

cat2Sql

Richard J. Mathar

Generated by Doxygen 1.8.2

Mon Oct 15 2012 11:12:32

Contents

1	Main Page	1
2	Todo List	1
3	Bug List	1
4	Module Index	1
4.1	Modules	1
5	Data Structure Index	2
5.1	Data Structures	2
6	File Index	2
6.1	File List	2
7	Module Documentation	2
7.1	ASCII table to FITS/SQL converter	2
7.1.1	NAME	2
7.1.2	ENVIRONMENT	2
7.1.3	INSTALLATION	2
7.1.4	EXAMPLES	3
7.1.5	NOTES	4
7.1.6	MAKEFILE	4
7.1.7	ALSO	4
8	Data Structure Documentation	4
8.1	altCfg Class Reference	4
8.1.1	Detailed Description	5
8.1.2	Constructor & Destructor Documentation	5
8.1.3	Member Function Documentation	6
8.1.4	Field Documentation	8
8.2	Cfg Class Reference	8
8.2.1	Detailed Description	8
8.2.2	Constructor & Destructor Documentation	8
8.2.3	Member Function Documentation	10
8.2.4	Field Documentation	13
8.3	Col Class Reference	13
8.3.1	Detailed Description	14
8.3.2	Constructor & Destructor Documentation	14
8.3.3	Member Function Documentation	15
8.3.4	Field Documentation	18

9 File Documentation	19
9.1 cat2Sql.cxx File Reference	19
9.1.1 Macro Definition Documentation	19
9.1.2 Function Documentation	20
9.1.3 SYNOPSIS	20
9.1.4 DESCRIPTION	20
9.1.5 OPTIONS	20
9.1.6 FILES	21

Index

26

1 Main Page

2 Todo List

Group cat2Sql

Handle P and G and maybe H Fortran specifiers.

Are there complex data formats in SQL?

We may need masking escape sequences for some input lines.

Global Cfg::awkPrintf () const

the implementation duplicates [Cfg::awkPrintf\(\)](#) which ought be merged into an interface in the Cols class to avoid this.

Global Col::fitst () const

figure out whether this needs to be adorned with the repetition factor for vector columns.

3 Bug List

Group cat2Sql

If the standard input contains special characters like parenthesis or quotation marks, the generated output will likely be confusing the SQL interpreter, because no masking of these has yet been added to this program.

Global Cfg::Cfg (ifstream &inp, const char *baseFname, const string sqlkey)

the expansion of the vector columns needs to be suppressed if the output is of FITS type. So it must be moved to the place where the interface is an output to SQL.

4 Module Index

4.1 Modules

Here is a list of all modules:

ASCII table to FITS/SQL converter

2

5 Data Structure Index

5.1 Data Structures

Here are the data structures with brief descriptions:

altCfg	4
Cfg	8
Col	13

6 File Index

6.1 File List

Here is a list of all files with brief descriptions:

cat2Sql.cxx	19
-----------------------------	----

7 Module Documentation

7.1 ASCII table to FITS/SQL converter

Since

2006-05-24

Author

Richard J. Mathar

7.1.1 NAME

`cat2Sql` converts an ASCII catalog file into a SQL feed sequence file or a **FITS** file with a binary table.

7.1.2 ENVIRONMENT

No environment variables are used.

7.1.3 INSTALLATION

Under unix, a command

```
unix> make cat2Sql
```

is sufficient to compile the C++ source code into the executable. It may make sense to be more specific with the optimization and use s.th. like

```
unix> c++ -O -o cat2Sql cat2Sql.cxx
```

If the optional output into FITS binary tables is needed, the **cfitsio** and **CCfits** packages need to be available, and the header files and libraries be specified to the compiler and linker:

```
unix> c++ -O -DHAVE_CCFITS -o cat2Sql -I<include> cat2Sql.cxx -L<library> -lcfitsio -lCCFits
```

7.1.4 EXAMPLES

1. The first lines of the configuration file for the **WDS** catalog could look as follows:

```
A10 den bytes 1 to 10
A7 disc bytes 11 to 17
A5 comp
I4 datefst first year
I4 datelst last year
4X nobs skip the number of observations
I3 posfst first position
# teaser
I3 poslst last position columns 43-45
F5.1 sepfst [mas] separation first
```

If this is in a file wds.cfg, the SQL commands are created with

```
cat wdsnew*.txt | cat2Sql -c wds.cfg > wds.sql
```

supposed that the ASCII files had been in wdsnew*.txt .

2. the first lines of the "alternative" configuration file of the **CHARM2** catalog could look as follows

```
1-4 I N [] sequential order in the table
6-22 A Source [] Name of the source
24 A VLTI [] Presence of VLTI observations
26-29 A Type [] Type of source
31-54 A Id1 [] cross-indentification
55-78 A Id2 [] cross-indentification
80-103 A Id3 [] cross-indentification
105-129 A Id4 [] cross-indentification
131-135 A Method [] Method of observation
137-142 A Lambda [] Wavelength of observation
144-149 F UD [mas] uniform disk angular diameter
151-155 F e_UD [mas] mean error on UD
157-161 F5.2 ULUD [mas] Upper limit for Uniform disk diameter
163-168 F6.2 LD [mas] Limb-darkened angular diameter
170-173 F4.2 e_LD [mas] Mean error on LD
175 A1 l_Shell Limit flag on Shell
176-181 F6.1 Shell [mas] Circumstellar shell
182-185 A4 n_Shell Note on Shell (Y, Y? or 20x3)
187-192 F6.1 Sep [mas] Binary: separation
196-200 F5.1 PA [deg] Binary: position angle
202-206 F5.1 Ratio Binary: Brightness ratio
208 I1 Ratio2 ? Upper limit of brightness ratio
209-210 A2 n_Ratio2 [2m]
212-213 A2 Bin Binary: type
215-331 A117 Com Comments extracted from the reference or inserted by
the catalogue compiler
333-334 I2 RAh [h] Right Ascension J2000
336-337 I2 RAM [min] Right Ascension J2000
339-344 F6.3 RAs [s] Right Ascension J2000
346 A1 DE- Declination sign (J2000)
347-348 I2 DED [deg] Declination (J2000)
350-351 I2 DEM [arcmin] Declination (J2000)
353-357 F5.2 DES [arcsec] Declination (J2000)
359-366 F8.5 pmRA [as/a] Proper motion in right ascension
368-376 F9.6 pmDE [as/a] Proper motion in declination
378-382 F5.2 Bmag [mag] B magnitude
384-388 F5.2 Vmag [mag] V magnitude
390-394 F5.2 Kmag [mag] K magnitude (.2.2um)
396 A1 f_Kmag *Code for Kmag
398-404 F7.2 12um [Jy] Flux at 12um
406 A1 f_12um Code for 12um flux
408-420 A13 SpType MK Spectral class
421-424 A4 Var Variability note
426-431 F6.2 Plx [mas] Parallax
433-438 F6.1 km/s RV [km/s] Radial velocity
440-448 A9 ref1 References, in refs.dat file
450-458 A9 ref2 References, in refs.dat file
460-468 A9 ref3 References, in refs.dat file
```

