SHF Cardea Parser Manual

Contents

---

1 Overview

This manual was written to be read and understood by anybody regardless of their background in programming. However, technical terms may be present at times for sake of completeness/conciseness. These can simply be ignored if they are not applicable to one's use case. One should not have to be familiar with any technical terms to still understand this program or troubleshoot a problem. In addition to this manual, the source code is also thoroughly (if not excessively) commented so that it can be understood by someone not fluent with the specific programming language but having general programming knowledge.

This program was intentionally not written in the most efficient way, but was written to prioritize portability (all components of this program resides in one file containing the source code) and beginner-friendliness so that one would require minimal training to be able to maintain this program. For example, the amount of global variables (typically discouraged when writing a program) could have been minimized, but the decision was made to keep them for sake of simplifying/making easier to read the source code. Although the source code of this program may not be the best reference for general programming strategies, it better serves as an educational asset for anyone with basic programming knowledge wishing to learn the C++ programming language itself, as a diverse set of C++ concepts have been included in the implementation of this program along with their explanations throughout this manual and the source code itself.

This program was written in C++ and compiled in Windows 10 Visual Studio 2019, using (and requiring at the earliest) the ISO C++17 Standard. This program is intended for the Windows 10 operating system.

ASCII text art was generated from or adapted from https://patorjk.com/software/taag/. The Saving Hearts Foundation logo was adapted from http://loveascii.com/hearts.html.

Fundamentally, this program consists of three main stages:

1) Sanitation stage
2) Cardea-compatible conversion stage
3) Duplicate action stage

1.1 Sanitation Stage

CSV stands for comma-separated values. As the name implies, individual entries from spreadsheets, like the raw input CSV file for this program, are commonly separated by commas when the file is in the CSV format. This separator character (the comma in this case) is known as a "delimiter."

While there is nothing inherently wrong with commas as delimiters, it does become problematic when one wishes to parse through a comma-delimited document. The reason is that it is not uncommon that the entries delimited by commas contain these very commas themselves. This complicates the parsing process, as one must then keep track of which commas are true delimiters and which are just part of the entries. This is the motive for the sanitation stage.

In this stage, this program "sanitizes" the raw input CSV file by scanning the whole document for the comma delimiter (can be other characters also, see Advanced Setup), ignoring commas deemed to be from entries and replacing commas deemed to be delimiters with an alternative character, a "clean" delimiter that can be user-specified ('$' character by default, see Program Parameters; **DELIMITER_CLEAN**) and is not expected to appear within any entry. Once the document is sanitized, it can be manipulated freely without worry of confusion between characters within entries and as delimiters.

1.2 Cardea-Compatible Conversion Stage

Cardea requires a very specific layout of the file it accepts. While patient information is processed in a separate SHF server which then generates a CSV file storing information that Cardea needs, it does not produce a Cardea-compatible file. This is the motive for the conversion stage (and this program itself).

In this stage, the actual conversion of the raw input CSV file to a Cardea-compatible output file takes place. Instead of starting from the input file and taking away/modifying its entries, this program actually builds the output file from scratch, while importing any appropriate entries from the input as necessary. Other pertinent processing is also carried out during this stage, such as determining if a patient's consent form is on file or handling forms filled out in different languages. Additionally, problematic entries are corrected or pointed out by this program, such as removing extraneous spaces from entries (Cardea does not handle extra spaces well) and warning when fields were left empty. Once the document is parsed, it must now be handled differently from the raw input CSV file, as the new layout would be drastically altered from the original.

1.3 Duplicate Action Stage

After the conversion stage, the file can now technically be read by Cardea. However, it is possible that patients submit their information multiple times for the same screening (such as to update older information or due to network error), resulting in duplicated patient information that can lead to confusion in properly identifying patients during screenings. Unfortunately, the SHF server does not check for duplicate submissions. This is the motive for the duplicate action stage.

In this stage, this program detects and takes action against duplicates, based on a pre-determined configuration of what qualifies as a duplicate. There are two levels of strictness; if the stricter criteria are met, the

duplicate is automatically removed by this program, while only a warning is given if the less strict criteria are met. This program is also configured to detect and warn if patient first and last names are swapped, which might indicate a possible duplicate. Defining these criteria cannot be done through the program interface but must require editing the source code itself, which is explained later in the manual (see Functions and Assets; `track_duplicates_including_this()`). Once duplicates in the document are handled, a finalized output file is produced and ready to be accepted by Cardea.

