Working With CSV Data in RPG

Comma-separated values (CSV) files are still widely used for data interchange. The IBM i operating system and its predecessors provide a CL command called Copy From Import File (CPYFRMIMPF), which can read a CSV formatted file and convert it to a database file. At times, reading the CSV file, record-by-record, into an RPG program would be more convenient. There are even times when having more control over how the CSV file is read and parsed would be nice. In this article, I show you how to do that.

This article presents a utility that I wrote to make it easy to read a CSV file in RPG. I start out by explaining the complexity of the CSV format and the problems that need to be overcome. I follow that with a section that describes the finite state machine that I used to overcome the challenges. Finally, I describe the utility, which is an ILE service program, and show you how to use it. If you're uninterested in how things work, you can skip directly to the description of the utility in the section titled "The CSV Service Program."

CSV Versus Other Delimited Files

In a previous article I wrote, I demonstrated how to read a pipe-delimited file. The same technique used for a pipe-delimited file would work also work for a tab-delimited file. Would it work for CSV as well? No, unfortunately, it wouldn't. Indeed, I recommend that you use tab-delimited or pipe-delimited files in preference to CSV if you can, and I'll explain why.

A delimited file works by providing a particular character that acts as a "delimiter." That's how your program knows when one field ends and another field begins. It looks for that delimiter. That means that the delimiter character can't be a part of the data in the field. Consider the following example:

```
1001,John Smith,123 Main St,Milwaukee,WI
1002,Mary Johnson,321 Stately Ave,Chicago,IL
1003,George Jefferson,789 East Side Rd,New York,NY
```

This is an example of a comma-delimited file with three records, each record having five fields. The fields are the account number, customer name, street address, city, and state, respectively.

A program looking to use those fields would read a record and would scan for comma characters; when it finds a comma, the program knows it's done with the account number field, and then it reads data for the name field. When it finds a comma again, the program starts reading the address field, and so forth.

All delimited formats work essentially the same way, but instead of a comma, they might use a tab character or a pipe character. So why does it matter? Why do I say a tab or pipe is better? Because they're not used as frequently in data! Imagine this record:

```
1004,Klement, Scott C.,500 W Lily St, Apt B,Orlando,FL
```

This is supposed to be in the same format, except the name is written as "Klement, Scott C." and the address has an apartment number, separated from the street name by a comma. What will happen when this file is read? Ooops! The program will think the comma after 'Klement' is a delimiter and not part of the data. Another example is when you have a suffix, such as "Bob Smith, Jr." Commas just aren't good as a delimiter, and problems like this aren't as common when you use a tab or a pipe, because tabs and pipes rarely occur in business data. For example:

```
1004|Klement, Scott C.|500 W Lily St, Apt B|Orlando|FL
```

In this case, the file is pipe-delimited. The commas don't cause any particular problem because they're not delimiters. The pipes don't cause a problem, because pipes are not typically used in names or addresses.

However, comma-delimited files are hard to get away from, because they are so widely used in the industry. We can't always choose our file formats!

Because of the widespread problem of commas, most CSV tools place quotation marks around character fields. As long as the commas appear within the quoted string, they're considered data characters rather than delimiters. For example:

```
1004,"Klement, Scott C.","500 W Lily St, Apt B","Orlando","FL"
```

This is the format that CPYTOIMPF, Microsoft Excel, and many other software packages output. Do you see any flaws in this logic? I do! Okay, now I can have a comma in my data, but I can no longer have a quotation mark. For example, the following file is now invalid:

```
1005,"Robert "Crazy Bob" Johnson",5000 Music Lane,Anywhere,USA
```

Because the person's first name contains quote marks, you have the same problem again. When you put the field in quotes, the quote marks become a delimiter of sorts, so you can no longer have those in your data. CPYTOIMPF solves this problem by doubling the quote marks. So the output from CPYTOIMPF would be as follows:

```
1005,"Robert ""Crazy Bob"" Johnson",5000 Music Lane,Anywhere,USA
```

This is a smart solution because it doesn't introduce anymore characters that have special values. There are other vendors who would use a blackslash to escape the quotes, so you'd have "Robert \"Crazy Bob\" Johnson", but this only continues to cause more problems, because now the backslash character can't be used as data in the fields! So I prefer the double-quotes.

Finite State Machine

Adding quotes to the record also adds an additional measure of complexity. Now not only do I have to scan for commas, I have to operate in different modes. I need to read the characters from the record, looking for commas and splitting the data into fields. When I receive a quote, I need to switch to a different mode of operation in which I'm ignoring commas and just looking for additional quotes. When I receive the closing quote, I switch modes again, entering a mode that looks for a second closing quote, a comma, or any other character. Three different modes, depending on the situation! It's definitely more complicated than just scanning for the next comma.

How do I code this situation? I use something called a Finite State Machine (FSM). In this case, the term "state" is just another term for what I referred to as a "mode." My program can be in three different states:

1. **UNQUOTED** = In this state, I will be reading characters from the record. If I read a quote character, I will switch to QUOTED state. If I read a comma character, I will be done reading an entire field. Any other character is considered normal, and I'll add it to the contents of the field I'm reading.

2. **QUOTED** = This state occurs after I've received a quote mark. When I'm in this state, I ignore any commas and simply keep reading until I receive another quote mark. So if I receive a quote, I will switch to ENDQUOTE state. Any other character will be added to the contents of the field I'm reading.