If this is in the file charm2.cfg, it would be used like

```
unix> cat2sql -C charm2.cfg > tmp.cfg
unix> cat2sql -f -k N -c tmp.cfg < charm2.dat > charm2.fits
```

7.1.5 NOTES

The byte counts implied by the Fortran-specifiers in the configuration file are indeed byte counts. Tabulators, backspaces and other control characters should be expanded before feeding them into the program, see `expand(1)`. If the original input data have a free-format, one may also pipe it through a corresponding `awk(1)` with internal print commands to generate a fixed format on the fly, which can become the standard input to this program here. Example: If the original data are a name, an integer and a floating-point number, the pipe might look like

```
unix> cat origfile | awk '{print ".3s%4d%7f%10e",$1,$2,$3,$4}' | ...
```

to cast this input into a fixed format with a total range of $3+4+7+10$ bytes. The last lines of an output generated with the `-C` option show this type of command string for each particular configuration file.

The individual lines of the input table must be terminated by the standard character of the C++ implementation, which is the LF on UNIX systems. If the source table does not follow this guideline, one must use other tools like `cut(1)`, `wrap(1)`, `fold(1)` etc to ensure that the input lines can be interpreted starting with the first byte in each line according to the sequence of bytes of the configuration file.

Todo Handle P and G and maybe H Fortran specifiers.

Todo Are there complex data formats in SQL?

Todo We may need masking escape sequences for some input lines.

Bug If the standard input contains special characters like parenthesis or quotation marks, the generated output will likely be confusing the SQL interpreter, because no masking of these has yet been added to this program.

7.1.6 MAKEFILE

7.1.7 ALSO

`wds2Sql.pl`

8 Data Structure Documentation

8.1 altCfg Class Reference

Public Member Functions

- `altCfg (ifstream &inp)`
- `void sortnFill ()`
- `string awkPrintf () const`

Data Fields

- `vector< Col > cols`

Private Member Functions

- `int bStrt (const string bspec, int &bwidth) const`

8.1.1 Detailed Description

A class with the simpler configuration by specification of byte ranges. The simplification relative to the [Cfg](#) class is

- there is no need to characterize each byte in the input file, which means one may leave holes in the byte ranges which will effectively be skipped. These would have to be counted explicitly and used with `X` in the [Cfg](#) class.
- The order of the column specifications is arbitrary.
- Most astronomic data tables have documentation that specifies byte ranges starting with 1 at the start of the lines. This is equivalent to what is the input to this class here.

Since

2007-02-28

8.1.2 Constructor & Destructor Documentation

8.1.2.1 altCfg::altCfg (`ifstream & inp`) [inline]

ctor.

Parameters

<code>in</code>	<code>inp</code>	the file stream with the alternative input configuration
-----------------	------------------	--

```
{
    /* We loop over all lines of the configuration file
     */
    while ( !inp.eof() )
    {
        string buf ;
        getline(inp,buf) ;

        /* skip comment and empty lines
         */
        if ( buf[0] == '#' )
            continue ;
        if ( buf.empty() )
            continue ;

        istringstream line(buf) ;
        string bspec, f77fmt, dbcolname, commen ;

        /* each lines has a byte range specifier in the \c start
         * or \c start-end format, a Fortran specifier (where the width
         * becomes
         * redundant because it is implicit in the byte range), a name of
         * the column
         * for use in the SQL table or binary table, and a residual part
         * treated
         * as a comment (and optionally containing physical units)
         */
        line >> bspec >> f77fmt >> dbcolname ;

        /* push rest of the line into the comment */
        getline(line,commen,'\'0') ;

        int bwidth ;
        const int cstart = bStrt(bspec,bwidth) ;

        // f77fmt may contain a blank character, a character with a
        repetition,
        // and a character with a field width and/or precision
        string::size_type f77pos= f77fmt.find_first_of("APGXDEIF") ;

        /* discard invalid type specifications */
        if ( f77pos == string::npos )
        {
            cerr << "# invalid type spec " << f77fmt << " in " << buf <<
            endl ;
            continue ;
        }
        if ( f77fmt.at(f77pos) != 'X' )
        {
```

```

        /* we overwrite whatever may follow the character by the field
width from the first
         * part of the spec
         * @todo this ought actually honor any more detailed spec from
the
         * input line.
*/
f77fmt.erase(f77pos+1) ;
char bwidthstr[4];
snprintf(bwidthstr,4,"%d",bwidth) ;
f77fmt.append(bwidthstr) ;

/* debugging
 * cerr << "#" << cstart << " " << f77fmt << " " << dbcolname
<< " " << commen << endl ;
 */

/* For each line we create one instance of Col and add it to
the
 * vector cols .
*/
Col field(cstart,f77fmt,dbcolname,commen) ;
cols.push_back(field) ;
}
}

}

```

8.1.3 Member Function Documentation

8.1.3.1 string altCfg::awkPrintf() const [inline]

Construct a representation in the printf style of awk.

Returns

a string in the awk(1) style, like "%4d%7f%10s",\$1,\$2,\$3

Since

2007-03-07

```

{
    ostringstream fmt ;
    /* first construct the format specifiers, calling Col::awkPrintf() in a
     * loop over all columns in the table.
    */
    fmt << "\"";
    for(int c=0 ; c < cols.size() ; c++)
        fmt << cols[c].awkPrintf() ;
    fmt << "\"";

    /* append the list of the table columns.
    */
    int field=1 ;
    for(int c=0 ; c < cols.size() ; c++)
        if ( cols[c].ftype != -1 )
            fmt << ",$" << field++ ;

    return fmt.str() ;
}

```

8.1.3.2 int altCfg::bStrt(const string bspec, int & bwidth) const [inline], [private]

decompose the start- string into the start byte and the byte widths.

Parameters

in	<i>bspec</i>	a single number (meaning one byte only) or a range of bytes separated by a dash.
out	<i>bwidth</i>	the width of the column in units of bytes.

