

The GLAST SAE U9 Functionality

Dirk Petry

7 May 2004

Summary of changes since 6 May

All changes were made in section 4, the detailed description of the implementation:

1. A new datatype was added to the enum `type` in the quantity class: `VECTOR`. For a quantity of type `VECTOR`, the new string vector `vectorQs` is filled with the names of the quantities which form the vector elements. E.g., there are two radio fluxes, `F6cm` and `F11cm`, in the catalog and the A2 developer would like this to be available as an ordered list. Then an additional quantity “flux” would be created with the following properties:

variable	value
name	“radioflux”
comment	“vector of flux values”
type	<code>VECTOR</code>
unit	“Jy”
vectorQs	“F6cm”, “F11cm”

The vector quantities can be access by new methods

```
void getVecQNames(std::string name, std::vector<std::string> *names);
    // get the list of the names of the quantities
    // which form the elements of the VECTOR ‘‘name’’
void getVecValues(std::string name, int row, std::vector<double> *values);
    // get the values of the vector quantity
    // indentified by ‘‘name’’
    // in the given catalog row
void getVecStatErrors(std::string name, int row, std::vector<double> *errors);
    // get the stat. errors of the vector quantity
    // indentified by ‘‘name’’
    // in the given catalog row
void getVecSysErrors(std::string name, int row, std::vector<double> *errors);
    // get the syst. errors of the vector quantity
    // indentified by ‘‘name’’
```

```

        // in the given catalog row
void getSelVecValues(std::string name, int row, std::vector<double> *values);
        // get the values of the vector quantity
        // indentified by "name"
        // in the given (selected) catalog row
void getSelVecStatErrors(std::string name, int row, std::vector<double> *errors);
        // get the stat. errors of the vector quantity
        // indentified by "name"
        // in the given (selected) catalog row
void getSelVecSysErrors(std::string name, int row, std::vector<double> *errors);
        // get the syst. errors of the vector quantity
        // indentified by "name"
        // in the given (selected) catalog row

```

In order to simplify the API, there are no special vector selection methods. But the user can extract the quantity names using the method `getVecQNames` and then put cuts on them.

Summary of changes since 23 April

All changes were made in section 4, the detailed description of the implementation:

1. A new string variable “ucd” was added to the class “quantity”. This variable can contain the unified contents descriptor of the quantity if available. It works as an alternative name for the quantity. E.g. if you use the UCD “PHOT_FLUX_RADIO_5G” in a get method, e.g.,

```
getNValue("PHOT_FLUX_RADIO_5G", row, &flux);
```

you get in the Veron&Veron catalog the same result as for

```
getNValue("F6cm", row, &flux);
```

2. Two more new string variables “statError” and “sysError” were added to the class “quantity”. They contains the names of the quantities which contain the statistical and systematic errors of the quantity in question. If there is no error information and for string quantities, “statError” and “sysError” are empty.
3. Access methods for the error quantities were added:

```

void getStatErrorName(std::string name, std::string statErrName);
        // get the name "statErrName" of the quantity which

```

```

        // contains the statistical error of quantity “name”,
void getSysErrName(std::string name, std::string sysErrName);
        // get the name “sysErrName” of the quantity which
        // contains the systematic error of quantity “name”,

void getStatError(std::string name, int row, double *f);
        // get the value of the statistical error
        // for the quantity identified by “name”,
        // in the given catalog row;
        // value is negative if unavailable
void getSysError(std::string name, int row, double *f);
        // get the value of the systematic error
        // for the quantity identified by “name”,
        // in the given catalog row;
        // value is negative if unavailable

void getSelStatError(std::string name, int row, double *f);
        // get the value of the statistical error
        // for the quantity identified by “name”,
        // in the given catalog row;
        // value is negative if unavailable
void getSelSysError(std::string name, int row, double *f);
        // get the value of the systematic error
        // for the quantity identified by “name”,
        // in the given catalog row;
        // value is negative if unavailable

```

4. All float variables were replaced by double variables. A separate integer type was not introduced since we don't need to save memory and integers can also be contained in double variables.
5. The selection by parsed string was introduced. This serves to impose general selection criteria on the catalog elements.

