# C CSV Parser Documentation

## Overview

The included files provide functions to parse CSV (or other character separated) files and/or strings into an accessible structure in the C programming language. The underlying parsed structures make use of doubly linked lists and are allocated on the heap of the running program.

The reason I wrote this CSV parser for C is its ease of access. While there are several other CSV parsers available, I could not find anyone that was as convenient as calling a function on a string or filename parameter and the parsed CSV structure being returned. Most of the parsers I found were more complicated than what I needed.

This CSV parser parses the input file or string and returns the parsed structure allocated on the heap.

This documentation is intended to provide information on how to use the provided CSV parser.

The parser testing process and possible limitations are discussed in Testing and Evaluation respectively.

# Table of Contents

## Basic Tutorial

Once the header file and the implementation file are included in your project, CSV files and strings can be parsed using the following functions:

```
struct csv_table * parse_string_to_csv_table(char * string, char delim, char
quot_char, int strip_spaces, int discard_empty_cells);

struct csv_table * open_and_parse_file_to_csv_table(char * filename, char delim,
char quot_char, int strip_spaces, int discard_empty_cells);
```

The functions take the string to parse (string) or the address of the file to read and parse (filename). The other function parameters control the parser behaviour:

| **Parameter** | **Description** |
|---|---|
| `delim` | This is the character used to separate values. For CSVs it is `,` but it can be set to any symbol. |
| `quot_char` | Character used for making quotes, e.g. `"` or `'` |
| `strip_spaces` | Boolean flag. If true (1), parsed CSV values will be stripped of leading and trailing spaces. |
| `discard_empty_cells` | Boolean flag. If true (1), any value that is an empty string (`""`) will not be added to the parsed structure. |

*Note: If the delimiter is a space character* (`' '`)*,* `strip_spaces` *and* `discard_empty_cells` *will be overridden to* `0` *(false) and* `1` *(true) respectively.*

The return value is a pointer to the structure used to store the parsed CSV values, a `struct csv_table`.

`struct csv_table` is the highest level of the parsed CSV structure. It is made up of a doubly linked list of `struct csv_row`, which are in turn made up of a doubly linked list of `struct csv_cell`. The `struct csv_cell` contains the appropriate parsed string value in its `str` field.