Returns

the byte number of the start (0-based)

Since

2007-02-28

```
{
    istringstream line(bspec) ;
    int resul[2] ;

    /* the first number denotes the first byte */
    line >> resul[0] ;

    /* We search for the dash. If it is present we skip it
     * and put the second number in resul[1].
     */
    string::size_type dashpos= bspec.find_first_of("-") ;
    if ( dashpos != string::npos)
    {
        char dash ;
        line >> dash >> resul[1] ;
    }
    else
        resul[1] = resul[0] ;

    /* The input was 1-based. on output we convert it to 0-based indexing
     */
    bwidth = resul[1]-resul[0]+1 ;
    return resul[0]-1 ;
}
```

8.1.3.3 void altCfg::sortnFill() [inline]

Sort the columns from left to right according to the byte placement in the input files. The main action is to insert dummy X columns to render this format compatible with the main configuration file, such that coverage of an entire line is ensured.

Since

2007-02-28

place the columns in increasing order so we can detect gaps by scanning them linearly

fill in X specifiers if there are gaps; these are appended instead of merged into the vector, because insertion would garble the iterators

place the columns in increasing order

```
{
    sort(cols.begin(),cols.end()) ;

    int currCol =0 ;

    /* The cols.size() will change while looping, so we save it beforehand
     */
    const int tablec = cols.size() ;
    for(int c=0 ; c < tablec ; c++)
    {
        if ( cols[c].col[0] > currCol)
        {
            ostringstream f77fmt, cmt ;
            f77fmt << cols[c].col[0]-currCol << "X" ;
            cmt << "dummy" << currCol << " auto-generated by "__FILE__" ;
            Col xcol(currCol, f77fmt.str() , cmt.str()) ;
            cols.push_back(xcol) ;
        }
        currCol = cols[c].col[0]+cols[c].repeat*(cols[c].col[1]
        -cols[c].col[0]) ;
    }

    sort(cols.begin(),cols.end()) ;
}
```

8.1.4 Field Documentation

8.1.4.1 `vector<Col> altCfg::cols`

the specifications of the columns in the configuration

8.2 Cfg Class Reference

Public Member Functions

- `Cfg` (`ifstream &inp, const char *baseFname, const string sqlkey)`
- `string awkPrintf () const`
- `string creatTbl () const`
- `string creatDoc (int argc, char *argv[]) const`
- `string creatInsrt (const string ascii, const bool warn) const`
- `void uniq ()`
- `void keyExist ()`

Data Fields

- `vector< Col > cols`
- `string tblname`

Private Member Functions

- `int bWidth () const`

Private Attributes

- `bool autopkey`
- `string pkey`

8.2.1 Detailed Description

This class represents a range of columns as embodied by a configuration file.

Since

2006-05-24

Author

Richard J. Mathar

8.2.2 Constructor & Destructor Documentation

8.2.2.1 `Cfg::Cfg (ifstream & inp, const char * baseFname, const string sqlkey) [inline]`

Constructor.

Parameters

<code>in</code>	<code>inp</code>	the file stream with the configuration file, positioned at the first line.
<code>in</code>	<code>baseFname</code>	the name of the table to be created.
<code>in</code>	<code>sqlkey</code>	the name of the column that may serve as a unique key in the SQL sense (such that two different rows in the table do never have the same entry in this column).

Bug the expansion of the vector columns needs to be suppressed if the output is of FITS type. So it must be moved to the place where the interface is an output to SQL.

Since

2006-05-24

Author

Richard J. Mathar

```

    : pkey(
sqlkey)
{
    /* Construct the SQL table name from the basename without the suffix.
     * Supposed that this is a default chosen by the program, there would
     * probably a \c .cfg be part of the string, that we discard.
    */
    tablename = basename ;
    string::size_type dot= tablename.find('.') ;
    if ( dot != string::npos)
        tablename.erase(dot) ;

    while ( !inp.eof() )
    {
        /* one line of the configuration file is in buf
        */
        string buf ;
        getline(inp,buf) ;

        /* skip comment lines or lines that are empty.
        */
        if ( buf[0] == '#')
            continue ;
        if ( buf.empty() )
            continue ;

        /* debugging
        /*cout << buf << endl ;
        */

        istringstream line(buf) ;
        string f77fmt, dbcolname, commen ;

        /* from the configuration line we read the Fortran type specifier,
         * the column name, and push the rest of the line (potentially
including
         * the units) into the comment
        */
        line >> f77fmt >> dbcolname ;

        /* debugging
        /* cout << f77fmt << " " << dbcolname << endl ;
        */
        getline(line,commen,'\'0') ;

        /* We create one instance of Col from this line of
         * the configuration file.
        */
        Col field(bWidth(),f77fmt,dbcolname,commen) ;
        if ( field.repeat > 1 )
        {
            /* If this is a vector field, we create multiple copies such
             * that each of the columns is represented.
            * @bug this is an incorrect place to do this since we would
             * like to create FITS vector columns without such duplication.
            */
            for(int r=1 ; r <= field.repeat ; r++)
            {
                Col field_rep(bWidth(),f77fmt,dbcolname) ;
                field_rep.cloneName(r) ;
                cols.push_back(field_rep) ;
                /* debugging
                /* cerr << " col " << field_rep.dbcolname << " "
                /* << field_rep.col[0] << "... " << field_rep.col[1] <<
                endl ;
                /*
            }
        }
        else
            cols.push_back(field) ;
    }
}

```

```

}

/* If the user didn't specify a SQL key, generate our own one
 * with the default name, leaving room for integers with up to 6 decimal
places,
 * which ought be large enough for most applications..
*/
if ( pkey.length() == 0 )
{
    pkey= SQL_UNIQ_KEY_DEFAULT ;
    Col field(-1,"I6",pkey) ;
    cols.push_back(field) ;
    autopkey=true ;
}
else
    autopkey=false ;
}

```

8.2.3 Member Function Documentation

8.2.3.1 string Cfg::awkPrintf() const [inline]

Construct a representation in the printf style of awk.

Returns

a string in the awk(1) style, like "%4d%7f%10s",\$1,\$2,\$3

Since

2007-03-07

Todo the implementation duplicates [Cfg::awkPrintf\(\)](#) which ought be merged into an interface in the Cols class to avoid this.

```

{
    ostringstream fmt ;
    /* first construct the format specifiers, calling Col::awkPrintf() in a
     * loop over all columns in the table.
    */
    fmt << "\\" ;
    for(int c=0 ; c < cols.size() ; c++)
        fmt << cols[c].awkPrintf() ;
    fmt << "\\" ;

    /* append the list of the table columns.
    */
    int field=1 ;
    for(int c=0 ; c < cols.size() ; c++)
        if ( cols[c].ftype != -1 )
            fmt << ","$ << field++ ;

    return fmt.str() ;
}

```

8.2.3.2 int Cfg::bWidth() const [inline], [private]

compute the total ASCII width (so far), which is the lowest (0-based) byte not yet recognized by the input parser

Returns

the rightmost byte number (0-based) which is not yet covered by any of the columns. This is the potentially safe place to append a new column.