The condition is stored in a string `m_selection` which is set using the method `setCutString()`. This method parses the string and checks that there are no syntax errors. Otherwise it returns false.

The syntax is suggested to follow the FORTRAN 77 rules. Quantity names are enclosed by square brackets in order to permit spaces, e.g.

```
[f6cm].geq.[f12cm].and.([hr1]-[hr2]).lt.2.3.or.[f6cm].gt.0.05
```

The selection criterion is applied whenever the user uses one of the “select” methods to access catalog data.

The user can also test if the criterion is true for a given row using the method `bool selStringTrue(int row)`.

An example of the application is given.

6. A method to retrieve a list of the names of all supported catalogs was added: `void getCatNames(std::vector<std::string> *names)`

1 Purpose

The U9 functionality, as originally defined in the Standard Analysis Environment (SAE) design document (September 2002), is that of providing access from SAE tools to (a) the host of astronomical catalogs available from websites such as CDS and NED and (b) catalogs of GLAST origin.

Examples for catalogs of non-GLAST origin are the Third EGRET Catalog, catalogs of flat spectrum radio quasars, catalogs of millisecond pulsars, catalogs of OB associations etc. Examples for GLAST catalogs are first of all the GLAST pointsource catalog, but there may also be other sub-catalogs generated.

U9 shall provide the methods such that tools like the model definition tool U7 or the source identification tool A2 can import a catalog into an internal representation with which they can work on their respective tasks.

The catalog import method for a particular catalog will in general be dependent on the version of the catalog and therefore change with time.

In order to ensure that the appropriate catalog version be available, the tool shall also provide methods to save and load a copy of the imported catalog in a FITS format corresponding to the internal catalog representation mentioned above. This GLAST-internal representation of the catalog can then optionally be distributed with the tool.

2 Functionality

The tool shall offer methods for

1. importing the selected part of the catalog in the internal representation
2. accessing the data elements of the internal representation
3. saving the internal representation into a FITS file
4. loading the internal representation from a FITS file

2.1 Import

2.1.1 Catalog name

The user of U9, typically an application programmer, shall be able to select a catalog name from a predefined list. The developer of U9 will include all catalogs of interest in this list and test the proper download.

2.1.2 Subselection

The user shall be able to select a subset of the catalog identified by the input catalog name in at least three ways:

1. by region: the user defines an elliptical or “rectangular” region of the sky for which the catalog sources shall be selected

2. by imposing a lower and upper limit on a numerical property such as flux
3. by selecting certain source types (e.g. SNR, XRB etc.) from a list of all types available in the catalog.

To support this option, there shall be an ancillary method to query the catalog for the range of values which each of its data fields can take.

The “output” of the method shall be in the form of a vector of catalog member objects containing the catalogued properties for each selected astronomical object in the catalog.

Which quantities are part of the catalog member (“entry”) object will be decided during the development of U9 and become a fixed property of the code (only changeable by code revisions, but not by the user). Only really unnecessary parts of the catalog shall be omitted.

In addition, there shall be data members containing *summary items* such as “number of objects in the catalog”, “number of objects in the subselection” and minimum and maximum values for each of the quantities in the catalog and the subselection. Furthermore, there shall be a reference to the catalog description in the form of a URL.

2.2 Access

The access to the quantities in the catalog member objects shall be read-only to protect the developer from accidentally changing the catalog.

Wherever possible, the access methods shall have generic names and standardised units such that quantities with the same meaning are accessed in the same way for all catalogs.

2.3 Saving and loading

For each catalog there shall be a method to write the internal representation (the catalog object members and the summary items) into a file of a given name. The format of this catalog file shall be FITS.

There shall also be a method which is the exact inverse of the Save method. It shall restore the internal representation of a catalog by reading from a given FITS file. Sole input to this Load method shall be the catalog file name, all other information shall be contained in the file itself.

3 Implementation

U9 shall be a package (i.e. a class or a set of classes) written in C++ following the coding conventions established by the GLAST collaboration for the GLAST Science Tools.

The development shall make use of the CMT tool and verify that the tool be running both on Linux and on Windows machines.

