# Note 161: Table Tutorial

Using the AIPS++ Table Data system
start of DRAFT Version

**F.M. Olnon**

Last modified: $Date: 2003/07/16 03:49:04 $ By: $Author: aips2adm $

# Introduction

**NOTE: This document is very old and does not reflect the current status of the AIPS++ Table System anymore**.

The AIPS++ Table Data System is described in the reference document with that title, written by G. van Diepen and A. Farris. This tutorial is a stylized report of my own first experiences with the tables, and as such is an introduction to the table system for application programmers.

The tutorial is built around the example programs from the appendix of the reference document. It starts with a simple picture of an AIPS++ table and adds more and more *finesse*. After finishing the tutorial you certainly will not have a complete understanding of AIPS++ tables — especially not if you want to derive your own table classes —, but you should be able to pick up the rest more easily from the reference document and ultimately from the header files and the code.

Let me start with the basic user model of an AIPS++ table:

- A table consists of a set of keywords and a set of columns.
- Each keyword and column has a unique name in the table context.
- Each keyword and column value can be a scalar, an array or a table.
- All basic data types, including string and complex, are supported.
- Keywords can be attached to individual table columns and keywords.

The terminology of the AIPS++ table system is somewhat confusing, at least it still is to me. I am used to look at tables as fundamentally two-dimensional assemblies of fields that contain values — or, one step further, 'pointers' to values or value assemblies. The meaning of the AIPS++ table classes `Field` and `TableValue` is quite different from that: `TableValue` objects are what I tend to call fields: the table cells that contain the values (or 'pointers'), and `Field` objects are whole columns or individual keywords. This comes from a more one-dimensional view of tables, which does right to the basic difference between the two table directions: a Table is a set of Fields, and there are two kinds of Fields, columns and keywords — in fact, they are now implemented as a single class `Field`.

The strong similarity between AIPS++ table columns and keywords is another feature that I found difficult to work with. It helped me a lot to think of a table as consisting of two subtables, a 'normal' table with columns and rows and a degenerate table with only one row (the set of keywords). Sets of keywords, attached to a table, a column or a keyword, are then single-row tables in there own right. It is then more obvious that keywords and columns are accessed in the same way, and that sets of keywords are handled as if they are rows.

# 1 Table Description

Usually, a table is built according to an existing table description, often referred to as the *type* of the table. This description defines all the keywords and columns in the table and the attributes associated with them.

When a table is created, it makes a copy of the description. From then on the table and the original description are decoupled: the original description can be changed without affecting tables created from it, and, conversely, a table description may be modified during the creation of a table (*e.g.*, when reading in FITS data) without affecting the original description.

Table descriptions live as objects of the class `TabDesc` with a certain name, but they can be stored in files with the same name (plus the suffix `.tabdesc`) and be revived from those files.

Here is the program with which I created my first table description called `Example` and stored it in file `Example.tabdesc`:

```
//-*- C++ -*-
// Build a table description (see Note 1).
//
#include <aips/TabDesc.h>
#include <aips/FieldDesc.h>

main()
{
//
// Create empty description "Example" (see Note 2).
//
    TabDesc td ("Example", TabDesc::New);
//
// Add some keywords (two string scalars and a 1-dim float array) to the
// description (see Note 3).
//
    td.addKey ("date", TpString);
    td.addKey ("name", TpString);
    FieldDesc* fdp1 = td.addKey ("freq", 1, TpFloat);
//
// Add some columns to the description (see Note 3).
//
    td.addCol ("U", TpDouble);
    td.addCol ("V", TpDouble);
    td.addCol ("time", TpFloat);
    td.addCol ("baseline", TpInt);
    td.addCol ("vis", 1, TpComplex);
//
// Add attributes to the last keyword (and one deeper) (see Note 4).
//
    fdp1->setComment ("Frequency array");
    FieldDesc* fdp2 = fdp1->addKey ("unit", TpString);
    fdp2->setDef ("MHz");
```

```
//
// List table description on standard output.
//
    td.show();
//
// On exit the object will be deleted, but the description will be
// stored in the file Example.tabdesc in the current working directory
// (see Note 2).
// }
```