Since

2006-05-24

```
{
    int resul=0 ;
    for(int i=0; i < cols.size() ; i++)
        resul=max(resul,cols[i].col[1]) ;
    return resul ;
}
```

8.2.3.3 string Cfg::creatDoc (int argc, char * argv[]) const [inline]

construct a 2-line string with the "created" information in SQL comment format.

Returns

a SQL comment that is a useful tag for the SQL command line.

Since

2006-05-24

```
{
    string resul= "-- created " ;
    time_t now ;
    time(&now) ;
    resul += ctime(&now) ;
    resul += "-- by " ;
    resul += getenv("USER") ;
    resul += " running " ;
    resul += argv[0] ;
    resul += " (author Richard J. Mathar) with options " ;
    for(int i=1 ; i < argc ; i++)
    {
        resul += argv[i] ;
        resul += " " ;
    }
    return resul ;
}
```

8.2.3.4 string Cfg::creatInsrt (const string ascii, const bool warn) const [inline]

Construct the INSERT sql command for the ascii line provided.

Parameters

in	ascii	the ASCII line from the input
in	warn	a flag to indicate if warnings about incompatibility with the format of the configuration should be emitted.

Returns

the INSERT SQL command equivalent to the parsed ascii line.

Since

2006-05-24

```
{
    static int recordNo =0 ;
    string resul="INSERT INTO " ;
    resul.append(tblname) ;
    resul += " (" ;
    bool firsten = true ;
    for(int i=0 ; i < cols.size() ; i++)
    {
```

```

        if( cols[i].sqlfmt.empty() )      // skip X formats..
            continue ;
        if( !firsten)
            resul += ", " ;
        resul += "\"" + cols[i].dbcolname + "\" " ;
        firsten=false ;
    }
resul += ") VALUES(" ;
firsten = true ;

/* the natural number of columns may or may not include the last column
 * which we may have generated for the auto-generated primary key.
 */
const int natcolNo = autopkey ? ( cols.size()-1 ) : cols
.size() ;

// handle all but the primary key (last of the columns)
for(int i=0 ; i < natcolNo ; i++)
{
    if( cols[i].sqlfmt.empty() )      // skip X formats..
        continue ;
    if( !firsten)
        resul += ", " ;

    if( ! cols[i].numFormat )
        resul += "' " ;
    resul += cols[i].cut(ascii,warn) ;
    if( ! cols[i].numFormat )
        resul += "' " ;
    firsten=false ;
}

if( autopkey)
{
    // handle the primary key
    resul += ", " ;
    ostringstream auxFmt ;
    auxFmt << ++recordNo ;
    resul += auxFmt.str() ;
}

resul += ")" ;
return resul ;
}

```

8.2.3.5 string Cfg::creatTbl() const [inline]

construct the CREATE TABLE sql command.

Since

2006-05-24

Returns

the string with the CREATE TABLE SQL command, covering the full table width..

```

{
    string resul="CREATE TABLE " ;
    resul.append(tblname) ;
    resul += "(" ;
    bool firsten = true ;

    /* this is essentially a loop across all columns, concatenated in
     * the syntax required for the \c CREATE \c TABLE
     */
    for(int i=0 ; i < cols.size() ; i++)
    {
        if( cols[i].sqlfmt.empty() )      // skip X formats..
            continue ;
        if( !firsten)
            resul += ", " ;
        resul += cols[i].dbcolname + " " + cols[i].sqlfmt ;
        firsten=false ;
    }
    resul += ", PRIMARY KEY (" + pkey + ")" ;
    resul += ")" ; // no semicolon appended
    return resul ;
}

```

8.2.3.6 void Cfg::keyExist() [inline]

A check that the primary key actually exists as a column If the primary key does not match an existing column name, we write a warning to `stderr` on this fact.

Since

2006-05-24

```
{
    for(int i=0; i < cols.size() ; i++)
        if( cols[i].dbcolname == pkey)
            return ;
    cerr << "Warning: primary key column " << pkey << " does not exist
    \n" ;
}
```

8.2.3.7 void Cfg::uniq() [inline]

A cross check that all SQL column names are mutually different If some of the column names are the same, we write a warning to `stderr`.

Since

2006-05-24

```
{
    for(int i=0; i < cols.size()-1 ; i++)
        for(int j=i+1; j < cols.size() ; j++)
            if( cols[i].dbcolname == cols[j].dbcolname)
                cerr << "Warning: columns " << i << " and " << j << " share
                the same name \""
                << cols[i].dbcolname << "\n" ;
}
```

8.2.4 Field Documentation**8.2.4.1 bool Cfg::autopkey [private]**

boolean flag indicating whether we created the unique primary SQL key locally or whether it was taken from a user specification.

8.2.4.2 vector<Col> Cfg::cols

The non-overlapping columns present in each row (each line of the ASCII input).

8.2.4.3 string Cfg::pkey [private]

the primary SQL key

8.2.4.4 string Cfg::tblname

The name of the FITS binary table or the SQL table.

8.3 Col Class Reference**Public Member Functions**

- [Col](#) (int colstrt, const string f77fmt, const string name, const string comment="")
- [void cloneName](#) (int r)
- [string cut](#) (const string ascii, const bool warn) const
- [string awkPrintf](#) () const

- string `up_name () const`
- string `unit () const`
- string `fitst () const`

Data Fields

- int `col [2]`
- int `repeat`
- string `sqlfmt`
- int `ftype`
- string `dbcname`
- string `comm`
- bool `numFormat`

8.3.1 Detailed Description

The `Col` class contains types and ranges (in units of bytes) in an ASCII table of one coherent datum, that is one column in the ASCII input file.

Since

2006-05-24

Author

Richard J. Mathar

8.3.2 Constructor & Destructor Documentation**8.3.2.1 `Col::Col (int colstrt, const string f77fmt, const string name, const string comment = " ") [inline]`**

Constructor.

Parameters

in	<code>colstrt</code>	the byte number (0-based) meaning the first (leftmost) byte in each of the ASCII input lines.
in	<code>f77fmt</code>	a width and type specifier in the Fortran convention. Examples are <code>3I6</code> for 3 columns each 6 bytes wide with integers, or <code>10A</code> strings of 10 bytes widths, or <code>F5.4</code> a single precision floating point number spanning over 5 bytes. The numeric formats (<code>I</code> , <code>F</code> , <code>E</code>) may be blank-padded on the left.
in	<code>name</code>	the column name.
in	<code>comment</code>	a comment on the meaning of the column. This would probably start with a unit enclosed in brackets.