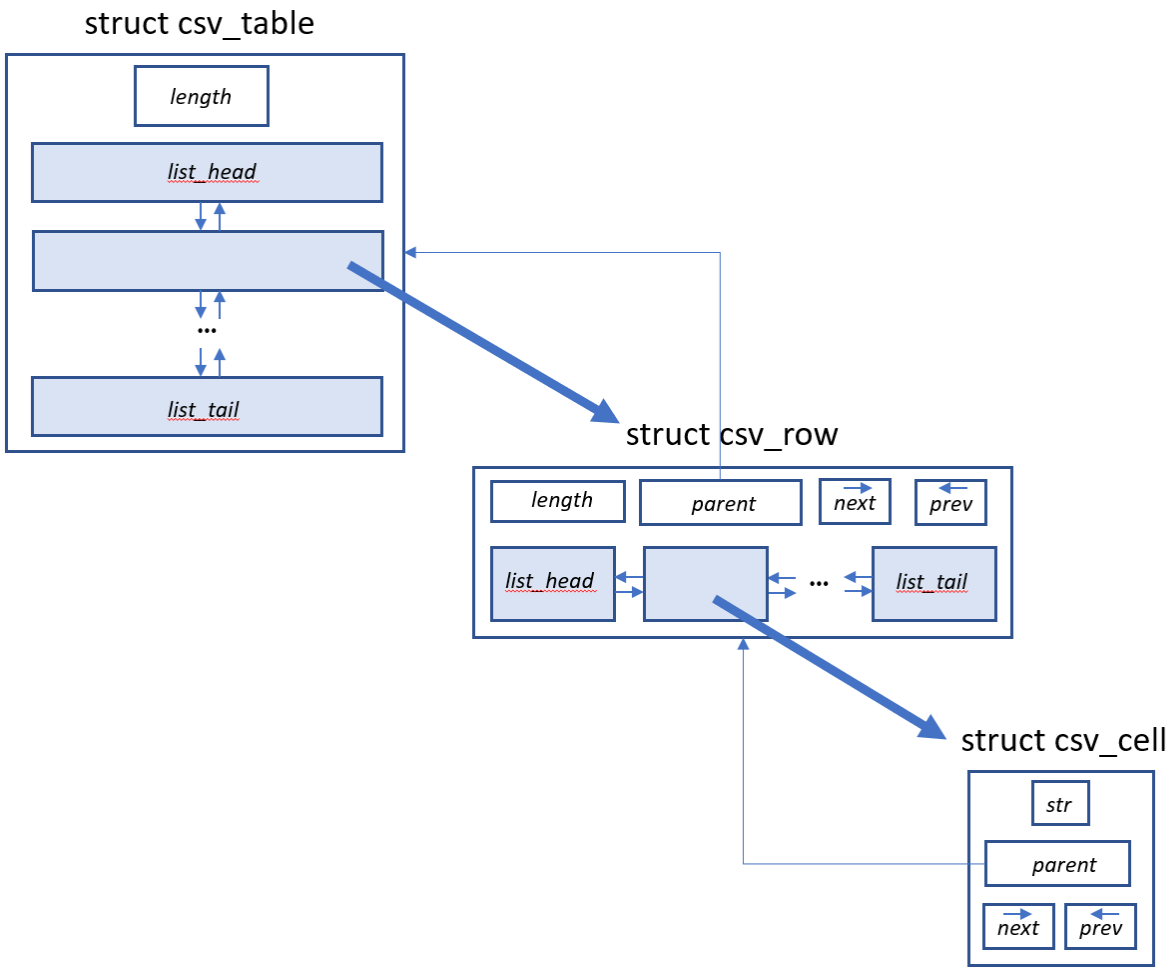


*Figure 1 - Parsed CSV Structure Layout*

The structure is designed to mirror the table layout in the unparsed CSV. The `length` field is the number of CSV rows or CSV Cells in the list of a CSV table or CSV row respectively.

Functions are provided to get a row/cell at a specific row and column index (see Get CSV Structures and Coordinates).

The parsed CSV structure is allocated on the heap and therefore must be freed manually. The header files include free functions for each of the CSV structures to make sure all allocated memory is freed (see Free CSV Structures).

# In-depth Tutorial

## Create CSV Structures

CSV structures can be created in code using the *new_csv* functions.

### Create CSV Cells

A new CSV cell can be created using the following:

```
// Create CSV cell with no string value
struct csv_cell *c1 = new_csv_cell();

// Create a cell with a string value
struct csv_cell *c2 = new_csv_cell_from_str("ACell2");
```

CSV cells can have their `str` field populated/overwritten with a string value using `populate_csv_cell_str`. If the str field already has a value, the function will free the current value and write in the string passed in.

```
// Populate c1 with string
populate_csv_cell_str(c1, "Cell1");

// Overwrite value in c2
populate_csv_cell_str(c2, "Cell2");
```

### Create CSV Rows

New CSV rows can be created with the `new_csv_row`.

```
struct csv_row *r1 = new_csv_row();
```

### Create CSV Tables

New CSV tables can be created with `new_csv_table`.

```
struct csv_table *t1 = new_csv_table();
```

## Clone CSV Structures

The clone functions are provided to clone the contents of the passed in pointer to the CSV structure.

```
struct csv_cell * clone_csv_cell(struct csv_cell * cell);
struct csv_row * clone_csv_row(struct csv_row * row);
struct csv_table * clone_csv_table(struct csv_table * table);
```

The clone function allocates a new CSV structure on the heap and then copies the contents of the source CSV structure. Cloned rows and tables will have their child cells cloned as well.

## Free CSV Structures

CSV Structures are allocated on the heap and therefore must be freed appropriately.

