CCSV Library

The ccsv library is a C-based toolset designed to facilitate the handling of CSV (Comma-Separated Values) files. It offers functionalities for both reading and writing CSV files, providing a flexible and efficient way to manage data stored in this popular tabular format.

FEATURES

CSV Reading: Includes functions to parse CSV files, extracting rows and fields based on configurable delimiters, quoting characters, and handling of comments or empty lines.

CSV Writing: Enables the creation and writing of CSV files, allowing users to compose rows and fields with specific delimiters and quoting characters.

USAGE

The library offers intuitive functions for interacting with CSV files, simplifying the process of data extraction, manipulation, and creation. Users can customize settings for CSV parsing or writing based on their specific requirements.

BENEFITS

Versatility: Accommodates various CSV file structures and configurations through customizable options.

Memory Management: Provides robust memory handling mechanisms to prevent memory leaks or overflows while reading or writing CSV data.

NOTE FOR USERS

Error Handling: Ensure proper error handling while using library functions to address potential issues during file operations or data processing, the library provides meaningful error codes.

Memory Management: Properly manage memory allocation and deallocation, freeing resources after usage to prevent memory leaks.

CONTRIBUTING

The ccsv library is open-source and welcomes contributions from the community. Feel free to contribute bug fixes, enhancements, or additional features via pull requests on the official repository.

For detailed usage instructions, function definitions, and code examples, refer to the library's documentation and inline comments in the source code.

LIBRARY CONTENTS

The ccsv library provides functions to read and write csv files:

Reading:

The ccsv library provides fast, powerful functions for seamless reading of CSV (Comma-Separated Values) files in C. It simplifies the process of extracting data from CSV files, offering flexible configurations and efficient parsing capabilities.

Features

Configurable Options

Users can customize various options to tailor the CSV reading process according to specific requirements:

Delimiter Settings: Define the delimiter character used to separate fields within a CSV file. Configure the library to recognize custom delimiters or default to standard comma separation.

Quoting Characters: Specify quoting characters used to enclose fields, allowing for the inclusion of delimiter characters within fields without splitting.

Handling Comments & Empty Lines: Control how the reader manages comments and empty lines within CSV files, providing flexibility in data extraction.

CSV Format

RFC 4180 is an informational memo which attempts to document the CSV format, especially with regards to its use as a MIME type. There are a several parts of the description documented in this memo which either do not accurately reflect widely used conventions or artificially limit the usefulness of the format.

The main differences between the RFC and ccsv are:

1. Each record is located on a separate line, delimited by a line break (CRLF).
   But CCSV recognises CRLF, CR and LF as line terminators.

2. Each line should contain the same number of fields throughout the file.
   CCSV does not care if there are different or same number of fields in all rows.

3. Spaces are considered part of a field and should not be ignored.
   CCSV offers a reader option to skip_initial_spaces
   If skip_initial_spaces = 1, then CCSV will ignore the leading spaces before the start of the field
   If skip_initial_spaces = 0, then CCSV will consider the leading spaces as the part of the field.

Usage

To read CSV files using the ccsv library, users can follow these steps:

Initialize Reader: Create a CSV reader instance with desired options using ccsv_init_reader.

Read Rows: Utilize the read_row function to extract rows from the CSV file. This function parses each row into fields based on the specified delimiter and other configurations.

Access Data: Retrieve the extracted data in the form of a CSVRow structure, containing an array of fields representing each row's values.

Memory Management: Properly handle memory deallocation by freeing resources allocated for rows and fields using the provided free_row function.

Initializing reader -

```
// Reader object
ccsv_reader *reader = ccsv_init_reader(NULL); // NULL for default options
```

Or

```
ccsv_reader_options options = {
        .delim = ',',
        .quote_char = '"',
        .skip_initial_space = 0,
        .skip_empty_lines = 1,
        .skip_comments = 1};

// Reader object
ccsv_reader *reader = ccsv_init_reader(&options);

/* USAGE */

free(reader); // Free the memory allocated to the reader after using
```

<u>Reading a row -</u>

First open the file

```
FILE *fp = fopen("data.csv", "r"); // Specify the path to your file
```

```
if (fp == NULL)
{
	printf("Error opening file\n");
	exit(1);
}
```

Then read row using –

```
CSVRow *row = read_row(fp, reader);
```

read_row will return NULL if all rows are read.