Since

2006-05-24

Author

Richard J. Mathar

```
: dbcname(name), repeat(1), comm(comment), ftype
(-1)
{
    col[0]=colstrt ;
```

```

int wid =0 ;
int prec =0 ;
int r =0 ;
ostringstream auxFmt ;

// any repeat factor? Ignore attempts to introduce negative ones.
sscanf(f77fmt.c_str(),"%d",&r) ;
if (r>1)
    repeat=r ;

string::size_type f77pos= f77fmt.find_first_of("APGXDEIF") ;
switch( f77fmt[f77pos] )
{
case 'X' :
    sscanf(f77fmt.c_str(),"%d",&wid) ;
    col[1] = col[0]+wid ;
    repeat = 1 ; // no repeat factor allowed with X
    numFormat = false ;
    break ;
case 'I' :
    sscanf(f77fmt.c_str()+f77pos+1,"%d",&wid) ;
    col[1] = col[0]+wid ;
    sqlfmt="INTEGER";
    numFormat = true ;
    ftype= TLONG ;
    break ;
case 'A' :
    sscanf(f77fmt.c_str()+f77pos+1,"%d",&wid) ;
    col[1] = col[0]+wid ;
    auxFmt << "char(" << wid << ")" ;
    sqlfmt=auxFmt.str() ;
    numFormat = false ;
    ftype= TSTRING ;
    break ;
case 'D' :
case 'E' :
case 'F' :
    sscanf(f77fmt.c_str()+f77pos+1,"%d.%d",&wid,&prec) ;
    col[1] = col[0]+wid ;
    auxFmt << "DEC(" << wid ;
    if ( prec > 0 )
        auxFmt << "," << prec ;
    auxFmt << ")" ;
    sqlfmt=auxFmt.str() ;
    numFormat = true ;
    break ;
default:
    cerr << "Unrecognized field specification " << f77fmt << endl ;
    exit(EXIT_FAILURE) ;
}

switch( f77fmt[f77pos] )
{
case 'D' :
case 'E' :
    ftype=TDOUBLE ;
    break ;
case 'F' :
    ftype=TFLOAT ;
    break ;
}
// cerr << " cols " << col[0] << " .. " << col[1] << " prec " << prec
<< " " << sqlfmt << endl ;
}

```

8.3.3 Member Function Documentation

8.3.3.1 string Col::awkPrintf() const [inline]

Construct a representation in the printf style of awk.

Returns

a string in the awk(1) style, like %4d, %7f or %10s

Since

2007-03-07

```
{
    char f[FLEN_VALUE+1] ;
    f[0]='\0' ;
    /* as in ffgbcll() of fitscore.c */
    switch(ftype)
    {
        case -1 :
            // sprintf(f,"%%%ds",col[1]-col[0]) ;
            for(int i=col[0] ; i < col[1] ; i++)
                strcat(f," ") ;

            break ;
        case TBYTE :
            sprintf(f,"%%c") ;
            break ;
        case TSTRING :
            sprintf(f,"%%%ds",col[1]-col[0]) ;
            break ;
        case TSHORT :
        case TLONG :
        case TLONGLONG :
            sprintf(f,"%%%dd",col[1]-col[0]) ;
            break ;
        case TFLOAT :
            sprintf(f,"%%%df",col[1]-col[0]) ;
            break ;
        case TDOUBLE :
            sprintf(f,"%%%de",col[1]-col[0]) ;
            break ;
        case TBIT :
        case TLOGICAL :
        case TCOMPLEX :
        case TDBLCOMPLEX :
        default:
            /* no warning here with this version, because the output will only
             * be used in a comment.
            */
    }
    return string(f) ;
}
```

8.3.3.2 void Col::cloneName(int r) [inline]

Clone a name by appending a number. We attach the number r to the base name of the column to have different column names in case this is used for SQL table construction.

Parameters

in	r	the sequential number of this vector column.
----	---	--

Since

2006-05-24

```
{
    ostringstream auxFmt ;
    auxFmt << dbcolname << r ;
    dbcolname = auxFmt.str() ;
}
```

8.3.3.3 string Col::cut(const string ascii, const bool warn) const [inline]

Chop off a number of bytes (vertically) in the ASCII string. This acts like the UNIX cut(1) operation for a consecutive byte range, and leaves only the bytes that belong to this column instance here.

Parameters

in	ascii	the ASCII line
----	-------	----------------

Returns

the substring that consists of the consecutive bytes from within this ASCII string as specified by this instances col parameter.

Since

2006-05-24

```
{
    if ( ascii.length() < col[1] )
    {
        if ( warn )
            cerr << "Warning: Input line " << ascii << " too short (" <<
        ascii.length()
            << " found," << col[1] << " needed)\n" ;
        return string() ;
    }
    else
        return string(ascii,col[0],col[1]-col[0]) ;
}
```

8.3.3.4 string Col::fitst() const [inline]

Create a binary table column type string

Since

2007-02-28

Returns

the representation like J or D or 10A as requested for cfitsio instantiation of this particular column.

Todo figure out whether this needs to be adorned with the repetition factor for vector columns.

```
{
    char f[FLEN_VALUE+1] ;
    /* as in ffgbcll() of fitscore.c */
    switch(ftype)
    {
    case TBIT :
        sprintf(f,"X") ; break ;
    case TBYTE :
        sprintf(f,"B") ; break ;
    case TLOGICAL :
        sprintf(f,"L") ; break ;
    case TSTRING :
        sprintf(f,"%dA",col[1]-col[0]) ; break ;
    case TSHORT :
        sprintf(f,"I") ; break ;
    case TLONG :
        sprintf(f,"J") ; break ;
    case TLONGLONG :
        sprintf(f,"K") ; break ;
    case TFLOAT :
        sprintf(f,"E") ; break ;
    case TDOUBLE :
        sprintf(f,"D") ; break ;
    case TCOMPLEX :
        sprintf(f,"C") ; break ;
    case TDBLCOMPLEX :
        sprintf(f,"M") ; break ;
    }
    return string(f) ;
}
```

8.3.3.5 string Col::unit() const [inline]

Extract a unit from the comment.

Returns

the substring between the first pair of [and], and an empty string if no such specification is in the comment.

Since

2007-02-28

```
{
    /* Search for the opening and the closing bracket */
    string::size_type op_bra = comm.find_first_of("[");
    string::size_type clo_bra = comm.find_first_of("]");

    /* If both are present and in the correct order, extract the
     * string in between, otherwise return an empty string.
     */
    if (op_bra != string::npos && clo_bra != string::npos && clo_bra >
        op_bra)
        return comm.substr(op_bra+1,clo_bra-op_bra-1);
    else
        return string("");
}
```

8.3.3.6 string Col::up_name() const [inline]

```
{
    char uname[FLEN_VALUE+1];
    uname[0]='\0';
    strncat(uname,dbcolname.c_str(),FLEN_VALUE);
    for(int c=0; c < strlen(uname); c++)
        uname[c]=toupper(uname[c]);
    return string(uname);
}
```

8.3.4 Field Documentation**8.3.4.1 int Col::col[2]**

The byte range of this column in the input ASCII file. This is 0-based such that `col[0]` = 0 would indicate the first byte in each line. `col[1]` contains the first byte after the column. If this is a column which is repeated, `col[1]` refers to the first instance of the columns, such that it actually coincides with the location of the first byte of the 2nd column in the ASCII file.