### Free CSV Cells

An allocated CSV cell can be de-allocated using `free_csv_cell`.

```
free_csv_cell(c1);
```

### Free CSV Rows

An allocated CSV row can be de-allocated using `free_csv_row`.

```
free_csv_row(r1);
```

When a CSV row is freed, all cells in its cell list are also freed (using `free_csv_cell`). **Therefore, if you are working with a cell mapped to a row, be sure to clone the cell contents before freeing the row.**

### Free CSV Tables

An allocated CSV table can be de-allocated using `free_csv_table`.

```
free_csv_table(t1);
```

When a CSV table is freed, all rows in its row list are also freed (using `free_csv_row`).

## Print CSV Cell Structures

The contents of csv structures can be printed using the print functions.

### Print CSV Cells

CSV cells can be printed using `print_csv_cell`.

```
print_csv_cell(c1);
```

The function prints the string contained in the cell's `str` field surrounded by quotes.

```
"Cell1"
```

### Print CSV Rows

CSV rows can be printed using `print_csv_row`.

```
print_csv_row(r1);
```

The function prints all the cells in the row's cell list in the order that they appear. The cell list is surrounded by square brackets and a new line character is also printed.

```
["Cell1", "Cell2"]
```

There is also `pretty_print_csv_row`, which prints CSV cell contents with tabulation and newlines.

```
[
        "Cell1",
```

```
        "Cell2"
]
```

### Print CSV Tables

A basic print can be done using the `print_csv_table` function.

```
print_csv_table(t1);
```

This prints the set of CSV rows with their cell contents on one line. A newline character is also printed.

```
[["Cell1", "Cell2"], ["Cell3", "Cell4"]]
```

The function `pretty_print_csv_table` organizes the rows as they are printed. Each row is printed on one line with tabulation.

```
[
        ["Cell1", "Cell2"],
        ["Cell3", "Cell4"]
]
```

There is also `super_pretty_print_csv_table`, which performs a pretty print to the CSV cells in each CSV row in addition to the row organization from `pretty_print_csv_table`.

```
[
        [
                "Cell1",
                "Cell2"
        ],

        [
                "Cell3",
                "Cell4"
        ]
]
```

## Get CSV Structures and Coordinates

### Get CSV Structure at Specified Coordinate

#### *Get Structure Pointers*

The following functions are provided to get a pointer to the CSV structure at a specified coordinate:

```
struct csv_cell * get_cell_ptr_in_csv_row(struct csv_row *row, int index);
struct csv_row * get_row_ptr_in_csv_table(struct csv_table *table, int index);
struct csv_cell * get_cell_ptr_in_csv_table(struct csv_table *table, int rowindx, int
colindx);
```

The functions take the parent row/table and the index for the cell/row.

Use `get_cell_ptr_in_csv_row` to get the pointer to CSV cells at a specific index.

```
// r1 = ["Cell1", "Cell2"]

// Getting "Cell1"
struct csv_cell *cell0 = get_cell_ptr_in_csv_row(r1, 0);

// Getting "Cell2"
struct csv_cell *cell1 = get_cell_ptr_in_csv_row(r1, 1);
```

Use `get_row_ptr_in_csv_table` to get the pointer to the CSV row at the specific index.

```
// t1 = [["Cell1", "Cell2"], ["Cell3", "Cell4"]]

// Getting ["Cell1", "Cell2"]
struct csv_row *row0 = get_row_ptr_in_csv_table(t1, 0);
```

To get a specific cell in a CSV table, `get_cell_ptr_in_csv_table` can be used.

```
// t1 = [["Cell1", "Cell2"], ["Cell3", "Cell4"]]

// Getting "Cell1"
struct csv_cell *cell0 = get_cell_ptr_in_csv_table(t1, 0, 0);

// Getting "Cell3"
struct csv_cell *cell2 = get_cell_ptr_in_csv_table(t1, 1, 0);
```

If the specified index is out of range of the parent CSV structure or the pointer to the parent CSV structure, a NULL pointer is returned.