In order to make each individual catalog accessible, the developer may have to write *custom catalog parsers* which are tailored to read the individual catalog as it is available from public databases. The *Virtual Observatory project* is presently making an attempt to standardize the format of catalogs and publish them in a consistent XML format (named VOTable) on the web. But investigation by the author has shown that so far this format is not mature enough to permit reading all catalogs with just one standard parser.

So the reading process will go roughly through the following steps:

1. get catalog name and subselection criteria from user
2. identify which access method and parser to use for the requested catalog
3. access the catalog (locally or via the internet)
4. parse and import (applying subselection criteria and accumulating summary quantities)
5. finalize summary quantities

If the VOTable format for a particular catalog is mature, this format shall be the preferred input format for the import of this catalog.

4 Detailed description of the implementation

The two classes “quantity” and “catalog” described in the following contain the whole functionality. While “quantity” is just a structure, essentially, for the use in “catalog”, it is the *import* methods of “catalog” which contain the knowledge of the system about individual catalogs. Most other methods of catalog are completely general.

The example contains no actual code for any method. But the class definition and the comments should make it clear what each method has to do.

4.1 The “quantity” class

```
class quantity {

public:
    std::string name;           // The name of the quantity, e.g., "hr1"
    std::string comment;        // Explanation, e.g., "hardness ratio 1-3keV/3-6keV"
    enum {NUM, STRING, VECTOR} type; // The datatype
    std::string unit;           // The unit description, "1" if dimensionless,
                               // "" if string
    std::string ucd;            // Unified contents descriptor (if available, "--" otherwise
                               // follows either UCD1 standard or UCD1+ (still t.b.d.)
    std::vector<std::string> vectorQs; // If type == VECTOR, this vector of strings
```

```

                                // contains the ordered list of quantity names
                                // which constitute the vector elements.
                                // All vector elements have to be numerical
                                // quantities.
                                // If type!=VECTOR, vectorQs is empty.
int index;                                // Where to find the quantity in the Strings or
                                         // Numericals vector
bool isGeneric;                            // True if the quantity is part of the set of
                                         // quantities common to all catalogs
bool isLoaded;                             // True if the data corresponding to this quantity
                                         // was loaded into memory
std::string statError;                     // For numerical quantities: The name of the quantity from
                                         // the same catalog which contains the statistical error
                                         // of this quantity; empty if unavailable or for strings
std::string sysError;                      // For numerical quantities: The name of the quantity from
                                         // the same catalog which contains the systematic error
                                         // of this quantity; empty if unavailable or for strings

// selection criteria
std::vector<string> excludedS;           // for numerical values: empty
                                         // for string values: stores the list of values
                                         // which lead to exclusion of a row
std::vector<string> necessaryS;            // for numerical values: empty
                                         // for string values: stores the list of values
                                         // which are necessary for the inclusion of a row
double lowerCut;                          // for string values: undefined; default == 1E-99
                                         // for numerical values: stores the minimum value
                                         // necessary for inclusion of a row
double upperCut;                          // for string values: undefined; default == 1E99
                                         // for numerical values: stores the maximum value
                                         // necessary for inclusion of a row
std::vector<double> excludedN;            // for string values: undefined; default: empty
                                         // for num values: stores the list of values which
                                         // lead to exclusion of a row
std::vector<double> necessaryN;            // for string values: empty; default: empty
                                         // for num values: stores the list of values which
                                         // are necessary for the inclusion of a row
}