8.3.4.2 string Col::comm

Comment of the column in the FITS sense. This ought contain the physical unit of the data in the cfitsio convention, which means in a first piece embraced by [...].

8.3.4.3 string Col::dbcolname

Name of the column in the SQL database or the FITS binary table. If the output is a FITS binary table, all characters will be forced into uppercase.

8.3.4.4 int Col::ftype

The FITS column type in the cfitsio sense as defined in the `fitsio.h` file of its C source.

8.3.4.5 bool Col::numFormat

`true` if the column is of numerical type , `false` for character or name formats.

8.3.4.6 int Col::repeat

Repetition number for vector type columns. Default is 1 which means the next, adjacent column is specified by its own instance of this class. Resembles the `r` specifier in the formatted input/output of Fortran.

8.3.4.7 string Col::sqlfmt

the data type in the SQL sense. This can be INTEGER, char, double etc.

9 File Documentation

9.1 cat2Sql.hxx File Reference

Data Structures

- class [Col](#)
- class [Cfg](#)
- class [altCfg](#)

Macros

- `#define TBIT 1 /* codes for FITS table data types */`
- `#define TBYTE 11`
- `#define TSBYTE 12`
- `#define TLOGICAL 14`
- `#define TSTRING 16`
- `#define TUSHORT 20`
- `#define TSHORT 21`
- `#define TUINT 30`
- `#define TINT 31`
- `#define TULONG 40`
- `#define TLONG 41`
- `#define TINT32BIT 41 /* used when returning datatype of a column */`
- `#define TFLOAT 42`
- `#define TLONGLONG 81`
- `#define TDOUBLE 82`
- `#define TCOMPLEX 83`
- `#define TDBLCOMPLEX 163`
- `#define FLEN_VALUE 71`
- `#define SQL_UNIQ_KEY_DEFAULT "sql_key"`

Functions

- `ostream & operator<< (ostream &os, const Col &c)`
- `bool operator< (const Col &left, const Col &right)`
- `ostream & operator<< (ostream &os, const Cfg &cfg)`
- `ostream & operator<< (ostream &os, const altCfg &cfg)`
- `void usage (char *argv[])`
- `int main (int argc, char *argv[])`

9.1.1 Macro Definition Documentation

9.1.1.1 #define FLEN_VALUE 71

9.1.1.2 #define SQL_UNIQ_KEY_DEFAULT "sql.key"

This is the name of the SQL column with the unique key if no other one has been specified on the command line.

```
9.1.1.3 #define TBIT 1 /* codes for FITS table data types */

9.1.1.4 #define TBYTE 11

9.1.1.5 #define TCOMPLEX 83

9.1.1.6 #define TDBLCOMPLEX 163

9.1.1.7 #define TDOUBLE 82

9.1.1.8 #define TFLOAT 42

9.1.1.9 #define TINT 31

9.1.1.10 #define TINT32BIT 41 /* used when returning datatype of a column */

9.1.1.11 #define TLOGICAL 14

9.1.1.12 #define TLONG 41

9.1.1.13 #define TLONGLONG 81

9.1.1.14 #define TSBYTE 12

9.1.1.15 #define TSHORT 21

9.1.1.16 #define TSTRING 16

9.1.1.17 #define TUINT 30

9.1.1.18 #define TULONG 40

9.1.1.19 #define TUSHORT 20
```

9.1.2 Function Documentation

```
9.1.2.1 int main( int argc, char * argv[] )
```

9.1.3 SYNOPSIS

```
cat2Sql [-w] [-k keyname] [-a] [-n sqlname] -c fmt.cfg < inputcat.txt > out.sql
cat2Sql -f [-w] [-k keyname] [-a] [-n sqlname] -c fmt.cfg < inputcat.txt > out.
    fits
cat2Sql -C altfmt.cfg > fmt.cfg
```

9.1.4 DESCRIPTION

The program converts an ASCII table provided by the standard input into a sequence of SQL commands (which would be used with the sqlite3(1) command .read, for example, or a similar mysql(1)) that feed a SQL interpreter with the contents of this table.

Each line of the standard input becomes one line in the SQL table; the column layout of the standard input and the names of the columns in the SQL table must be detailed with the mandatory '-c' switch pointing to an ASCII configuration file with read-access rights.

9.1.5 OPTIONS

-a The program usually ignores lines of the standard input that start with the '#' character. This can be disabled by using the command line switch '-a' ("all") which triggers that even the lines that start with the sharp are considered to end up in the SQL database.

- c Names the configuration file that details the widths and names of the table columns of the standard input.
- C Indicates that a configuration file altfmt.cfg with an alternative format is converted into the standard format for later use with the option -c.
- f Specifies that the standard output should be in the FITS format. Only available if enabled at compile time.
- n Changes the SQL table name that is created from the default (which is the configuration file name after removal of any directory name and/or suffix) to a user-specific value.
- w Switches on a higher level of warnings on stderr. This current includes (i) Warnings if SQL column names are not unique. (ii) Warnings if the standard input has too short lines to fill the SQL table with all values. (iii) Warnings if a column selected with the -k switch doesn't exist.
- u Select one of the column names mentioned in the configuration file as the SQL key. The argument of the switch is the name of the column. If this column is specified with a repetition factor (see below as explained in the configuration file), the argument of the switch must include the explicit index number.