Note: **The pointer to the CSV structure mapped in the parent structure is returned, meaning any changes made to the pointer's value will affect the parent structure. To get a deep copy, refer to Get Structure Clones.**

#### *Get Structure Clones*

The following functions are provided to get the clone of the CSV structure at the specified index.

```
struct csv_cell * get_cell_clone_in_csv_row(struct csv_row *row, int index);
struct csv_row * get_row_clone_in_csv_table(struct csv_table *table, int index);
struct csv_cell * get_cell_clone_in_csv_table(struct csv_table *table, int rowindx, int
colindx);
```

In this case, the pointer returned points to a clone of the CSV structure at the specified coordinates. This means that it is separate from the parent structure it was taken from.

### Get CSV Cell for a String

The functions below are provided to get the pointer to the CSV cell in the row/table whose `str` field is the same string as the `string` parameter.

```
struct csv_cell * get_cell_for_str_in_csv_row(struct csv_row *row, char * string);
struct csv_cell * get_cell_for_str_in_csv_table(struct csv_table *table, char * string);
```

The pointer to the CSV cell returned will be the first match found in the parent structure.

### Get Coordinates for CSV Structure

For a passed in CSV structure, the provided functions return the coordinates to the CSV structure in the passed in parent CSV structure which contains the same contents as the reference CSV structure.

```
int get_cell_coord_in_csv_row(struct csv_row *row, struct csv_cell *cell);
int get_row_coord_in_csv_table(struct csv_table *table, struct csv_row *row);
```

For example, consider the CSV row r1 which is `["Cell1", "Cell2"]`. We can get the index of the cell that contains the string "Cell1" using the following.

```
struct csv_cell *search_cell = new_csv_cell_from_str("Cell1");
int index = get_cell_coord_in_csv_row(r1, search_cell); // index is 0
```

Note that the reference CSV structure does not have to be a part of the parent CSV structure (row/table) since the search is performed using structure content comparison.

If there is no match found, -1 is returned.

`get_cell_coord_in_csv_table` is provided to get the coordinates of a CSV cell structure in the parent table with the same contents as the reference CSV structure.

```
int get_cell_coord_in_csv_table(struct csv_table *table, struct csv_cell *cell, int
*rowindx, int *colindx);
```

Since a cell coordinate in a table has both a row and column, the function also takes pointers to store the resulting row and column index.

If a match is found, the function will populate the index pointers and return 0. Otherwise, it will return -1.

### Get Coordinates for a String

The below functions have the same behaviour as the functions discussed in Get Coordinates for CSV Structure but instead take a string parameter instead of a reference CSV structure.

```
int get_str_coord_in_csv_row(struct csv_row *row, char *string);
int get_str_coord_in_csv_table(struct csv_table *table, char *string, int *rowindx,
int *colindx);
```

In this case, the returned coordinates are to the first CSV cell that contains the same string as the reference string.

## Combine CSV Structures

The below functions add CSV structures to the list contained in the parent CSV structure.

### Combine Structure Pointers

The below functions are provided to map a child CSV structure into the list of a parent CSV structure.

```
void map_cell_into_csv_row(struct csv_row *rowptr, struct csv_cell *cellptr);
```

```
void map_row_into_csv_table(struct csv_table *tableptr, struct csv_row *rowptr);
```

When a CSV structure is mapped into a parent CSV structure, the pointer to the child CSV structure is added to the end of the doubly linked list contained in the parent CSV structure. Therefore, any changes made to the child CSV structure will be reflected in the parent.

```
struct csv_cell *c1 = new_csv_cell_from_str("Cell1");
struct csv_cell *c2 = new_csv_cell_from_str("Cell2");
struct csv_row *r1 = new_csv_row();

map_cell_into_csv_row(r1, c1); // r1 = ["Cell1"]
map_cell_into_csv_row(r1, c2); // r1 = ["Cell1", "Cell2"]

populate_csv_cell_str(c2, "Apple"); // c2 = "Apple"
                                    // r1 = ["Cell1", "Apple"]
```