---

2 Directions

1) Place the raw input CSV file in the same location as this program.
2) Launch this program and follow the prompts.

   To follow this step an alternative way, see Advanced Setup; Accelerated Parameter Initialization.
3) Before the console closes, an event log should be listed, followed by a prompt to exit this program.
4) The Cardea-compatible output should be in the same location as this program along with the log file.

   NOTE: If one runs this program with the same event name, the event logs will be appended to (not write over) the existing log file.

---

3 Program Parameters

Immediately below is a list of all the user-adjustable parameters, their default values, and their C++ data types, separated by colons. Ignore the quotation marks. Each parameter is described in detail afterward.

Express-Available Parameters: one can freely initialize any of the following parameters without worry.

NAME_INPUT : “inputForCardea” : `const std::string`

NAME_OUTPUT : “outputForCardea” : `const std::string`

NAME_EVENT : “screeningName” : `const std::string`

NAME_FORM_PATH : “C:\Users\Bryan\SHF\Heart Screenings\Forms” : `const std::string`

Custom-Only (Advanced) Parameters: one should be more wary about initializing the following parameters. These can only be initialized when this program is run in custom mode (see Advanced Setup; Custom Mode).

FORM_YES : “Yes” : `const std::string`

FORM_NO : ”” : `const std::string`

NAME_LOG : “log_**NAME_EVENT**.txt” : `const std::string`

NAME_INPUT_CLEAN : “sanitized.csv” : `const std::string`

NAME_OUTPUT_DUPLICATES : “duplicates.csv” : `const std::string`

NAME_ENGLISH : “English” : `const std::string`

DELIMITER_CLEAN : '$' : `const char`

DELIMITER_CSV : ',' : `const char`

Below here is a list of parameters that cannot be adjusted through the program interface. Each parameter is described in detail afterward. Adjusting these parameters would require altering the source code and possibly other dependent functions.

Fixed Parameters: one cannot modify these parameters without altering the source code itself.

NAME_FORM_VAR

NAME_FORM

NUM_LANG_FIELDS

HEADERS

3.1 Express-Available Parameters

One can freely initialize any of the following parameters without worry.

3.1.1 NAME_INPUT

Description:

The name of the raw input CSV file to convert into a Cardea-compatible format.

Data Type:

`const std::string`

Default:

"inputForCardea"

Notes:

This program automatically appends the ".csv" extension to the name.

3.1.2 NAME_OUTPUT

Description:

The desired name for the final Cardea-compatible output file.

Data Type:

`const std::string`

Default:

"outputForCardea"

Notes:

This program automatically appends the ".csv" extension to the name.

3.1.3 NAME_EVENT

Description:

The name of the current screening event based on the naming of the consent form files.

Data Type:

```
const std::string
```

Default:

"screeningName"

Notes:

This program automatically checks if patients have already filled out consent forms by scanning the files labeled with this particular event name.

3.1.4 NAME_FORM_PATH

Description:

The absolute path to the directory/folder containing the patient consent forms.

Data Type:

```
const std::string
```

Default:

"C:\Users\Bryan\SHF\Heart Screenings\Forms"

Notes:

Although Windows paths use backslash '\' as the separator, this program accepts forward slashes '/' as well. This program does not proceed until a valid path is given.

3.2 Custom-Only (Advanced) Parameters

One should be more wary about initializing the following parameters. These can only be initialized when this program is run in custom mode (see Advanced Setup; Custom Mode).

3.2.1 FORM_YES

Description:

The text to be written into cells of the "Consent" column of the Cardea-compatible output file when the consent form of a patient is found in the path **NAME_FORM_PATH**.

Data Type:

`const std::string`

Default:

"Yes"

Notes:

Keep in mind that the text used to initialize this parameter may be seen in the final output file itself.

This parameter was included for sake of user control if it ever need be modified in the future. In the meantime, it likely never needs to be changed from the default.

3.2.2 FORM_NO

Description:

The text to be written into cells of the "Consent" column of the Cardea-compatible output file when the consent form of a patient is NOT found in the path **NAME_FORM_PATH**.

Data Type:

`const std::string`

Default:

""

Notes:

Keep in mind that the text used to initialize this parameter may be seen in the final output file itself.

This parameter was included for sake of user control if it ever need be modified in the future. In the meantime, it likely never needs to be changed from the default (which is empty).