9.1.6 FILES

stdin The standard input must contain lines that are at least as long as demanded by the concatenation of the format specifiers of the configuration file (see below). Trailing columns/bytes will be ignored. The standard input must contain what would be called fixed format in the Fortran nomenclature, which means that the start and end of its columns is determined by the byte position within each row, not by separators (delimiters) like spaces, commas etc.

configuration file The configuration file contains

- empty lines which are ignored
- lines that start with the '#' and are ignored
- lines that contain (i) optional white space, (ii) a Fortran-type format/width specifier, The Fortran-type format specifiers are one of the following:
 - An (the capital letter A followed by a positive integer n) indicating an "ASCII" column of width 'n'
 - nX (a positive integer n followed by the capital letter X) indicating a number of 'n' Bytes to be skipped
 - In (the capital letter I followed by a positive integer n) indicating an (optionally signed) integer number with up to n bytes.
 - Fw.d
 - Ew.d
 - Dw.d (a letter F, E or D followed by a width number w, a dot and a decimal precision number d) indicating a floating point number. The width w includes any signs, decimal points, the letter 'e' etc. The A,F,E and D specifier can be preceded by positive integers to indicate field repetition. The SQL column names will be generated in this case by attaching the numbers 1, 2 and so on to the SQL column name that follows. The lines in the standard input must then provide a multiple of 'w' bytes to fill these repeated columns with information. (iii) white space, (iv) a SQL/FITS column name, (v) optionally white space and trailing contents. This is generally ignored but is used to generate the units and the comment in case of output in FITS format. The units follow the cfitsio convention that they are the first part of the comment, enclosed by brackets.

alternative configuration file The alternative configuration file can be translated into the standard format with a call using the -C option. It is a little bit more user-friendly and contains

- empty lines which are ignored
- lines that start with the '#' and are ignored
- lines that contain (i) optional white space, (ii) a byte-range: a single integer with a 1-based byte number or two integers separated by a dash (but no additional white space) indicating a byte range. This represents the range of bytes in each input line of the ASCII table for one column in the final FITS or SQL table. (iii)

a Fortran-type format/width specifier, (iv) white space, (v) a column name, (vi) optionally white space and trailing contents (which is ignored). The Fortran-type format specifiers are one of the following:

- A (the capital letter A) indicating an "ASCII" column
- I (the capital letter I) indicating an (optionally signed) integer number.
- F
- E
- D (a letter F, E or D) indicating a floating point number. The A,F,E and D specifier can be preceded by positive integers to indicate field repetition. In these cases, the byte-range specified before represents only the first of these columns.

Since

2006-05-24

Author

Richard J. Mathar

scan the input file provided by the -C command line switch

in case there were undefined bytes in the input, insert X specifiers for these Make sure that the output has a monotonically increasing order because this type of information is implicit in the output file.

Generate the list of column specs

```
{
    /* command line switches */
    char oc ;

    /* The unix file name with a configuration file (both syntaxes) */
    string cfgfile ;

    /* name of the SQL or binary FITS table */
    string sqlTabname ;

    /* SQL unique key (its column name ) */
    string sqlkey ;

    /* flag to indicate that also configuration lines starting with hashes
     * are to be interpreted instead of being treated as comments
     */
    bool allCfg = false ;

    /* warning level
     */
    bool warn = false ;

    /* If true, generated output is a FITS file with a binary table,
     * else a sequence of SQL feeds.
     */
    bool useFits = false ;

    /* If true, only a conversion from the simple to the compact
     * syntax of the configuration is performed, else the standard input
     * is used to create a table in one of the two formats.
     */
    bool cfg2cfg = false ;

    /* trivial syntax check */
    if( argc < 3)
    {
        usage(argv) ;
        return 1 ;
    }

    /* potential command line options include the \c f switch if
     * this was successfully compiled with \c CCFITS support.
     */
#ifndef HAVE_CCFITS
    while ( (oc=getopt(argc,argv,"C:fc:an:wk:")) != -1 )
#else
    while ( (oc=getopt(argc,argv,"C:c:an:wk:")) != -1 )
#endif
    {
        switch(oc)
        {
        case 'c' :

```

```

        cfgfile = optarg ;
        break ;
    case 'C' :
        cfg2cfg = true ;
        cfgfile = optarg ;
        break ;
    case 'n' :
        sqlTabname = optarg ;
        break ;
    case 'k' :
        sqlkey = optarg ;
        break ;
    case 'a' :
        allCfg = true ;
        break ;
    case 'f' :
        useFits = true ;
        break ;
    case 'w' :
        warn = true ;
        break ;
    case '?' :
        cerr << "Invalid command line option " << char(optopt) << "
ignored...\n" ;
        break ;
    }
}

/* we try to open the input file with the configuration */
ifstream inp(cfgfile.c_str(),ios::in) ;
if ( !inp )
{
    /* If the configuration file of the -c or -C switch isn't readable,
     * return with an error.
     */
    cerr << "Cannot open " << cfgfile.c_str() << " for reading\n" ;
    return 1 ;
}
else if ( cfg2cfg )
{
    altCfg conf(inp) ;

    conf.sortnFill() ;

    cout << conf ;

    /* generate the stdout file with a time stamp, calling the output
operator
 */
    time_t now = time(0) ;
    cout << "# created from " << cfgfile << " by " << __FILE__ << " " <<
ctime(&now) ;
}
else
{
    // cerr << "reading " << cfgfile.c_str() << endl ;
    char ifbase[256] ;
    strncpy(ifbase,cfgfile.c_str(),255) ; ifbase[255]='\0' ;

    // overload table name with user request
    if ( sqlTabname.length() )
    {
        memset(ifbase,'\0',256) ;
        sqlTabname.copy(ifbase,255) ;
    }

    Cfg conf(inp, basename(ifbase), sqlkey) ;
    inp.close() ;

    /* if -f switch used (and support compiled in)
 */
    if ( useFits )
    {
#ifndef HAVE_CCFITS
        /* enforce use of upper-name column names
 */
        for(int c=0 ; c < conf.cols.size() ; c++)
            conf.cols[c].dbcolname = conf.cols[c].up_name() ;

        /* debugging
         * for(int c=0 ; c < conf.cols.size() ; c++)
         *   cerr << c << " " << conf.cols[c].dbcolname << endl ;
         */

        /* create the table header according to the configuration
 */
        FITS *f = conf ;

```

```

/* handle lines of the standard input, one by one
 */
while ( !cin.eof() )
{
    string buf ;
    getline(cin,buf) ;

    // skip comment lines unless the 'allCfg' flag had been set.
    if ( !allCfg && buf[0] == '#' )
        continue ;
    if ( buf.empty() )
        continue ;

    ExtHDU & hdu = f->extension(1) ;

    /* append this ascii input to the row. Since the CCfits
    interface
     * uses a 1-based row enumeration, we add 1 to the current row
    count.
     */
    conf.createInsrt(hdu,buf,hdu.rows()+1L, warn) ;
}
delete f ;
#endif
}
else
{
    /* If the warning flag has been activated, call Cfg.uniq() and
    Cfg.keyExist()
     * to generate warnings.
     */
    if( warn)
    {
        conf.uniq() ;
        conf.keyExist() ;
    }

    /* Create the SQL configuration commands
    */
    cout << conf.createDoc(argc, argv) << endl ;
    cout << conf.createTbl() << ";\n" ;

    /* loop over the \c stdin lines to generate one feed per line into
    the SQL table. */
    while ( !cin.eof() )
    {
        string buf ;
        getline(cin,buf) ;

        // skip comment lines unless the 'allCfg' flag had been set.
        if ( !allCfg && buf[0] == '#' )
            continue ;
        if ( buf.empty() )
            continue ;

        cout << conf.createInsrt(buf, warn) << ";\n" ;
    }
}

return 0 ;
}
}

```

9.1.6.1 bool operator< (const Col & left, const Col & right)

Comparison operator. A column is smaller than another one if its byte location is left from it in the ASCII input.

