CUDDLE - BOOTSTRAP

CODE YOUR OWN PANDA, BECAUSE EVEN DATA NEEDS A CUDDLE

CUDDLE - BOOTSTRAP

Welcome to the **Cuddle** project!
This bootstrap is designed to help you getting started.

Its exercises lead you to create a `dataframe_t` data structure and read a simple CSV file.

Overview

In this bootstrap, you will:

- ✓ Set up the project structure.
- ✓ Create the necessary header files.
- ✓ Define basic data types and enumerations.
- ✓ Implement the dataframe_t structure.
- ✓ Write a function to read a CSV file line by line.
- ✓ Parse CSV lines to extract data.
- ✓ Store data into the dataframe_t structure.
- ✓ Determine and store the data types of each column.
- ✓ Implement basic dataframe operations (df_info, df_shape).
- ✓ Test your implementation with sample data.

Exercise 1: Defining basic data types and enumerations

Define the column_type_t enumeration

Add the following enumeration to dataframe.h:

```
typedef enum {
    BOOL,
    INT,
    UINT,
    FLOAT,
    STRING,
    UNDEFINED
} column_type_t;
```

How will column_type_t be used when parsing and storing data?
Are there other data types or structures that might be helpful to define at this stage?

Exercise 2: Implementing the dataframe_t Structure

Design the dataframe_t structure

In dataframe.h, define dataframe_t with the following members:

```
typedef struct {
    int nb_rows;            // Number of rows in the dataframe
    int nb_columns;         // Number of columns in the dataframe
    char **column_names;    // Array of column names
    column_type_t *column_types; // Array of column types
    void ***data;           // 2D array of data values
} dataframe_t;
```

Consider data storage options

Think about alternative ways to store data, such as using an array of column_t structures where each column_t contains its own data array.

What are the advantages and disadvantages of different data storage methods?
How does using void * allow for flexibility in data types?

Exercise 3: Writing a function to read a CSV file line by line

Create a sample CSV file

Create a file named `data.csv` with the following content:

```
name,age,city
Alice,25,Paris
Bob,30,London
Charlie,35,Berlin
```

Implement a function to open and read the file

In a new source file, write a function `df_read_csv` that opens a CSV file for reading.

```c
#include "dataframe.h"

dataframe_t *df_read_csv(const char *filename, const char *separator) {
    // Open the file
    // Read the file line by line
    // Return a new dataframe_t instance
}
```

Handle file opening errors

Ensure your function checks if the file is successfully opened and handles errors properly.

Can you open a file that doesn't exist? a file you can't access? a very very big file? ...etc...

Read and print each line

For now, read each line using `fgets` and print it to verify that the file is being read correctly.

What functions are available in C for file I/O operations?
How can you ensure that your program handles files with varying line lengths?

Exercise 4: Parsing CSV lines to extract data

Split each line into tokens

Use the `strtok` function to split each line based on the specified separator (defaulting to a comma , if separator is `NULL`).

Store the tokens temporarily

For now, collect the tokens from each line and print them to verify correct parsing.

```
char *token = strtok(line, separator);
while (token != NULL) {
    printf("Token: %s\n", token);
    token = strtok(NULL, separator);
}
```

Handle end-of-line characters

Be aware of newline characters (`\n`) at the end of each line and remove them if necessary.

How does `strtok` work, and what are its limitations?
How can you handle cases where data fields contain the separator character?

Exercise 5: Storing data into the dataframe_t Structure

Initialize the `dataframe_t` instance

- ✓ Allocate memory for a new `dataframe_t` instance.
- ✓ Initialize `nb_rows` and nb_columns appropriately.

Store column names

- ✓ The first line of the CSV file contains the column names.
- ✓ Store them in the column_names array.

Allocate memory for data storage

- ✓ Allocate memory for the data member to hold all the data.
- ✓ Consider using a dynamic array or reallocating as you read more lines.

Store data rows

For each subsequent line, parse the tokens and store them in the data array.
Initially, store all data as strings.

Update row count

Increment `nb_rows` as you read each data row.

How can you dynamically resize your data arrays as you read more data?
What memory management considerations are there when storing data?