To insert a CSV structure at a given coordinate in the parent CSV structure, the *insmap* functions can be used.

```
int insmap_cell_into_csv_row(struct csv_row *row, struct csv_cell *cell, int index);
int insmap_row_into_csv_table(struct csv_table *table, struct csv_row *row, int index);
int insmap_cell_into_csv_table(struct csv_table *table, struct csv_cell *cell, int rowindx,
int colindx);
```

The parameters for the function are the parent CSV structure to insert into, the CSV structure to be inserted and the coordinates at which it is inserted. As it is a mapping, the pointer is inserted into the list at the given coordinates.

```
// r1 = ["Cell1", "Apple"]
struct csv_cell *c3 = new_csv_cell_from_str("Cell3");

insmap_cell_into_csv_row(r1, c3, 1);
// r1 = ["Cell1", "Cell3", "Apple"]
```

The return value of the function indicates if the insertion is successful. It returns 0 if it was successful and false otherwise.

The *insmap* functions can also be used to insert new mappings into the parent CSV structure. This is done by passing in the length of the parent CSV structure as the insertion index.

```
// r1 = ["Cell1", "Cell3", "Apple"]
struct csv_cell *c4 = new_csv_cell_from_str("Cell4");

insmap_cell_into_csv_row(r1, c4, r1->length);
// r1 = ["Cell1", "Cell3", "Apple", "Cell4"]
```

Note that this is the only case where the insmap functions accept a coordinate that is outside of the range of the parent structure's list.

For `insmap_cell_into_csv_table`, if `rowindx` is table->length and `colindx` is 0, a new row is created and the cell is placed as the first entry.

```
// t1 = [["Cell1", "Cell2"], ["Cell3", "Cell4"]]
struct csv_cell *c5 = new_csv_cell_from_str("Cell5");

insmap_cell_into_csv_table(t1, c5, t1->length, 0);
// t1 = [["Cell1", "Cell2"], ["Cell3", "Cell4"], ["Cell5"]]
```

If `colindx` is not 0 for this case, the insertion will not be done. Otherwise, `rowindx` is used to get the row at the appropriate index, and `colindx` is used as the insertion index for the retrieved row using `insmap_cell_into_csv_row`.

```
// t1 = [["Cell1", "Cell2"], ["Cell3", "Cell4"], ["Cell5"]]
struct csv_cell *c6 = new_csv_cell_from_str("Cell6");

insmap_cell_into_csv_table(t1, c6, 1, 2);
// t1 = [["Cell1", "Cell2"], ["Cell3", "Cell4", "Cell6"], ["Cell5"]]
```

### Combine Structure Clones

The previous functions only map the given pointers into the parent structures, meaning that changes to the pointer will affect the parent structure. The below functions clone the structure before mapping it into the specified parent structure.

To add a structure clone to the end of the parent structure (equivalent to `clone.*` and `map.*`):

```
void add_cell_clone_to_csv_row(struct csv_row *rowptr, struct csv_cell *cellptr);
void add_row_clone_to_csv_table(struct csv_table *tableptr, struct csv_row *rowptr);
```

To insert a structure at a specific coordinate in the parent structure ( equivalent to `clone.*` and `insmap.*`):

```
int insert_cell_clone_into_csv_row(struct csv_row *rowptr, struct csv_cell *cellptr, int
index);
int insert_row_clone_into_csv_table(struct csv_table *tableptr, struct csv_row *rowptr, int
index);
int insert_cell_clone_into_csv_table(struct csv_table *tableptr, struct csv_cell *cellptr,
int rowindx, int colindx);
```

The return types are the same as the *insmap* functions.