Notes:

1. As a rule, every class used needs its own header file. So, if you use class `X`, you must include `X.h`. Anyway, the compiler will report any undefined classes.
2. With constructor option `New` a new description is created, and saved on destruction. This might overwrite an existing "Example" description; use option `NewNoReplace` when you do not want to run that risk. The options `Old` (default) and `Update` would recreate the "Example" description from the file Example.tabdesc and allow modification of the description. The difference between these two options is that a modified `Old` description will **not** be saved.
3. The functions `addKey` and `addCol` have many more forms than the two shown here, but they all return a pointer to the field description — remember that keywords and columns are just two kinds of fields — which can then be used to define additional attributes for that keyword or column (see next Note).
4. The optional comment attribute will be shown in the listing of the table. Default values can be defined for a scalar and for part or all of an array; the data type of the default should match the field data type. Keywords can be attached to table keywords or columns in much the same way as keywords are added to a table description. These keywords are treated in exactly the same way as table keywords and columns, so comment, default(s) and keywords can also be defined for them, *etc.*.
5. Although not demonstrated in the example, keywords and columns can not only be added to a table or field description; they can also be renamed and removed:

   ```
   td.renameKey ("title", "name");   //change table-keyword name to "title"
   td.removeKey ("date");            //remove keyword "date" from table
   fdp1->removeKey ("unit");         //remove keyword "unit" from field "freq"
   ```

   The functions `renameCol` and `removeCol` have the same signature.

# 2 Simple Table

Once the table description exists, we can create and fill a proper table according to that description. As mentioned before, we can freely modify the description of the table during the building-process, because the table keeps its own copy of the original description.

Like table descriptions, tables live as objects of the class `Table` with a certain name, but they can be stored in files with the same name (without suffix) and be revived from those files.

The following program builds table `TableExample` according to table description `Example`, and stores it in file `TableExample`:

```
//-*- C++ -*-
// Build a table.
//
#include <aips/Table.h>
#include <aips/Row.h>
#include <aips/TableValue.h>
#include <aips/Vector.h>
#include <aips/Field.h>

main()
{
//
// Create new table of type "Example" (see Note 1).
//
    Table tab ("TableExample", "Example", Table::New);
//
// Fill in values for the keywords (see Note 2).
//
    Row keyrow = tab.keywords();           //access keywords as a table row
    keyrow["date"].put ("24-12-93");       //index in row by name and put
    keyrow["name"].put ("Friso Olnon");
    TableValue fkey = keyrow["freq"];      //indexing gives a TableValue
//
// "freq" is a 1-dim array with as yet undefined length.
// That length must be defined before data can be put in.
//
    uInt nrfreq = 32;
    fkey.setDim (nrfreq);                  //set number of frequencies
    Vector<float> freqval(nrfreq);         //create a vector of type float
    for (int i=0; i<nrfreq; i++) {         //and fill it
        freqval(i) = i*100;
    }
    fkey.put (freqval);                    //put frequency values in
//
// We want "KHz" as unit for "freq" instead of the default "MHz".
// "unit" is a keyword attached to table keyword "freq" (see Note 3).
//
// The statement used is the same as:
```

```
//    Field* kfp = tab.getKey("freq");     //access table keyword "freq"
//    Row frow = kfp->keywords();          //access keywords of "freq"
//    TableValue ukey = frow["unit"];      //access keyword "unit" of "freq"
//    ukey.put("KHz");                     //and put value in
//
    tab.getKey("freq")->keywords()["unit"].put ("KHz");
//
// Now fill the table columns, row for row (see Note 4).
//
    uInt nrrow = 1000;                     //number of rows to create
    Row row;                               //declare a row object
    Int ucol   = tab.getColIndex("U");     //for efficiency: use indices
    Int vcol   = tab.getColIndex("V");
    Int timcol = tab.getColIndex("time");
    Int bascol = tab.getColIndex("baseline");
    Int viscol = tab.getColIndex("vis");
    for (i=1; i<nrrow; i++) {
        row = tab.addRow();                //add a row
        row[ucol].put(0.0);                //fill in per column
        row[vcol].put(1.0);
        row[timcol].put(2.0);
        row[bascol].put(3);
//      row[viscol].put(visvector);
    }
//
// On exit the object will be deleted and the table contents will be stored
// in the file TableExample in the current working directory (see Note 1).
//
}
```