```

4.2 The catalog class

```
class catalog {  
  
private:  
  
    std::string m_title;  
    std::string m_URL;  
    std::string m_ reference;  
  
    std::vector<quantity> m_quantities; // the definition of the catalog  
  
    std::vector<std::vector<std::string>> m_strings;  
        // stores all string contents of the catalog;  
        // e.g. strings[3] gives you the vector containing  
        // the values of quantity 3 from the catalog;  
        // strings[3][25] gives you the value of quantity 3  
        // for catalog entry 25  
  
        // comment: the number string quantities in the catalog == strings.size()  
  
    std::vector<std::vector<double>> m_numericals;  
        // stores all numerical contents of the catalog  
  
        // comment: the number of numerical quantities in the catalog == numericals.size()  
  
    int m_numRows;  
        // number of catalog rows loaded into memory  
        // ( == strings[0].size() if strings.size() != 0 which should always  
        // be true due to the existence of generic quantities )  
        // ( and also == numericals[0].size if numericals[0] != 0 which should  
        // always be true due to the existence of generic quantities )  
  
    std::vector<bool> m_rowIsSelected;  
        // true by default; rowIsSelected.size() == numRows;  
        // if selection criteria were applied, true if  
        // selection criteria are true for the corresponding row  
  
    bool m_selRegion; // true if an elliptical region is to be selected;  
                    // default == false  
    double m_selEllipseCentRA_deg; // the RA of the center of the ellipse (degrees)  
    double m_selEllipseCentDEC_deg; // the DEC of the center of the ellipse (degrees)  
    double m_selEllipseMinAxis_deg; // the size of the minor axis (degrees)  
    double m_selEllipseMajAxis_deg; // the size of the major axis (degrees)  
    double m_selEllipseRot_deg;
```

```

        // the rotation angle, i.e. the angle between major axis and the
        // celestial equator (degrees); default == 0

    std::string m_selection; // to contain a general cut which is parsed
                            // by the method selStringTrue()

public:

    // Methods for accessing data
    //-----

    // accessing the catalog definition

    void getQuantityIter(std::vector<quantity>::const_iterator *iter);
        // get an iterator on the quantities
    void getQuantityDescription(std::vector<quantity> *myQuantities);
        // get a copy of the quantity vector
    void getQuantityNames(std::vector<std::string> *names); // get only the names
    void getQuantityUnits(std::vector<std::string> *units); // get only the units
    void getQuantityUCDs(std::vector<std::string> *ucds); // get only the ucds
    void getQuantityTypes(std::vector<std::string> *types); // get only the types

    void getStatErrorName(std::string name, std::string statErrName);
        // get the name ‘‘statErrName’’ of the quantity which
        // contains the statistical error of quantity ‘‘name’’
    void getSysErrorName(std::string name, std::string sysErrName);
        // get the name ‘‘sysErrName’’ of the quantity which
        // contains the systematic error of quantity ‘‘name’’

    void getVecQNames(std::string name, std::vector<std::string> *names);
        // get the list of the names of the quantities which
        // form the elements of the VECTOR ‘‘name’’

    // accessing all the catalog contents in memory
    //      (ignoring in-memory selection criteria)

    void getNumRows(int *nrows); // get the number of rows in the catalog
    void getSValue(std::string name, int row, std::string *string);
        // get the value of the string quantity
        // indentified by ‘‘name’’
        // in the given catalog row
    void getNValue(std::string name, int row, double *f);
        // get the value of the numerical quantity
        // indentified by ‘‘name’’
        // in the given catalog row

```

```

void getVecValues(std::string name, int row, std::vector<double> *values);
    // get the values of the vector quantity
    // indentified by ‘‘name’’
    // in the given catalog row
void getStatError(std::string name, int row, double *f);
    // get the value of the statistical error
    // for the quantity identified by ‘‘name’’
    // in the given catalog row;
    // value is negative if unavailable
void getSysError(std::string name, int row, double *f);
    // get the value of the systematic error
    // for the quantity identified by ‘‘name’’
    // in the given catalog row;
    // value is negative if unavailable
void getVecStatErrors(std::string name, int row, std::vector<double> *errors);
    // get the stat. errors of the vector quantity
    // indentified by ‘‘name’’
    // in the given catalog row
void getVecSysErrors(std::string name, int row, std::vector<double> *errors);
    // get the syst. errors of the vector quantity
    // indentified by ‘‘name’’
    // in the given catalog row
void getObjName(int row, std::string *name);
    // access to generic quantity ‘‘name’’
double RA_deg(int row);           // access to generic quantity ‘‘RA’’ (degrees)
double DEC_deg(int row);          // access to generic quantity ‘‘DEC’’ (degrees)
double posError_deg(int row);     // access to generic quantity
                                // ‘‘position uncertainty’’ (degrees)
double l_deg(int row);           // access to generic quantity ‘‘l’’ (degrees)
double b_deg(int row);           // access to generic quantity ‘‘b’’ (degrees)

bool selStringTrue(int row);      // returns true if the condition described by
                                // m_selection is true for the given row,
                                // otherwise false

// possible values and range
void getSValues(std::string name, std::vector<string> values);
    // for numerical values: empty
    // for string values: the list of different values assumed
    // by this quantity in the given catalog
double minVal(std::string name);
    // for string values: undefined; default == 1E-99
    // for numerical values: the minimum value of this
    // quantity in the catalog
double maxVal(std::string name);
    // for string values: undefined; default == 1E99