### Combine String into Structures

The add and insertion functions are also available for strings alone. These functions create a new cell for the string and place them in the parent structure appropriately.

```
int add_str_to_csv_row(struct csv_row *rowptr, char * string);
int insert_str_into_csv_row(struct csv_row *rowptr, char *string, int index);
int insert_str_into_csv_table(struct csv_table *tableptr, char *string, int rowindx, int
colindx);
```

## Separate CSV Structures

### Pop CSV Structures

Cells/rows can be popped from their parent rows/tables using the *pop* functions.

```
struct csv_cell * pop_cell_from_csv_row(struct csv_row * row, int index);
struct csv_row * pop_row_from_csv_table(struct csv_table * table, int index);
struct csv_cell * pop_cell_from_csv_table(struct csv_table * table, int rowindx, int
colindx);
```

The functions take the coordinates of a structure in the parent structure. If the coordinates the valid, the structure at that location is removed from the list of the parent structure and pointer to it is returned. If the coordinates are invalid, a NULL pointer is returned.

```
// r1 = ["Cell1", "Cell3", "Apple", "Cell4"]
struct csv_cell *apple_cell = pop_cell_from_csv_row(r1, 2);

// r1 = ["Cell1", "Cell3", "Cell4"]
populate_csv_cell_str(apple_cell, "Orange"); // apple_cell = "Orange"
                                             // r1 = ["Cell1", "Cell3", "Cell4"]
```

### Delete CSV Structures

CSV Structures can also be deleted from their parent structures using the *delete* functions.

```
void delete_cell_from_csv_row(struct csv_row *row, int index);
void delete_row_from_csv_table(struct csv_table *table, int index);
void delete_cell_from_csv_table(struct csv_table *table, int rowindx, int colindx);
```

These functions perform a pop and free for the specified coordinates.

## Compare CSV Structures

The contents of CSV structures can be compared using the *equals* functions.

```
int csv_cell_equals(struct csv_cell *cell1, struct csv_cell *cell2);
int csv_row_equals(struct csv_row *row1, struct csv_row *row2);
int csv_table_equals(struct csv_table *table1, struct csv_table *table2);
```

These functions return TRUE if the contents of the CSV structures being compared are the same. The below code shows the results of the comparison for a variety of CSV cells.

```
struct csv_cell *c1 = new_csv_cell_from_str("Apple");
struct csv_cell *c2 = new_csv_cell_from_str("Banana");
struct csv_cell *c3 = new_csv_cell_from_str("Apple");
struct csv_cell *c4 = new_csv_cell_from_str("");
struct csv_cell *c5 = new_csv_cell();
struct csv_cell *c6 = NULL;
struct csv_cell *c7 = NULL;

csv_cell_equals(c1, c3); // TRUE (1)
csv_cell_equals(c1, c2); // FALSE (0)
csv_cell_equals(c1, c4); // FALSE

csv_cell_equals(c4, c5); // FALSE
csv_cell_equals(c4, c6); // FALSE

csv_cell_equals(c5, c6); // FALSE
csv_cell_equals(c6, c7); // TRUE
```

For structures that have structure lists (CSV rows and tables), the lists must have the same content in the same order.

```
// r1 = ["Cell1", "Cell3", "Cell4"]
// r2 = ["Cell1", "Cell3", "Cell4"]
// r3 = ["Cell4", "Cell3", "Cell1"]
// r4 = ["Cell1"]
// r5 = []
// r6 = NULL

csv_row_equals(r1, r2); // TRUE
csv_row_equals(r1, r3); // FALSE
csv_row_equals(r1, r4); // FALSE
csv_row_equals(r1, r5); // FALSE

csv_row_equals(r5, r6); // FALSE
```

## Check for Content in CSV Structures

The below functions check if the contents of the specified reference structure are in the parent structure.

```
int is_cell_in_csv_row(struct csv_row *row, struct csv_cell *cell);
int is_row_in_csv_table(struct csv_table *table, struct csv_row *row);
int is_cell_in_csv_table(struct csv_table *table, struct csv_cell *cell);
```

For example, the function `is_cell_in_csv_row` returns TRUE if there is a CSV cell in row that contains the same contents (`str` field) as cell.

```
// r1 = ["Cell1", "Cell3", "Cell4"]
struct csv_cell *c1 = new_csv_cell_from_str("Cell1");
struct csv_cell *c2 = new_csv_cell_from_str("Cell2");

is_cell_in_csv_row(r1, c1); // TRUE
is_cell_in_csv_row(r1, c2); // FALSE
```