CSVRow struct has two values –

1. fields_count : number of fields present in that row
2. fields : array of fields

read_row returns the row, which is allocated on heap dynamically. So, you must free it to ensure no memory leaks.

Free the row using –

```
free_row(row); // Free the memory allocated to the row
```

<u>Accessing row values -</u>

You can read all rows easily using a while loop like this –

```
// Read each row and print each field
while ((row = read_row(fp, reader)) != NULL)
{
	int row_len = row->fields_count; // Get number of fields in the row
	for (int i = 0; i < row_len; i++)
	{
		printf("%s\t", row->fields[i]); // Print each field
	}
	printf("\n");
	free_row(row); // Free the memory allocated to the row
}
```

Also, the reader struct contains a variable rows_read which specifies how many rows are read using that reader.

```
printf("Rows read: %d\n", reader->rows_read); // Print number of rows read
```

<u>Managing memory –</u> free(), free_row() are used to deallocate memory as used above.

Also remember to close the file using fclose().

Handling errors during reading rows

The functions used above return meaningful error code, if any error occurs during parsing rows. Like in case of Memory allocation failure for the values

CCSV_ERNOMEM – if memory allocation fails

Here, is a code demonstrating how you can manage those situations –

```
#include <stdio.h>
#include <stdlib.h>

#include "ccsv.h"

int main(void)
{
    FILE *csv_file = fopen("../../ign.csv", "r");
    if (csv_file == NULL)
    {
        perror("Error opening file");
        return 1;
    }

    ccsv_reader_options options = {
        .delim = ',', // Example delimiter, change according to your CSV
file
        .quote_char = '"',
        .skip_initial_space = 0,
        // Add other options if necessary
    };

    ccsv_reader *reader = ccsv_init_reader(&options);
    if (reader == (void *)CSV_ERNOMEM)
    {
        fprintf(stderr, "Error initializing CSV reader\n");
        fclose(csv_file);
        return 1;
    }

    CSVRow *row;
    while (1)
    {
        row = read_row(csv_file, reader);
        if (row == NULL)
        {
            break;
        }
        else if (row == (void *)CSV_ERNOMEM)
        {
            fprintf(stderr, "Memory allocation failure while reading
row\n");
            break;
        }

        int fields_count = row->fields_count;
        for (int i = 0; i < fields_count; ++i)
        {
            printf("%s\t", row->fields[i]);
        }
        printf("\n");
        free_row(row);
    }
    printf("\n\nRows read: %d\n", reader->rows_read);

    fclose(csv_file);
    free(reader);

    return 0;
}
```

Writing:

The ccsv library offers comprehensive tools for composing and writing data into CSV (Comma-Separated Values) files efficiently. It simplifies the process of creating structured CSV data, providing customizable options and robust writing capabilities.

Features

Customizable Configuration

Users can tailor various options to suit their specific CSV writing needs:

Delimiter Definition: Define the delimiter character to separate fields within the CSV file. Flexibility to specify custom delimiters or utilize default settings.

Quoting Characters: Specify quoting characters used to enclose fields, ensuring proper handling of special characters or delimiters within fields.

Efficient Writing Mechanism

The library facilitates the creation of structured CSV files by efficiently formatting and organizing data into rows and fields according to user-defined settings.

Dynamic Field Management

Users can dynamically add fields to rows, enabling the flexibility to handle varying data structures or changing content during the writing process.

Usage

To write data into CSV files using the ccsv library, follow these steps:

Initialize Writer: Create a CSV writer instance with desired options using ccsv_init_writer.

Write Rows: Use functions like write_row or write_row_from_array to compose and write rows to the CSV file. These functions allow the addition of rows based on provided field data and configurations.

Field Composition: Specify fields within each row using an array or individual field values, allowing for structured data representation.

Memory Management: Properly handle memory deallocation by freeing allocated resources after writing data to the CSV file.

Initialize writer:

```
// Initialize ccsv_writer_options
ccsv_writer_options options = {
    .delim = ',',
    .quote_char = '"'
    // Add other options if necessary
};

// Initialize the writer
ccsv_writer *writer = ccsv_init_writer(&options); // Pass NULL for default
options
```

You should check for any error before proceeding,

```
if (writer == NULL || writer == (void *)CCSV_ERNOMEM)
{
    fprintf(stderr, "Error initializing CSV writer\n");
    return 1;
}
```

Writing rows:

Before writing to a csv file you should open the file using fopen() and ensure correct opening. For the ccsv library to correctly write content, you must open the file using '**w+**' or '**a+**' mode.