```

```

// for numerical values: the maximum value of this
// quantity in the catalog

// accessing the catalog contents with in-memory selection criteria applied
// the row index relates to the selected rows, i.e. is continuous!

void getNumSelRows(int *nrows);
    // get the number of selected rows in the catalog
void getSelSValue(std::string name, int srow, std::string *string);
    // get the value of the string quantity
    // indentified by "name"
    // in the given catalog row
void getSelNValue(std::string name, int srow, double *f);
    // get the value of the numerical quantity
    // indentified by "name"
    // in the given catalog row
void getSelVecValues(std::string name, int row, std::vector<double> *values);
    // get the values of the vector quantity
    // indentified by "name"
    // in the given catalog row
void getSelStatError(std::string name, int row, double *f);
    // get the value of the statistical error
    // for the quantity identified by "name"
    // in the given catalog row;
    // value is negative if unavailable
void getSelSysError(std::string name, int row, double *f);
    // get the value of the systematic error
    // for the quantity identified by "name"
    // in the given catalog row;
    // value is negative if unavailable
void getSelVecStatErrors(std::string name, int row, std::vector<double> *errors);
    // get the stat. errors of the vector quantity
    // indentified by "name"
    // in the given catalog row
void getSelVecSysErrors(std::string name, int row, std::vector<double> *errors);
    // get the syst. errors of the vector quantity
    // indentified by "name"
    // in the given catalog row
void getSelObjName(int srow, std::string *name);
    // access to generic quantity "name"
double selRA_deg(int srow);
    // access to generic quantity "RA" (degrees)
double selDEC_deg(int srow);
    // access to generic quantity "DEC" (degrees)
double selPosError_deg(int srow);

```

```

                // access to generic quantity "position uncertainty" (degrees)
double selL_deg(int srow);
                // access to generic quantity "l" (degrees)
double selB_deg(int srow);
                // access to generic quantity "b" (degrees)
// more quantities here? (the same as above of course)

// possible values and range

void getSValues(std::string name, std::vector<string> values);
        // for numerical values: empty
        // for string values: the list of different values assumed
        // by this quantity in the given catalog
double minSelVal(std::string name);
        // for string values: undefined; default == 1E-99
        // for numerical values: the minimum value of this
        // quantity in the catalog
double maxSelVal(std::string name);
        // for string values: undefined; default == 1E99
        // for numerical values: the maximum value necessary
        // for inclusion of a row

// Methods for selecting data
//-----
// All setting or unsetting of cuts immediately takes effect,
// i.e. the value of the vector m_rowIsSelected is recalculated.
// The next time any of the above getSelectValue methods is
// used, it is taking into account the changed cuts.

void unsetCuts();           // unset all cuts on all quantities except the
                           // selection ellipse;
                           // this also deletes the selection string

// comment: all "'unset'" methods have only an effect if the cuts were
// applied after loading the data and the eraseNonSelected method was not called

void unsetCuts(std::string name);
                           // unset all selection criteria relating to
                           // quantity "name"

void setLowerCut(std::string name, double cutValue);
        // set and apply a cut on quantity "name"
        // such that all values >= cutValue pass,