Exercise 6: Determining and storing data types of each column

Analyze data to determine column types

After reading all data, iterate over each column to determine its type.
Implement functions to check if data in a column can be converted to `INT`, `UINT`, `FLOAT`, `or BOOL`.

Store column types

Populate the `column_types` array in your `dataframe_t` with the determined types.

Convert and store data in correct types

Based on the column types, convert the data from strings to the appropriate types.
Update the data array to store data in their actual types (e.g., integers, floats).

Handle mixed-type columns

If a column contains mixed types, default its type to `STRING`.

Free unnecessary memory

If you allocated temporary storage during type conversion, ensure you free it.

How can you implement functions to check if a string represents a valid integer or float?
What should you do if a value cannot be converted to the determined column type?

YOU FORGOT OUR ANNIVERSARY?! I KNEW I SHOULDN'T HAVE MARRIED SOMEONE THAT RUNS OUT OF MEMORY SO QUICK.
8 GB
32 GB
©2014 Nate Fakes |Facebook.com/natefakescartoons
NATE FAKES

Exercise 7: implementing basic dataframe operations

Implement the `df_info` function

Write a function `void df_info(dataframe_t *dataframe)` that:

- ✓ Prints the number of columns.
- ✓ For each column, prints its name and type (in lowercase).

Example output:

```
3 columns:
- name: string
- age: int
- city: string
```

Implement the `df_shape` function

- ✓ Define a structure `dataframe_shape_t` with members `nb_rows` and `nb_columns`.
- ✓ Write a function `dataframe_shape_t df_shape(dataframe_t *dataframe)` that returns the shape of the dataframe.
- ✓ In your main program, print the shape like this: `Shape: 3 rows, 3 columns`.

Test the functions

After reading the CSV, call `df_info` and `df_shape` in your main program, to verify they work correctly.

How does providing such information help users understand their data?
Are there additional useful summary statistics you might include in future functions?

Exercise 8: Testing your implementation with sample data

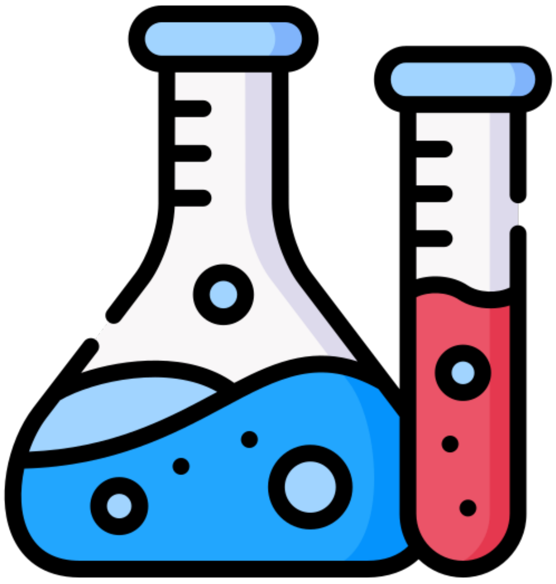
✓ Run your program:
Compile your code and run your program to read `data.csv`.

✓ Check the output:
Ensure that the output from `df_info` and `df_shape` matches the expected results.

✓ Inspect data storage:
Optionally, print out the data stored in the dataframe to verify that data values are correctly stored with the appropriate types.

✓ Test with different data:
Modify `data.csv` to include different types of data (e.g., add a column with boolean values) and test your program again.

Does your program handle different data types correctly?
How does it behave if the CSV file has inconsistent data?



Conclusion

Congratulations! You have successfully:

- ✓ set up a well-organized project structure ;
- ✓ created necessary header and source files ;
- ✓ defined the `dataframe_t` data structure and related types ;
- ✓ written code to read and parse a CSV file ;
- ✓ stored data into the dataframe with appropriate data types ;
- ✓ implemented basic operations to display dataframe information.

Next steps

- ✓ Implement additional functions such as `df_write_csv` to write dataframes back to CSV files.
- ✓ Explore more advanced data manipulation functions like filtering, sorting, and aggregation.
- ✓ Add error handling and input validation to make your library more robust.

Regularly test your code with different datasets to ensure reliability.
Document your code with comments to explain your logic and decisions.
Keep functions small and focused to make debugging easier.

v 1.0.1

{EPITECH}