
`colIndex`	The index of the column to have precision set.

RETURN VALUE

A negative value indicates an error or warning.

REMARKS

See also: *Column Option Constants*

2.7.6 Rename

NAME

Rename

PROTOTYPE

```
#include "SSTable.h"

void SSTable::Rename(const char * label);
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable();
pTable->Rename("MyTable");
```

PURPOSE

Rename changes the table name.

RECEIVES

`label`	The new name for the table.

RETURN VALUE

None

REMARKS

See also: *GetName*

2.7.7 GetName

NAME

GetName

PROTOTYPE

```
#include "SSTable.h"

const char * SSTable::GetName();
```

EXAMPLE

```
#include <string.h> // for strcpy
#include "SSTable.h"
const int MAX_STRING_LEN = 255;

SSTable * pTable = new SSTable("MyTable");
char buffer[MAX_STRING_LEN + 1];
...
strcpy(buffer, pTable->GetName());
```

PURPOSE

GetName returns the name of the table.

RECEIVES

No Arguments

RETURN VALUE

A constant string representing the name of the table.

REMARKS

See also: *Rename*

2.7.8 GetNumColumns

NAME

GetNumColumns

PROTOTYPE

```
#include "SSTable.h"

int SSTable::GetNumColumns();
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable("MyTable");

// Add some columns and stuff, then loop over them

for (int i = 0; i < pTable->GetNumColumns(); i++) {
  // Do something to each column
}
```

PURPOSE

GetNumColumns returns the number of columns in the table.

RECEIVES

No Arguments

RETURN VALUE

The number of columns in the table.

REMARKS

See also: *GetNumRows*

2.7.9 GetNumRows

NAME

GetNumRows

PROTOTYPE

```
#include "SSTable.h"

int SSTable::GetNumRows();
```

EXAMPLE

```
#include "SSTable.h"

SSTable * pTable = new SSTable("MyTable");

// Add some columns and stuff, then loop over the rows

for (int i = 0; i < pTable->GetNumRows(); i++) {
  // Do something to each row
}
```

PURPOSE

GetNumRows returns the number of rows in the table. Each row might not be complete.

RECEIVES

No Arguments

RETURN VALUE

The number of rows in the table, i.e. the length of the longest column.

REMARKS

See also: *GetNumColumns*

2.7.10 GetColumnNames

NAME

GetColumnNames

PROTOTYPE

```
#include "SSTable.h"

ReVarCifArray<CifString> * SSTable::GetColumnNames();
```

EXAMPLE

```
#include <string.h>
#include <stdlib.h>
#include "SSTable.h"
extern char ** ConvertRVCACifString(ReVarCifArray<CifString>*);

SSTable * pTable = new SSTable("MyTable");

... // add some columns and stuff

ReVarCifArray<CifString> * cifarray = pTable->GetColumnNames();
char ** colNames = ConvertRVCACifString(cifarray);

// sort the names
qsort(colNames, pTable->GetNumColumns(), sizeof(char *), strcmp);

// do something to each column, alphabetically
for (int i = 0; i < pTable->GetNumColumns(); i++) {
  int index = pTable->GetColumnIndex(colNames[i]);
  ...
}
```

PURPOSE

GetColumnNames returns a list of the column names.

RECEIVES

No Arguments

RETURN VALUE

A pointer to the names of the columns returned as a ReVarCifArray<CifString>, and in column order.

REMARKS

None

2.7.11 GetColumnIndex

NAME

GetColumnIndex

PROTOTYPE

```
#include "SSTable.h"

int SSTable::GetColumnIndex(const char * columnName);
```

EXAMPLE

```
#include <string.h>
#include <stdlib.h>
#include "SSTable.h"
extern char ** ConvertRVCACifString(ReVarCifArray<CifString>*);

SSTable * pTable = new SSTable("MyTable");

... // add some columns and stuff

ReVarCifArray<CifString> * cifarray = pTable->GetColumnNames();
char ** colNames = ConvertRVCACifString(cifarray);

// sort the names
qsort(colNames, pTable->GetNumColumns(), sizeof(char *), strcmp);

// do something to each column, alphabetically
for (int i = 0; i < pTable->GetNumColumns(); i++) {
  int index = pTable->GetColumnIndex(colNames[i]);
  ...
}
```

PURPOSE

GetColumnIndex returns the index of the column with the given name.

RECEIVES

`columnName`	The name of the column.

RETURN VALUE

An integer representing the position of the column in the table. For example, an index of 0 represents the 1st column.
A negative value means that an error has occured, i.e. the column with the specified name was not found in the table.

REMARKS

None

2.8 Debugging Methods

This subsection has public methods used for debugging purposes only.

2.8.1 PrintTable

NAME

PrintTable

PROTOTYPE

```
#include "SSTable.h"

void SSTable::PrintTable();
```

PURPOSE

PrintTable prints a representation of the table to standard output, but is not guaranteed to have a nice look.