```

```

void setUpperCut(std::string name, double cutValue);
    // set and apply a cut on quantity ``name''
    // such that all values <= cutValue pass
void setLowerVecCuts(std::string name, std::vector<double> *cutValues);
    // set and apply a cuts on quantities in
    // VECTOR type quantity ``name''
    // such that all values >= cutValue[i] pass,
void setUpperVecCuts(std::string name, std::vector<double> *cutValues);
    // set and apply a cut on quantities in
    // VECTOR type quantity ``name''
    // such that all values <= cutValue[i] pass
void excludeS(std::string name, std::string val);
    // exclude all rows which have string quantity ``name''
    // == val
void useOnlyS(std::string name, std::string val);
    // only include all rows which have string
    // quantity ``name'' == val
void excludeN(std::string name, double val);
    // exclude all rows which have num quantity ``name''
    // == val
void useOnlyN(std::string name, double val);
    // only include all rows which have num
    // quantity ``name'' == val

void setSelEllipse(double centRA_deg, double centDec_deg,
                  double minAx_deg, double majAx_deg, double rot_deg);
    // set and apply an elliptical selection region
    // (box cuts can be achieved with the methods below)
void unsetSelEllipse(); // remove the effects of the ellipse selection

void eraseNonSelected(); // erase all non-selected rows from memory

// for convenience: methods for cutting on the generic quantities
void unSetCutsObjName(); // unset all selection criteria relating to the object name
void excludeObjName(std::string s);
    // exclude all rows which have object name == s
void useOnlyObjName(std::string s);
    // only include all rows which have object name == s

void unsetCutsRA(); // unset all selection criteria relating to RA
void setMinRA(double cutValue);
    // set and apply a lower cut on RA
void setMaxRA(double cutValue);
    // set and apply an upper cut on RA
    // such that all values <= cutValue pass

```

```

void unsetCutsDEC();           // unset all selection criteria relating to DEC
void setMinDEC(double cutValue);
                           // set and apply a lower cut on DEC
void setMaxDEC(double cutValue);
                           // set and apply an upper cut on DEC
                           // such that all values <= cutValue pass

void unsetCutsL();            // unset all selection criteria relating
                           // to L (gal. long.)
void setMinL(double cutValue);
                           // set and apply a lower cut on L
void setMaxL(double cutValue);
                           // set and apply an upper cut on L
                           // such that all values <= cutValue pass

void unsetCutsB();            // unset all selection criteria relating
                           // to B (gal. lat.)
void setMinB(double cutValue);
                           // set and apply a lower cut on B
void setMaxB(double cutValue);
                           // set and apply an upper cut on B
                           // such that all values <= cutValue pass

// general cut described by string to be parsed
bool setCutString(std::string s);
                           // set the selection string m_selection
                           // Syntax:
                           // a) quantities are described by giving their name
                           //    in square brackets, e.g. [f6cm]
                           // b) otherwise FORTRAN syntax is used, e.g.
                           //    "[f6cm].geq.[f12cm].and.([hr1]-[hr2]).lt.2.3"
                           //
                           // returns false if there is a syntax error,
                           // true otherwise.

void getCutString(std::string *s); // get a copy of m_selection

// Methods for importing, saving, loading
//-----
void getCatNames(std::vector<std::string> *names);
                           // return a list of all supported catalog names

int import(std::string catName);
                           // import method for loading an entire catalog without selection

```

```

// - recognizes catalog name and identifies the files to load
// - translates the generic quantities
// - loads non-generic quantities
// the method returns the number of loaded rows, -1 if error

void importDescription(std::string catName);
    // read only the catalog description (i.e. fill the
    // m_quantities vector and m_title etc.)

int importSelected();
    // if a catalog description was already loaded, this method
    // does the same as import(), however, it applies selection criteria
    // such that quantities which are not passing the criteria are not
    // loaded;
    // the method returns the number of loaded rows, -1 if error

void save(std::string filename);
    // save the catalog information presently in memory to a FITS file

void saveSelected(std::string filename);
    // like save(), however, storing only the selected rows

int load(std::string filename);
    // same as import(), only the data is loaded from a fits file
    // compatible with the save() method

// Methods for sorting
//-----

// will use sorting methods present in the standard template library

void sortAscend(std::string quantityName);
    // reassign the row numbers by sorting by the given
    // quantity in ascending order

void sortDecend(std::string quantityName);
    // reassign the row numbers by sorting by the given
    // quantity in decending order

}