3. **ENDQUOTE** = This state checks the next character to see if it's a second quote. If it is, the quote is added to the field data, and I reenter QUOTED state. If I receive a comma, it's the end of the field. Any other character is added to the data for the current field and the state will change to UNQUOTED.

Those three states of operation are implemented by the following RPG code:

```
for pos = start to len;

    p_char = %addr(CSV.buf) + VARPREF + pos;

    select;
    when state = UNQUOTED;

       select;
       when char = CSV.flddel;
          leave;
       when char = CSV.strdel1
         or char = CSV.strDel2;
          state = QUOTED;
          qchar = char;
       when %len(peFldData) < max;
          peFldData += char;
       endsl;

    when state = QUOTED;

        select;
        when char = qchar;
           state = ENDQUOTE;
        when %len(peFldData) < max;
           peFldData += char;
        endsl;

    when state = ENDQUOTE;

       select;
       when char = qchar;
           state = QUOTED;
           if (%len(peFldData) < max);
              peFldData += char;
           endif;
       when char = CSV.flddel;
           leave;
       when char = CSV.strdel1
         or char = CSV.strDel2;
           state = QUOTED;
           qchar = char;
       when %len(peFldData) < max;
           state = UNQUOTED;
           peFldData += char;
       endsl;

    endsl;

endfor;
```

The preceding code is part of a subprocedure that reads one field at a time from a delimited file. In order to keep things as flexible as possible, I've used variables for the delimiters. For a CSV file, CSV.flddel will be a comma, CSV.strdel1 will be a double-quote, and CSV.strdel2 will be a single quote, in case both single and double quotes are used as string delimiters. If I want to read a file that's delimited by different characters, such as tabs or pipes, I can change the value of CSV.flddel to be a tab or a comma.

The line that begins with p_char uses pointer logic to read the next character from my record. I could have used %subst() instead, but I chose pointers for performance reasons.

Next, you'll see a big SELECT group that implements the state machine. The variable named *state* is just a number that I compare against named constants that represent the three states: UNQUOTED, QUOTED, and ENDQUOTE. Within each state, there's code that acts differently, checking for different character values, as discussed above. Each character that's not a delimiter is added to the peFldData variable, which will be the value of a field.

The CSV Service Program

If you've followed me so far, you should now understand why it's tricky to write code to properly handle a CSV file. Because I don't want to have to repeat the effort of writing this code every time I read a CSV file, I've put all my CSV logic into a service program, written in RPG, named CSVR4. At the end of this article, you can download the code for the CSVR4 service program so that you can use it in your own programs.

The service program provides six routines:

1. **CSV_open** = opens a CSV file in the IFS. When you open the file, you can specify which delimiters (e.g., commas, pipes, tabs, etc.) should be used. The routine returns a handle to track your open CSV files. You can have as many as 200 CSV files open simultaneously.

2. **CSV_loadrec** = loads the next record from the CSV file into memory. Returns *ON if the record was loaded successfully or *OFF if not.

3. **CSV_getfld** = gets the next field from the CSV record. You can call this in a loop to read all the fields. Fields are always returned as VARYING character fields, though you can use RPG's built-in functions to convert them to other data types.

4. **CSV_rewindfile** = moves the file cursor back to the start so you can reread the same CSV file without having to close and reopen it.

5. **CSV_rewindrec** = moves the field cursor back to the start, so you can reread the fields in the currently loaded record.

6. **CSV_close** = closes an open CSV file.

Here's a simple example that prints a report from a CSV file that contains four fields: account number, name, address, city and state fields. It uses a program-described report, just to keep the code simple.

```
H DFTACTGRP(*NO) ACTGRP('KLEMENT') BNDDIR('CSV')

FQSYSPRT   O    F  132        PRINTER oflind(Overflow)

 /copy CSV_H

D acct            s             4s 0
```

```
D temp            s              4a   varying
D name            s             30a   varying
D addr            s             30a   varying
D city            s             15a   varying
D state           s              2a   varying

D csvfile         s                   like(CSV_HANDLE)
D fc              s             10i 0
D fld             s            132a   varying
D prt             ds           132

 /free

     csvfile = csv_open('/home/klemscot/addrtest.csv');
     except heading;

     dow csv_loadrec(csvfile);

        CSV_getfld(csvfile: temp:  %size(temp));
        CSV_getfld(csvfile: name:  %size(name));
        CSV_getfld(csvfile: addr:  %size(addr));
        CSV_getfld(csvfile: city:  %size(city));
        CSV_getfld(csvfile: state: %size(state));
        acct = %dec( temp: 4: 0);

        if (OverFlow);
           except heading;
           Overflow = *OFF;
        endif;

        except rec;

     enddo;

     csv_close(csvfile);
     *inlr = *on;

 /end-free

OQSYSPRT   E            heading        1  3
O                                         +0 'Acct'
O                                         +2 'Name           '
O                                         +0 '               '
O                                         +2 'Street Address '
O                                         +0 '               '
O                                         +2 'City           '
O                                         +2 'St'
O          E            heading        1
O                                         +0 '----'
O                                         +2 '---------------'
O                                         +0 '---------------'
```

```
O                                          +2 '---------------'
O                                          +0 '---------------'
O                                          +2 '---------------'
O                                          +2 '--'
O          E            rec            1
O                       acct                +0
O                       name                +2
O                       addr                +2
O                       city                +2
O                       state               +2
```

Code Download

You can download the CSVR4 service program from the following link:
http://www.scottklement.com/csv/CSVutil.zip