README

The program sorter.c reads the contents of a CSV file from the standard input
and sorts its contents using a column selected by the user.
The program sorter.c can be compiled using the provided Makefile as follows:

    make sorter

The make process can also be started by giving the "make" command alone in the
source directory.
After building the executable file, the program can be run in the following way:

    ./sorter -c <column_name>

column_name is the name of the column to use for sorting the CSV data. The column
name must correspond to one of the column names found at the top of the CSV data.
When the command is run, the CSV data must be given line by line and ending with
EOF or Ctrl-D in the linux terminal. After running, the program will output the
sorted CSV data on the standard output.
The CSV contents can also be passed to the program using pipes as follows:

    cat file.csv | ./sorter -c <column_name>

This will give the contents of file.csv as input to the sorter program. Also, a
new file with the sorted contents can be created by redirecting the standard
output as follows:

    cat file.csv | ./sorter -c <column_name> > sorted.csv

The file sorted.csv will contain the CSV data from file.csv sorted by the
selected column_name.

In order to make the sorter program general and allow to use is to sort arbitrary
CSV files we abstracted the CSV data format using structures.
The main structure used by the program is the CSV structure defined as follows:

```
typedef struct
{
    Data  *header;  /* contains the names of the columns */
    int    *types;  /* type of each column */
    Data   **rows;  /* rows of data */
    int     nrows;  /* csv table dimensions */
    int     ncols;
}CSV;
```

The structure contains all the data for a given CSV and organizes it in special
fields to be handled more easily.
The first field is the header data and it is used to hold the contents of the
first line of the CSV, which is assumed to contain the names for the CSV columns.
The second field is called types and is an array of integers, each integer represents
the type for the corresponding column. We can detect and manage only the following
types and corresponding type values:

```
STRING  = 0
INTEGER = 1
FLOAT   = 2
```

The following field is called rows and it contains all the actual data from the
CSV. We chose to organize the data by rows, each row being an array of columns.
By using this approach, the rows data is effectively a matrix and every single
data in the csv can be accessed directly as a matrix. For instance, the data at
row r and column c can be accessed using rows[r][c].
The last two fields correspond to the CSV data dimensions, which are the number of
rows and the number of columns.

As it can be seen, the data is represented by a special type called Data. We used this approach in order to access the data in each column as it was intended. Instead of accessing all data in the file as character strings, the data can be used as any of the 3 types described previously.
In order to handle this multitype data elements we define a union that holds any of the supported data types. By using this, only one data type is available for a given column and the selected data type for the column can be verified in the types field for the CSV.
The union definition is the following:

```
typedef union
{
    char *s;    /* string data */
    int   i;    /* integer data */
    double f;   /* float data */
}Data;
```

Initially all data is loaded as string data. After all data from the CSV is loaded in the CSV structure, the data is scanned to detect the type of each column. The array of types is then set to indicate the corresponding columns type. Finally, all the columns that are not of the STRING type are converted to either INTEGER or FLOAT. The sorting functions use a special comparison function that reads the column type from the types array and compares the data using any of the s, i or f fields in the Data union.