There are also functions to perform a check if a string is within a CSV structure.

```
int is_string_in_csv_row(struct csv_row *row, char *string);
int is_string_in_csv_table(struct csv_table *table, char *string);
```

## Iterate Through Row/Cell Lists

The functions `has_next_cell` and `has_next_row` are implemented to provide a conditional check when iterating through a structure's list in a for-loop.

```
int has_next_cell(struct csv_row * row, struct csv_cell * cur_cell);
int has_next_row(struct csv_table * table, struct csv_row * cur_row);
```

These are intended to be used in the case where the for-loop has to go through every element and the resulting implementation is more similar to a for-each loop.

```
// r1 = ["Cell1", "Cell3", "Cell4"]

// Initialization: Set cur_cell to beginning of list
// Conditional: has_next_cell with parent structure and current value iterator
// Incrementation: Set cur cell to the next cell

for( struct csv_cell *cur_cell=r1->list_head; has_next_cell(r1, cur_cell);
cur_cell=cur_cell->next ){
	printf("%s\n", cur_cell->str);
}
```

If specific behaviour needs to be done with the tail of the list, a simple while loop can be done.

```
// r1 = ["Cell1", "Cell3", "Cell4"]
struct csv_cell *cur_cell = r1->list_head;

while ( cur_cell != NULL && cur_cell->next != NULL ){

	printf("%s, ", cur_cell->str);

	cur_cell = cur_cell->next;
}

if ( r1->list_tail != NULL ){
	printf("%s\n", r1->list_tail->str);
}
```

This behaviour applies to rows within tables as well.

```
int total_cells = 0;
for ( struct csv_row *cur_row=table->list_head; has_next_row(table, cur_row);
cur_row=cur_row->next){
	total_cells += cur_row->length;
}

printf("Table has %d cells\n", total_cells);
```

# Evaluation

## Big-O Runtime

The CSV parser and its associated data structure functions all operate on average/worst case O(n).

| Use Case | Big-O | Reasoning |
|---|---|---|
| Parsing CSV | O(n) | There are two cases, in both of which the parser results in O(n). (Calculations are based on the behaviour of the base parser function `parse_fileptr_or_char_array_to_csv_table`).<br><br>Case 1: No cell merges need to happen. Case for parsing string or `fgets` has entire CSV row in its buffer.<br>For a given word of length n, we traverse the buffer n times to find the word delimiter ∴ O(n).<br>The word is copied from the buffer into the allocated cell: |

| | | |
|---|---|---|
| | | O(1) for cell allocation<br>O(n) for word copy (copied character by character)<br>For one word: O(n) + O(n) + O(1) = O(2n) + O(1).<br>This scales to the other words in the buffer: O(2n) + O(1) ≅ O(n)<br><br>Case 2: Cell merges happen (`fgets` does not have entire CSV row in the buffer).<br>In the worst case where a word is split across two buffers from `fgets`, it will have to be merged. To perform this:<br>O(n) to traverse buffer and find delimiter of first part of the word<br>O(n) + O(1) to copy the first part of the word into an allocated cell<br>O(n) to traverse the new buffer and find delimiter and therefore second part of the word<br>O(n) + O(n) + O(1) to copy the first part of the word and the second part of the word to a new cell that has the combined string<br>O(1) to delete old cell from the table.<br>O(n) + O(n) to strip combined string of quotes and spaces<br>O(n + n + 1 + n + n + n + 1 + 1 + n + n) = O(7n + 1) = O(7n) ≅ O(n) |
| Sequential Access | O(n) | Accessing the beginning and end of the list is O(1) since there is the list head and tail pointers.<br>Get CSV Structure functions are designed to start from the end closest to the specified index, therefore worst-case scenario will mean going through n/2 elements ∴ O(n/2) ≅ O(n). |
| Inserting Structures | O(n) | Appending to the end of the list is O(1) since adjusting `list_tail` and incrementing length is constant time.<br>The worst case for an insertion:<br>In a list of n CSV structures, we are inserting the structure at location n-1.<br>This would mean traversing past the first n-1 elements ∴ O(n-1).<br>Adjusting the list pointers is O(1).<br>Worst case: O(n-1) + O(1) ≅ O(n). |
| Checking for Content | O(n) | No starting optimization can be made as content to search for can be placed anywhere in the list. Worst case is going through all list elements to find them therefore O(n). |