3.2.3 NAME_LOG

Description:

The desired name for the log file detailing the events and set parameters from previous runs of this program.

Data Type:

`const std::string`

Default:

"log_**NAME_EVENT**.txt"

Notes:

If this program is run more than once for the same event name initialized to **NAME_EVENT**, the respective log elements are appended to the same log file. In other words, this program never erases previously created logs, only add on to them. The most recent logs are found at the bottom of the log file.

If the source code is ever modified such that the initialization order of parameters is altered, note that the initialization of this parameter depends the prior initialization of **NAME_EVENT** and thus should come after it. This also means that if a typo was made in the initialization of **NAME_EVENT**, a whole separate log file would be created with that typo in its name.

3.2.4 NAME_INPUT_CLEAN

Description:

The name for the intermediate file produced by this program following the sanitation stage.

Data Type:

`const std::string`

Default:

“sanitized.csv”

Notes:

The “.csv” extension is not automatically applied to this parameter.

There is nothing inherently special about the default name of this parameter; the name was chosen to be descriptive.

Unless this program is running in Retain Mode (see Advanced Setup; Retain Mode), files having this name in the same folder as this program are automatically deleted by `Parse()` (see Functions and Assets; `Parse()`). To avoid unintended file deletions, avoid initializing other file-name parameters with the same name initialized for this parameter.

3.2.5 NAME_OUTPUT_DUPLICATES

Description:

The name for the intermediate file produced by this program following the Cardea-compatible conversion stage.

Data Type:

`const std::string`

Default:

“duplicates.csv”

Notes:

The “.csv” extension is not automatically applied to this parameter.

There is nothing inherently special about the default name of this parameter; the name was chosen to be descriptive.

Unless this program is running in Retain Mode (see Advanced Setup; Retain Mode), files having this name in the same folder as this program are automatically deleted by `Parse()` (see Functions and Assets; `Parse()`). To avoid unintended file deletions, avoid initializing other file-name parameters with the same name initialized for this parameter.

3.2.6 NAME_ENGLISH

Description:

The exact text from the the raw input CSV file indicating that a specific form was filled out by a patient in English.

Data Type:

`const std::string`

Default:

"English"

Notes:

Incorrect initialization of this parameter may result in significant portions of form data being missing in the final Cardea-compatible output file.

This parameter was included for sake of user control if it ever need be modified in the future. In the meantime, it likely never needs to be changed from the default.

3.2.7 DELIMITER_CLEAN

Description:

The single character to designate as the sanitized delimiter.

Data Type:

`const char`

Default:

'$'

Notes:

This should be the singular character to forbid from being entered when patients submit forms, as it is heavily relied upon in this program for sanitation and duplicate tracking. Incorrect initialization of this parameter may result in a significantly disfigured output file that would be incompatible with Cardea, if not crash this program.

If the default character is anticipated to be present, this parameter can be modified as to represent an alternative character. Otherwise, it never needs to be changed from the default.

3.2.8 DELIMITER_CSV

Description:

The character used as the delimiter in the raw input CSV file.

Data Type:

`const char`

Default:

','

Notes:

This parameter was included for sake of user control if it ever need be modified in the future. Although most CSV files use commas as the delimiter, it is possible that the raw input CSV file may use a different delimiter, warranting this parameter to be modified. In the meantime, however, it likely never needs to be changed from the default.

3.3 Fixed Parameters

One cannot modify these parameters without altering the source code itself.

3.3.1 NAME_FORM_VAR

Description:

The single character to represent the variables in **NAME_FORM**.

Data Type:

`const char`

Default:

'#'

Notes:

Be sure to place this character within **NAME_FORM** itself.

This parameter only appears in `consent_form_file_name()` (see Functions and Assets; `consent_form_file_name()`).

3.3.2 NAME_FORM

Description:

The format of the consent form file name, adding **NAME_FORM_VAR** where a variable would be.

Data Type:

`const std::string`

Default:

"SHF-Consent_#_#_#_#-SIGNED.pdf"

Notes:

The function `consent_form_file_name()` depends on this parameter (see Functions and Assets; `consent_form_file_name()`). If different variables or order is needed, `consent_form_file_name()` would have to be rewritten to reflect those changes, as its current definition is customized to the current default of **NAME_FORM**.

3.3.3 NUM_LANG_FIELDS

Description:

The number of columns unique to a language.