Notes:

1. A table can be constructed with the same options as a table description. In addition to the already known `Old` (default), `Update`, `New` and `NewNoReplace` options, we can also use `Scratch` to create a temporary table(description), and `Delete` to delete an existing table(description).
2. The basic classes for accessing data in a table are:

   `Row` — The set of all column cells in a particular row of the table, but also the set of all keywords in the table (or the set of keywords attached to a column or a keyword, *etc.*). A `Row` object can be constructed by operations on the `Table` object. It gives access to the `TableValue` objects.

   `TableValue` — A single keyword (a cell in a row) or a particular cell in a column. A `TableValue` object can be constructed by indexing the row with the name or the index number of the keyword or column. It gives access to the contents of the keyword or column cell (a scalar, an array or a table).

   `Field` — A column (all cells together) or a keyword (there is only one cell). A pointer to `Field` can be obtained with the `Table` functions `getKey` and getCol or index operator [] with a name as argument. It gives access to the attributes of the column or keyword.

3. In the case of a keyword Field and TableValue may appear very similar, but they are fundamentally different: a TableValue is only one of the constituents of Field, just like the set of keywords attached to the Field; there is no direct association between the keywords and individual TableValues.
4. For keywords we used indexing in a row by name. For columns that would be inefficient, so we use indexing by column number. Such an index can also be used to get a Field pointer:

```
//Field* cfp = tab.getCol(ucol);    is the same as:
  Field* cfp = tab.getCol("U");
```

The next example shows how to get access to the contents of the table we just made by looping through all rows or through all columns.

```
//-*- C++ -*-
// Read a table.
//
#include <aips/Table.h>
#include <aips/Row.h>
#include <aips/TableValue.h>
#include <aips/Field.h>
#include <aips/Vector.h>
#include <aips/ArrayIO.h>                  //for cout << vector

main()
{
//
// Open existing table (option Table::Old is default).
//
    Table tab ("TableExample");
//
// Access columns row by row (see Notes 1 and 2).
//
    Int ucol = tab.getColIndex("U");
    Int vcol = tab.getColIndex("V");
    Row row;
    double U, V;
    for (uInt i=0; i<tab.nrow()/100; i++){ //loop through first rows
        row = tab[i];                      //get row i
        row[ucol].get(U);                  //read per column
        row[vcol].get(V);
        cout << U << " " << V << endl;     //and write out
    }
//
// Access columns in one go (see Note 3).
//
    Vector<double> vec(tab.nrow());        //create vector of right size
    Field* cfp = tab.getCol("U");          //get pointer to column
    cfp->getColumn (vec);                  //get column values into vector
```

```
        cout << "U values: " << vec << endl;   //and write out
    //
    // Loop through columns (see Note 4).
    //
        for (uInt j=0; j<tab.ncol(); j++){     //get and write column names
            cout << tab.getCol(j)->getName() << endl;
        }
    }
```

Notes:

1. `nrow()` is one of the `Table` functions to get the table attributes. Others are: `getName()` to get the name of the table, `getType()` to get the name of the table description, `nkey()` to get the number of keywords, `ncol()` to get the number of columns, and the already known `keywords()` to get the row of table keywords.
2. We can get a row from a table by indexing on row-number. Table indexing on name gives a pointer to a column or a keyword, as we already saw in the previous example.
3. A column can be accessed in one go by treating it as an array of values. So a column of scalars is a vector. The `Field` functions `getColumn` and `putColumn` must be used to access an entire column. When getting a column, the receiving array must be empty (*i.e.* zero-length) or its shape must be conformant.
4. The `Field` function `getName()` is an example of how to get the attributes attached to a keyword or column. Other similar functions are: `dataType()` to get the datatype, `comment()` to get the comment string, and the already known `keywords()` to get the row of field keywords.

# 3 Arrays in Tables

# 4 Tables in Tables

# 5 Selecting, Sorting and Iterating

# 6 Table Vectors

# Table of Contents