Parameters

in	<i>left</i>	column to the left of the comparison sign
in	<i>right</i>	column to the right of the comparison sign

Returns

true or false

Author

Richard J. Mathar

Since

2007-02-28

```
{
    return ( left.col[0] < right.col[0]) ;
}
```

9.1.6.2 ostream& operator<< (ostream & os, const Col & c)

```
{
    if ( c.repeat > 1)
        os << c.repeat ;
    switch( c.ftype)
    {
        case TSTRING:
            os << c.col[1]-c.col[0] << "A" ; break ;
        case TLONG:
            os << "I" << c.col[1]-c.col[0] ; break ;
        case TFLOAT:
            os << "F" << c.col[1]-c.col[0] ; break ;
        case TDOUBLE:
            os << "D" << c.col[1]-c.col[0] ; break ;
        case -1:
            os << c.col[1]-c.col[0] << "X" ; break ;
    }
    os << " " << c.dbcolname << " " << c.comm ;
    return os ;
}
```

9.1.6.3 ostream& operator<< (ostream & os, const Cfg & cfg)

Output of a configuration represented in [altCfg](#) style in the style of the configuration file [Cfg](#). This is done by looping over all the individual columns.

Parameters

in,out	<i>os</i>
in	<i>cfg</i>

Since

2007-03-07

```
{
    for(int c=0 ; c < cfg.cols.size() ; c++)
        os << cfg.cols[c] << endl ;
    return os ;
}
```

9.1.6.4 ostream& operator<< (ostream & os, const altCfg & cfg)

Output of a configuration represented in [altCfg](#) style in the style of the configuration file [Cfg](#). This is done by looping over all the individual columns. An aid of how ASCII files that use tabulator stops in [starbase style](#) can be formatted with the UNIX [awk](#) is added as a comment at the end.

Parameters

in,out	<i>os</i>
in	<i>cfg</i>

Since

2007-02-28

```
{  
    for(int c=0 ; c < cfg.cols.size() ; c++)  
        os << cfg.cols[c] << endl ;  
  
    os << "# Hint: conversion of rgb tab-separated input files into blank  
    separated:\n" ;  
    os << "# awk -F '\\t' '{ printf " << cfg.awkPrintf() << "}' <  
    rgb.txt > ascii.txt\n" ;  
    return os ;  
}
```

9.1.6.5 void usage (char * argv[])

Print a syntax note to stderr.

Parameters

in	argv	the UNIX command line arguments, including the command name (executable)
----	------	--

```
{  
    cerr << "usage: " << argv[0] << " [-f] [-w] [-k keyname] [-a] [-n  
    sqlTblname] -c fmt.cfg < inputcat.txt > outfile.sql " << endl ;  
    cerr << "usage: " << argv[0] << " -C altfmt.cfg < simpleAltFmt.cfg >  
    fmt.cfg " << endl ;  
}
```

Index

ASCII table to FITS/SQL converter, 2
altCfg, 4
 altCfg, 5
 altCfg, 5
 awkPrintf, 6
 bStrt, 6
 cols, 7
 sortnFill, 7
autopkey
 Cfg, 13
awkPrintf
 altCfg, 6
 Cfg, 10
 Col, 15

bStrt
 altCfg, 6
bWidth
 Cfg, 10

cat2Sql.cxx, 19
 FLEN_VALUE, 19
 main, 20
 operator<, 24
 operator<<, 25
 TBIT, 19
 TBYTE, 20
 TCOMPLEX, 20
 TDBLCOMPLEX, 20
 TDOUBLE, 20
 TFLOAT, 20
 TINT, 20
 TINT32BIT, 20
 TLOGICAL, 20
 TLONG, 20
 TLONGLONG, 20
 TSBYTE, 20
 TSHORT, 20
 TSTRING, 20
 TUINT, 20
 TULONG, 20
 TUSHORT, 20
 usage, 26
Cfg, 7
 autopkey, 13
 awkPrintf, 10
 bWidth, 10
 Cfg, 8
 cols, 13
 creatDoc, 10
 creatInsrt, 11
 creatTbl, 12
 keyExist, 12
 pkey, 13
 tblname, 13
 uniq, 13

cloneName
 Col, 16
Col, 13
 awkPrintf, 15
 cloneName, 16
 Col, 14
 col, 18
 comm, 18
 cut, 16
 dbcasename, 18
 fitst, 17
 ftype, 18
 numFormat, 18
 repeat, 18
 sqlfmt, 18
 unit, 17
 up_name, 18
col
 Col, 18
cols
 altCfg, 7
 Cfg, 13
comm
 Col, 18
creatDoc
 Cfg, 10
creatInsrt
 Cfg, 11
creatTbl
 Cfg, 12
cut
 Col, 16
dbcasename
 Col, 18
FLEN_VALUE
 cat2Sql.cxx, 19
fitst
 Col, 17
ftype
 Col, 18
keyExist
 Cfg, 12
main
 cat2Sql.cxx, 20
numFormat
 Col, 18
operator<
 cat2Sql.cxx, 24
operator<<
 cat2Sql.cxx, 25

pkey
 Cfg, 13

repeat
 Col, 18

sortnFill
 altCfg, 7

sqlfnt
 Col, 18

TBIT
 cat2Sql.cxx, 19

TBYTE
 cat2Sql.cxx, 20

TCOMPLEX
 cat2Sql.cxx, 20

TDBLCOMPLEX
 cat2Sql.cxx, 20

TDOUBLE
 cat2Sql.cxx, 20

TFLOAT
 cat2Sql.cxx, 20

TINT
 cat2Sql.cxx, 20

TINT32BIT
 cat2Sql.cxx, 20

TLOGICAL
 cat2Sql.cxx, 20

TLONG
 cat2Sql.cxx, 20

TLONGLONG
 cat2Sql.cxx, 20

TSBYTE
 cat2Sql.cxx, 20

TSHORT
 cat2Sql.cxx, 20

TSTRING
 cat2Sql.cxx, 20

TUINT
 cat2Sql.cxx, 20

TULONG
 cat2Sql.cxx, 20

TUSHORT
 cat2Sql.cxx, 20

tblname
 Cfg, 13

uniq
 Cfg, 13

unit
 Col, 17

up_name
 Col, 18

usage
 cat2Sql.cxx, 26