Data Type:

`const int`

Default:

10

Notes:

The function `MakeCardeaCompatible()` contains a section that depends on this parameter (see Functions and Assets; `MakeCardeaCompatible()`).

3.3.4 HEADERS

Description:

The names and order of headers in the Cardea-compatible format. Each header represents the name of a column.

Data Type:

`const std::vector<std::string>`

Default:

{ “MSN”, “LastName”, “FirstName”, “Email”, “PGNam”, “PGPhone”, “Race”, “Birthdate”, “Gender”, “Weight”, “Height”, “Sport”, “OMI”, “Meds”, “ExPain”, “Sync”, “SOB”, “Murmur”, “HiBP”, “FamHist”, “SCD”, “FamDisabled”, “Consent”, “Notes” }

Notes:

The functions `MakeCardeaCompatible()` and `RemoveDuplicatesFrom()` depend on this parameter (see Functions and Assets; `MakeCardeaCompatible()`, `RemoveDuplicatesFrom()`).

---

4 Functions and Assets

Below are all the functions and global variables associated with those functions that this program uses.

Notes in this section contain helpful information to keep in mind in the event that any function or asset in the source code need be altered.

1) **Main Functions**
    1) `Parse()`
    2) `CreateLogFile()`
2) **Functions for** `Parse()`
    1) `Sanitize()`
    2) `MakeCardeaCompatible()`
    3) `RemoveDuplicatesFrom()`

3) **Assets for** `MakeCardeaCompatible()`
    1) `remove_spaces_from_this()`
    2) `file_exists()`
    3) `ten_digit_phone_number()`
    4) `consent_form_file_name()`
4) **Assets for** `RemoveDuplicatesFrom()`
    1) `track_duplicates_including_this()`
    2) `swapped()`
    3) `potential_duplicate_to_warn`
    4) `potential_duplicate_to_remove`
    5) `potential_swap`
    6) `rows_scanned_for_duplicates`
    7) `rows_scanned_for_swaps`
    8) `list_of_duplicates`
    9) `list_of_swaps`
    10) `list_of_duplicates_found`
    11) `list_of_swaps_found`
5) **Assets for** `CreateLogFile()`
    1) `add_log()`
    2) `logs`

4.1 Main Functions

4.1.1 `Parse()`

Description:

Runs the program workflow, consisting of three stages: sanitation, Cardea-compatible conversion, and duplicate action.

Serves as a container function for the parser functions responsible for each stage of this program, managing their proper inputs and, unless otherwise specified (see Advanced Setup; Retain Mode), clearing out any intermediate files.

Declaration:

```
void Parse();
```

Notes:

This function opens and closes C++ `std::ifstream` objects, passes by reference these objects to their respective parser functions, and removes intermediate files generated by the parser functions, using C++ function `std::remove()`.

4.1.2 `CreateLogFile()`

Description:

Appends significant program events to a log file with its name initialized by **NAME_LOG**.

Formats the output log file and iterates through all logs recorded in the asset `logs` to append each of them to that log file. Program mode and parameter initializations are also listed.

Declaration:

```
void CreateLogFile();
```

Notes:

Since the log-file formatting is hard coded, if any program parameter name is added or altered, this function may require alterations in the section where the program-parameter initializations are listed.

Newline characters are automatically appended to each log in the asset `logs`.

This function prints a message directly to the program interface.

4.2 Functions for `Parse()`

4.2.1 `Sanitize()`

Description:

Sanitizes input file by replacing the default delimiter, specified by **DELIMITER_CSV**, with a clean delimiter, specified by **DELIMITER_CLEAN** (see Program Parameters).

Declaration:

```
void Sanitize(std::ifstream& input);
```

Parameters:

`input`

A C++ `std::ifstream` object of the raw input CSV file, passed by reference.

Notes:

When an entry in a CSV file delimited by commas contains a comma, the contents of that entry is surrounded by quotation marks. This function makes use of that property to distinguish commas that are true delimiters from those that are just part of the entries.

The input file is unaltered. The output file is built character-by-character from the input file, replacing commas outside quotation marks with the clean delimiter and ignoring commas within quotation marks. In the end, the output file should be identical to the input file, except for the delimiter used.

4.2.2 `MakeCardeaCompatible()`

Description:

Outputs file from a sanitized input file according to Cardea-compatibility requirements.

Declaration:

```
void MakeCardeaCompatible(std::ifstream& input);
```

Parameters:

`input`

A C++ `std::ifstream` object of the intermediate sanitized file, passed by reference.