Writer output –

For string – "hi", writer will write 'hi' to file (without quotes)

For string – "Hello, World!", writer will write "Hello, World!" (with quotes)

For string – "\"escapedword\"", writer will write """escapedword""" (escaped quotes inside quotes)

Functions provided by ccsv for writing rows –

1. write_row_from_array(fp, writer, fields, row_len)
    a. fp – file pointer
    b. writer – pointer to ccsv_writer struct
    c. fields – array of fields

   d. row_len – number of fields in the array

2. write_row(fp, writer, row)
   a. fp – file pointer
   b. writer – pointer to ccsv_writer struct
   c. row – CSVRow struct

Write row from array –

```
char *row_string[] = {"hi", "hello", "hello, world!", "\"escapedword\"",
"hola", "bonjour"};

write_row_from_array(file, writer, row_string, ARRAY_LEN(row_string)); /*
Write row to file */
```

ARRAY_LEN, a macro provided by ccsv library to calculate length of array

Write from CSVRow struct –

```
CSVRow *row = read_row(source_file, reader);
write_row(dest_file, writer, *row); // Pass the value of the row pointer
```

Additionaly ccsv provides some macros to write with more flexibility,

- CCSV_WRITE_ROW_START(fp, writer)
- CCSV_WRITE_FIELD(fp, writer, string)
- CCSV_WRITE_ROW_END(fp, writer, last_field)

CCSV_WRITE_ROW_START – This macro is used to initiate writing new row to file. Must be called first before writing fields.

CCSV_WRITE_FIELD –
This macro should only be called after calling CCSV_WRITE_ROW_START, otherwise unexpected behaviour occurs. Used for writing a single field to current row. Multiple of this can be called between row start and row end.

CCSV_WRITE_ROW_END –
This macro also should only be called after calling CCSV_WRITE_ROW_START or CCSV_WRITE_FIELD. This ends the row by terminating the line using CRLF. Additionally you can specify one last field to end row, pass NULL if you don't want to add one.

Example –

```
#include <stdio.h>
#include <stdlib.h>

#include "ccsv.h"

int main(void)
{
    // Initialize ccsv_writer_options
    ccsv_writer_options options = {
        .delim = ',',
        .quote_char = '"'
        // Add other options if necessary
    };


    // Initialize the writer
    ccsv_writer *writer = ccsv_init_writer(&options);
    if (writer == NULL || writer == (void *)CCSV_ERNOMEM)
    {
        fprintf(stderr, "Error initializing CSV writer\n");
        return 1;
    }

    FILE *file = fopen("output.csv", "w+");
    if (file == NULL)
    {
        fprintf(stderr, "Error opening file\n");
        free(writer);
        return 1;
    }

    CCSV_WRITE_ROW_START(file, writer);
    CCSV_WRITE_FIELD(file, writer, "hi");
    CCSV_WRITE_FIELD(file, writer, "hello, world!");
    CCSV_WRITE_FIELD(file, writer, "\"escapedword\"");
    CCSV_WRITE_ROW_END(file, writer, NULL);

    short err_status;
    if (ccsv_is_error(writer, &err_status))
    {
        fprintf(stderr, "Error writing CSV row from string: %s\n",
ccsv_get_status_message(err_status));
        fclose(file);
        free(writer);
        return 1;
    }

    fclose(file);
    free(writer);

    return 0;
}
```

The above program will write - hi,"hello, world!","""escapedword"""

More on handling errors

CCSV provides a function ccsv_is_error(), which takes the reader, or writer object as parameter, returns 1 if any error occurred on just previous call of any ccsv function, returns 0 if not error occurred. If error occurred it store the status code to the second parameter of the function.

```c
short err_status;
if (ccsv_is_error(writer, &err_status))
{
    /* Any required clean up */
    return 1;
}
```

ccsv_get_status_message(status) – Returns the string message for the status code

You can find more examples here.

(https://github.com/Ayush-Tripathy/ccsv/tree/main/examples)

AUTHOR

Ayush Tripathy (ayushtripathy33@gmail.com)

BUGS & ENHANCEMENTS

Your contributions are valued! Feel free to contribute enhancements, report bugs, or suggest new features on the GitHub repository: https://github.com/Ayush-Tripathy/ccsv