```

4.3 Examples for usage

4.3.1 Example 1

This program loads the Veron catalog into memory, identifies the 50 closest sources. It then prints the minimum and maximum flux (at 6 cm) of these and stores the 50 sources in a fits file.

```
int n = 50;
int numRows;
double zCut;
double minflux, maxflux;

catalog *myCatalog = new catalog();
numRows = myCatalog->import("VeronVeron2003");
                                         // the catalog names are predefined,
                                         // the import method "knows" them
if(numRows <= 0) exit; // in this example, error handling is unforgiving
myCatalog->sortAscend("z");
myCatalog.getNValue("z", n-1, &zCut);
myCatalog.unSetCuts();           // make sure no other cuts are applied
myCatalog.setLowerCut("z", zCut); // this sets and applies the cut

minflux = myCatalog.minSelVal("F6cm");
maxflux = myCatalog.maxSelVal("F6cm");
cout << "max flux = " << maxflux << ", min flux = " << minflux << endl;
myCatalog.saveSelected("veron50brightest.fits");
```

4.3.2 Example 2

This program loads a circular region around RA = 120.1°, DEC=25.3° with 5° diameter from the Veron catalog into memory and identifies the brightest source (at 11 cm) and prints its name and coordinates.

```
std::string name;
int numRows;

catalog *myCatalog = new catalog();
myCatalog->importDescription("VeronVeron2003");
myCatalog->setSelEllipse(120.1, 25.3, 2.5, 2.5, 0.);
numRows = myCatalog->importSelected();
if(numRows <= 0) exit; // in this example, error handling is unforgiving
myCatalog->sortDecend("F11cm");
getSelObjName(0, &name);
```

```

cout << "Brightest object is " << name
<< " at RA = " << selRA_deg(0)
<< "      DEC = " << selDEC_deg(0);

```

4.3.3 Example 3

This program loads the whole Veron catalog into memory and then prints all sources for which the flux at 6cm is larger than the flux at 11 cm.

```

std::string name;
int i, numRows;

catalog *myCatalog = new catalog();
numRows = myCatalog->import("VeronVeron2003");
if(numRows <= 0){
    cout << "Problem reading the catalog file." << std::endl;
    exit(1);
}
if(! myCatalog->setCutString("[F6cm].gt.[F11cm]") ){ // check return value
    cout << "Syntax error in selection string." << std::endl;
    exit(2);
}

getNumSelRows(&numRows);
for(i=0; i<numRows; i++){
    getSelObjName(i, &name);
    cout << "object " << i << ":" << name
        << "RA = " << selRA_deg(i)
        << " deg, DEC = " << selDEC_deg(i)
        << " deg" << std::endl;
}

```

4.3.4 Example 4

This program loads the whole Veron catalog into memory, then subselects those rows which have radio flux information and then prints the difference between the 6 cm and the 11 cm flux using the vector feature.

Then it subselects those rows which have only 6 cm information but not 11 cm information and prints those values.

```
std::string name, cut;
```

```

    std::vector<std::string> qNames;
    int i, numRows;
    std::vector<double> fluxes;

    catalog *myCatalog = new catalog();
    numRows = myCatalog->import("VeronVeron2003");
    if(numRows <= 0){
        cout << "Problem reading the catalog file." << std::endl;
        exit(1);
    }
    getVecQNames("flux", &qNames);

    excludeN(qNames[0], 0);
    excludeN(qNames[1], 0);

   getNumSelRows(&numRows);
    for(i=0; i<numRows; i++){
        getSelObjName(i, &name);
        getSelVecValues("flux", i, &fluxes);
        cout << "object " << i << ":" << name
            << "flux1-flux0 = " << fluxes[0]-fluxes[1]
            << std::endl;
    }

    unsetCuts();
    cut = qNames[0]+".gt.0.and."+qNames[1]+".eq.0";
    if(! myCatalog->setCutString(cut) ){ // check return value
        cout << "Syntax error in selection string." << std::endl;
        exit(2);
    }
    getNumSelRows(&numRows);
    for(i=0; i<numRows; i++){
        getSelObjName(i, &name);
        getSelVecValues("flux", i, &fluxes);
        cout << "object " << i << ":" << name
            << "flux0 = " << fluxes[0]
            << std::endl;
    }
}

```