Notes:

TODO

4.2.3 `RemoveDuplicatesFrom()`

Description:

Produces output for Cardea with duplicates removed and/or with warnings of them.

Declaration:

```
void RemoveDuplicatesFrom(std::ifstream& input);
```

Parameters:

`input`

> A C++ `std::ifstream` object of the intermediate Cardea-compatible file still containing duplicates, passed by reference.

Notes:

Consider swaps to be a special case of duplicates, even though this manual, in many instances, may refer to duplicates and swaps as separate entities. Swaps are only warned of, never to remove.

This function distinguishes the "reference duplicate" from the other duplicates, and the "reference swap" from the other swaps. Below are their definitions.

Reference duplicate: the duplicate which other potential duplicates are compared to.

> Out of all rows considered to be duplicates, this program considers the latest row to be the reference duplicate. The reason is that the same patient may submit their information multiple times, resulting in duplicate rows. However, their latest submission is probably the most accurate submission and is thus designated to be the reference duplicate.
>
> The reference duplicate is the only row involved with duplicates that is not removed or warned of in the event logs, but that the other duplicates refer to in the event logs.
>
> In `list_of_duplicates_found` (see Functions and Assets; `list_of_duplicates_found`), the last row number in the container holding the row locations is designated as the row number of the reference duplicate.

Reference swap: the swap which other potential swaps are compared to.

> Out of all rows considered to be swaps, this program considers the latest row of the original row pattern to be the reference swap. The reason for the latest row is that it is possible that a swap might have duplicates too (see reasoning for defining the reference duplicate just above). The reason for the original row pattern is that the original pattern naturally serves as the reference that other rows are swapped from.
>
> The reference swap is the only row involved with swaps that is not warned of in the event logs but that the other swaps refer to in the event logs. However, this does not necessarily imply that the reference swap is the "true" row, as it is possible that the original row was the mistake, and its swapped version should be the row to retain.
>
> In `list_of_swaps_found` (see Functions and Assets; `list_of_swaps_found`), the first row number in the container holding the row locations is designated as the row number of the reference swap.

This function consists of three major components:

1) First iteration
2) Second iteration

3) Logging

4.2.3.1 First Iteration

Purpose:

Find duplicates and swaps from the tracked columns.

Procedure:

1) After ignoring the header row, build assets `potential_duplicate_to_remove`, `potential_duplicate_to_warn`, and `potential_swap` from their respectively tracked columns (see Functions and Assets; `track_duplicates_including_this()`) for the current row.

2) Find duplicates to remove.

   A C++ iterator named `find_to_remove` scans through `rows_scanned_for_duplicates` for a match to `potential_duplicate_to_remove`. If no match is found, `potential_duplicate_to_remove` is inserted into `rows_scanned_for_duplicates`. Otherwise, `potential_duplicate_to_remove` is inserted into `list_of_duplicates` and `list_of_duplicates_found`.

3) Find duplicates to warn of.

   A C++ iterator named `find_to_warn` scans through `rows_scanned_for_duplicates` for a match to `potential_duplicate_to_warn`. If no match is found, `potential_duplicate_to_warn` is inserted into `rows_scanned_for_duplicates`. Otherwise, `potential_duplicate_to_warn` is inserted into `list_of_duplicates` and `list_of_duplicates_found`.

4) Find swaps.

   A C++ iterator named `find_swap` scans through `rows_scanned_for_duplicates` for a match to `potential_swap`. If no match is found, a `swapped()` version of `potential_swap` is inserted into `rows_scanned_for_duplicates`. Otherwise, `potential_swap` is inserted into `list_of_swaps` and a `swapped()` version of `potential_swap` is inserted into `list_of_swaps_found`.

5) Clear assets `potential_duplicate_to_remove`, `potential_duplicate_to_remove`, and `potential_swap`.

   This must be done explicitly since these assets have global scope.

6) Repeat the above steps for every row onward in the input file.

   - Each row is processed using `std::getline()`
   - A counter keeping track of the current row number is incremented

7) Set up for next iteration.

   - Reset `std::ifstream` object so that `std::getline()` starts scanning from the beginning of the document again
   - Increment iteration counter

Notes:

Before the first iteration begins, an `std::ofstream` object creates the final output file **NAME_OUTPUT** and adds the header row defined by **HEADERS** to the file. Nothing more is added until during the second iteration.