## Possible Limitations

### Wasted Characters for Stripping Quotes

The function `malloc_strip_quotes_and_spaces` is designed to remove trailing spaces and unescaped quotes from the string parameter.

```
char * malloc_strip_quotes_and_spaces(char  *string, int len, char quot_char, int
strip_quotes, int strip_spaces, int free_string);
```

The sanitized string is allocated on the heap and the pointer to the string location is returned. This is used when copying words from the raw string buffer into a CSV cell.

The limitation for the function is that source strings with quotes will have wasted character slots in the sanitized string that will have to be padded with null terminators.

Consider the case for the string `"His ""pretty happy"" dog"`. When the memory for the sanitized string is allocated, it is done for the `strlen(His ""pretty happy"" dog)+1` (Note starting and ending quotes are have already been removed).

The sanitized string will remove one of the quotes in each quote pair, resulting in `His "pretty happy" dog`. However, since the space for the sanitized string is allocated before stripping the extra quotes, resulting in two extra character slots which are padded with null terminators. Therefore, the sanitized string is `His "pretty happy" dog`, but as a char array it is `His "pretty happy" dog\0\0\0` when it could be `His "pretty happy" dog\0`.

With this implementation, there will always be a wasted slot for each quote that must be stripped. To fix this, new memory can be allocated for the sanitized string using its actual string length and the string is copied into the new memory without the extra null terminators. This new memory will be what is returned, and the initial allocation will be freed.

The decision was made to maintain the current implementation as the above solution would result in an extra character iteration to be done on every word that needs to be stripped. This would introduce an additional time complexity for every word to be parsed, with the benefits only being to words with several quotes. The additional time complexity for the solution was determined to be of more detriment than the additional space complexity incurred by wasting characters.

Perhaps a future iteration can introduce a flag to be able to control if the string should be doubly sanitized to remove the wasted space.

# Testing

## Testing Process and How to Run.

The parser was tested using the various input files in *testing/inputs*.

The input files were sourced from the GitHub repository: https://github.com/maxogden/csv-spectrum but also includes other added files.

The parser output was compared to the output of the parser included in the python csv module. This was done by running the C parser and python parser on the same input file, formatting the output of the python parser to match the content format printed using `print_csv_table` and performing a string comparison.

To run the test on your local system, follow the below steps.

1. Compile *testing/testparser.c* to *testing/testparser.out*.
2. Run the python script *testing/testingparser.py* in the directory *testing*.

*Note: The python script only prints the results failed test cases, to print all test cases. Use the flag `--all-tests` to print the results of all test cases. The flag `--print-files` can be used to print the contents of the input CSV files and the result.*

**Note: The starting and ending quotes are added in the printed format and are not part of the cell string.**

## Test Results

The only differing case from the python parser is for the file *inputs\my_escaped_quotes.csv*:

```
"ha """ha""" ha"
```

The python parser returned the result:

```
[["ha "ha""" ha""]]
```

The C parser returned the result:

```
[["ha "ha" ha"]]
```

The C parser stripped the starting and ending quotes from the input string and also stripped 2 out of the three quotation marks within it.

This is because the parser removes quotation marks that are not escaped. To escape a quotation mark, there must be a preceding quotation mark that is not used to escape another quotation mark.

```
"a" -> a
""a" -> a"
""a"" -> "a"
""""a"""" -> ""a""
```

In this case, the parser counted the first two quotations as an escaped quotation and therefore added it to the final string. The third quotation was not escaped since the previous quotation is already part of an escaped pair and was therefore stripped from the final string.

To see the raw code on how quotations are stripped, see `malloc_strip_quotes_and_spaces` in *csvparser.c*.