RECEIVES

No Arguments

RETURN VALUE

None

REMARKS

See also:	*Assert*
	PrintTree

2.8.2 PrintTree

NAME

PrintTree

PROTOTYPE

```
#include "SSTable.h"

void SSTable::PrintTree(int colIndex);
```

PURPOSE

PrintTree prints to standard output a representation of the table "tree" index on a column. This method is used for debugging.

RECEIVES

`colIndex`	The index of the column to have its tree representation printed.

RETURN VALUE

None

REMARKS

See also:	*Assert* *PrintTable*

2.8.3 Assert

NAME

Assert

PROTOTYPE

```
#include "SSTable.h"

int SSTable::Assert(int colIndex);
```

PURPOSE

Assert ensures internal consistency within the table. The assertion provided is very limited in nature. It should be used for debugging, if necessary.

RECEIVES

`colIndex`	The column to be asserted as consistent.

RETURN VALUE

A negative value indicates an inconsistency. Use GetErrorMessage to find out what it is.

REMARKS

See also: *GetErrorMessage*
PrintTable
PrintTree

3 SSTable Static Method Descriptions

This section describes the publicly accessible static methods of the SSTable.

3.0.1 SetIntersect

NAME

SetIntersect

PROTOTYPE

```
#include "SSTable.h"

static ReVarPCifArray<int> * SSTable::SetIntersect(
      ReVarPCifArray<int> * a, ReVarPCifArray<int> * b);
static ReVarPCifArray<int> * SSTable::SetIntersect(
      ReVarPCifArray<int> ** sets, int num);
```

EXAMPLE

```
#include "SSTable.h"

SSTable s("MyTable");
...
CifString cs("Target"), col1("column1");
CifString cs2("Target2"), col2("column2");
int errCode = 0;

ReVarPCifArray<int> * iArray;
iArray = SSTable::SetIntersect(s.SearchColumn(cs, col1, errCode),
                               s.SearchColumn(cs2, col2, errCode));
```

PURPOSE

*SetIntersect(ReVarPCifArray *, ReVarPCifArray *)* is a static method that returns the logical intersection of two sets of integers. *SetIntersect(ReVarPCifArray **, int)* is a static method that returns the logical intersection of any number of sets of integers.

RECEIVES

SetIntersect(ReVarCifArray<int> *, ReVarCifArray<int> *)

`a`	The first set to intersect.
`b`	The second set to intersect.

SetIntersect(ReVarCifArray<int> **, int)

`sets`	An array of pointers to sets of integers.
`num`	The number of elements in the given array.

RETURN VALUE

ReVarPCifArray<int> *, a pointer to a set representing the intersection of all sets passed to it.

REMARKS

These methods are useful for performing multiple column searches with logical AND operations.

3.0.2 SetUnion

NAME

SetUnion

PROTOTYPE

```
#include "SSTable.h"

static ReVarPCifArray<int> * SSTable::SetUnion(
          ReVarPCifArray<int> * a, ReVarPCifArray<int> * b);
static ReVarPCifArray<int> * SSTable::SetUnion(
          ReVarPCifArray<int> ** sets, int num);
```

EXAMPLE

```
#include "SSTable.h"

SSTable s("MyTable");
...
CifString cs("Target"), col1("column1");
CifString cs2("Target2"), col2("column2");
int errCode = 0;

ReVarPCifArray<int> * iArray = SSTable::SetUnion(s.SearchColumn(cs,
  col1, errCode), s.SearchColumn(cs2, col2, errCode));
```

PURPOSE

*SetUnion(ReVarPCifArray *, ReVarPCifArray *)* is a static method that returns the logical union of two sets of integers.
*SetUnion(ReVarPCifArray **, int)* is a static method that returns the logical union of any number of sets of integers.

RECEIVES

SetUnion(ReVarCifArray<int> *, ReVarCifArray<int> *)

`a`	The first set to take the union with.
`b`	The second set to take the union with.

SetUnion(ReVarCifArray<int> **, int)

`sets`	An array of pointers to sets of integers.
`num`	The number of elements in the given array.

RETURN VALUE

ReVarPCifArray<int> *, a pointer to a set representing the union of all sets passed to it.

REMARKS

These methods are useful for performing multiple column searches with logical OR operations.

3.0.3 GetErrorMessage

NAME

GetErrorMessage

PROTOTYPE

```
#include "SSTable.h"

static const char * SSTable::GetErrorMessage(const int errCode);
```

EXAMPLE

```
#include "SSTable.h"
#include "TableError.h"

SSTable s("MyTable");

CifString cs("Target"), col1("column1");
int errCode = 0;

errCode = s.AddColumn(col1);
if (errCode < 0) {
  log_error(SSTable::GetErrorMessage(errCode));
  if (errCode < SSTable::TABLE_WARNING) {
    log_error("FATAL ERROR");
    exit(errCode);
  }
}
```

PURPOSE

GetErrorMessage is a static method that returns the string representation of the error given by an error code. The error codes are located in TableError.h.

RECEIVES

`errCode`	The numerical code for the error to retrieve a string representation of.

RETURN VALUE

const char *, a string representing the error.

REMARKS

None