The logic of this procedure results in `list_of_duplicates` containing enough matches to match to every confirmed duplicate EXCEPT for the reference duplicate itself. This is important to set up for the second iteration when the matches in `list_of_duplicates` serves as a tally for each duplicate to take action against. Essentially, one match in `list_of_duplicates` corresponds to one duplicate row; when one duplicate row is removed or warned of, its corresponding match in `list_of_duplicates` is erased. In the end, what is left untouched would be the reference duplicate, as it does not have its own corresponding match in `list_of_duplicates`. This same

logic applies to the matches in `list_of_swaps`. For information about these assets, see Functions and Assets; `list_of_duplicates`, `list_of_swaps`.

4.2.3.2 Second Iteration

Purpose:

Take action against the found duplicates and swaps.

Procedure:

1) After ignoring the header row, build assets `potential_duplicate_to_remove`, `potential_duplicate_to_warn`, and `potential_swap` from their respectively tracked columns (see Functions and Assets; `track_duplicates_including_this()`) for the current row.

2) Take action against duplicates.

   A C++ iterator named `find_to_remove` scans through `list_of_duplicates` for a match to `potential_duplicate_to_remove`. One of two scenarios plays out.

   1) No match is found, meaning the current row is NOT a duplicate to remove.

      The current row is added to the final output file, with its **DELIMITER_CLEAN** delimiters replaced by the original **DELIMITER_CSV** delimiters. This is the only point in the second iteration that anything is added to the final output file **NAME_OUTPUT**.

      A C++ iterator named `find_reference_duplicate_of_remove` scans through `list_of_duplicates_found` for a match to `potential_duplicate_to_remove`. The purpose is to determine if the current row is the reference duplicate. If a match is found, then the current row number is added to the respective `list_of_duplicates_found` container, representing the location of the reference duplicate of duplicates to remove. This should be the last row number added (since the iterator `find_to_remove` did not find a match to `potential_duplicate_to_remove` in `list_of_duplicates`).

      Although the current row is not a duplicate to remove, it could still be a duplicate to warn of. The next part takes care of this possibility.

      A C++ iterator named `find_to_warn` scans through `list_of_duplicates` for a match to `potential_duplicate_to_warn`. One of two scenarios plays out. Notice the parallel to the procedure for handling duplicates to remove.

      1) No match is found, meaning the current row is NOT a duplicate to warn of.

         A C++ iterator named `find_reference_duplicate_of_warn` scans through `list_of_duplicates_found` for a match to `potential_duplicate_to_warn`. The purpose is to determine if the current row is the reference duplicate. If a match is found, then the current row number is added to the respective `list_of_duplicates_found` container, representing the location of the reference duplicate of duplicates to warn of. This should be the last row number added (since the iterator `find_to_warn` did not find a match to `potential_duplicate_to_warn` in `list_of_duplicates`).

      2) A match is found, meaning the current row IS a duplicate to warn of.

         One instance of this match is erased from `list_of_duplicates`. A C++ iterator named `record_row` scans through `list_of_duplicates_found` for `potential_duplicate_to_warn`. The purpose is to record the location of the duplicate to warn of. The current row number is then added to the respective `list_of_duplicates_found` container.

   2) A match is found, meaning the current row IS a duplicate to remove.

      Since the current row is a duplicate to remove, this row is not added to the final output file. This omission simulates its removal.

One instance of this match is erased from `list_of_duplicates`. A C++ iterator named `record_row` scans through `list_of_duplicates_found` for `potential_duplicate_to_remove`. The purpose is to record the location of the duplicate to remove. The current row number is then added to the respective `list_of_duplicates_found` container.

Since the current row meets the criteria to be a duplicate to remove (stricter), it is expected that it also meets the criteria to be a duplicate to warn of (less strict). Thus, a match should exist for `potential_duplicate_to_warn` in `list_of_duplicates` for the current row also.

A C++ iterator named `find_to_warn` scans through `list_of_duplicates` for a match to `potential_duplicate_to_warn`. If a match is found, One instance of this match is erased from `list_of_duplicates`. However, the current row number is not added to the respective `list_of_duplicates_found` container for `potential_duplicate_to_warn` as to not simultaneously log that the current row is both a duplicate that is removed and warned of. In other words, a row determined to be a duplicate to remove is processed as a duplicate to remove, overriding the processing of being a duplicate to warn of.

3) Take action against swaps.

A C++ iterator named `find_reference_swap` scans through `list_of_swaps_found` for a match to `potential_swap`. The purpose is to determine if the current row is the reference swap. If a match is found, then the current row number is added to the respective `list_of_swaps_found` container (which is locally renamed to `record_of_row_numbers` for readability), representing the location of the reference swap which other swaps are defined from. This should ultimately be the first row number added (since if `record_of_row_numbers` were empty beforehand, then adding the current row number would make it the first row number added, but if `record_of_row_numbers` were not empty beforehand, then the current row number is made to replace the previously added reference swap location).

A C++ iterator named `find_swap` scans through `list_of_swaps` for a match to `potential_swap`. If a match is found, meaning the current row is a swap, one instance of this match is erased from `list_of_swaps`. A C++ iterator named `record_row` scans through `list_of_swaps_found` for the `swapped()` version of `potential_swap` (which is the pattern of the reference swap). The purpose is to record the location of this current row being a swap. The current row number is then added to the respective `list_of_swaps_found` container.

4) Repeat the above steps for every row onward in the input file.

- Each row is processed using `std::getline()`
- A counter keeping track of the current row number is incremented

5) Close the `std::ofstream` object for **NAME_OUTPUT**.

- Guarantees that nothing more will be added to the final output file **NAME_OUTPUT**.

Notes:

This process assumes that the criteria to remove duplicates is stricter than the criteria to warn of duplicates (more columns marked `true` for `to_remove` than for `to_warn` in `track_duplicates_including_this()`; see Functions and Assets; `track_duplicates_including_this()`), justifying having the processing of duplicates to warn of to be nested within the processing of duplicates to remove.

4.2.3.3 Logging

Purpose:

Record the events that took place to the log.

Procedure:

1) Log duplicates.

   A C++ iterator named `found` scans through every duplicate stored in `list_of_duplicates_found`. For every duplicate scanned, another C++ iterator named `row_num` scans through every row number recorded in that respective `list_of_duplicates_found` container.

   If the duplicate scanned by `found` has **DELIMITER_CLEAN** as the first character of its row string, then that duplicate was removed. It is added to the log that the currently scanned row number was removed for being a duplicate of the row number of the reference duplicate. Otherwise, if the duplicate scanned by `found` does not have **DELIMITER_CLEAN** as the first character of its row string, then that duplicate is to be given a warning. A warning is added to the log that the currently scanned row number might be a duplicate of the row number of the reference duplicate.

   `row_num` then moves on to scan for the next recorded row number, and the above step is repeated.

   Once `row_num` reaches the last row number recorded for a duplicate, no logging occurs, as this row number represents the location of the reference duplicate. Instead, `found` moves on to scan for the next stored duplicate.

2) Log swaps.

   A C++ iterator named `found` scans through every duplicate stored in `list_of_swaps_found`. For every swap scanned, another C++ iterator named `row_num` scans through every row number recorded in that respective `list_of_swaps_found` container.

   When `row_num` reaches the first row number recorded for a swap, no logging occurs, as this row number represents the location of the reference swap. Instead, `row_num` moves on to scan for the next recorded row number.

   It is added to the log a warning that the currently scanned row number might be a swapped version of the row number of the reference swap.

   `row num` then moves on to scan the next recorded row number, and the above step is repeated. After the last recorded row number is scanned and logged, `found` moves on to scan for the next stored duplicate.

   Notes:

   This logging component realizes the importance of having separate entry-delimiter orders when building `potential_duplicate_to_remove` and `potential_duplicate_to_warn` with `track_duplicates_including_this()`; that way, the appropriate event log can be made with ease from the row string alone without necessitating including any additional outside complexities just to be able to distinguish duplicates to remove from duplicates to warn of. This program has the former be built delimiter-first while the latter entry-first. For more information about these assets and the definition of a row string, see Functions and Assets; `track_duplicates_including_this()`).

4.3 Assets for `MakeCardeaCompatible()`

4.3.1 `remove_spaces_from_this()`

Description:

Removes whitespace characters from the input string.

Rebuilds character-by-character the input string from a copy of the input string, ignoring whitespace characters.

Declaration:

```
void remove_spaces_from_this(std::string& entry);
```

Parameters:

`entry`

Reference to the string to remove whitespace characters from.

Notes:

Although the string passed by reference as the input argument is modified, nothing is returned.

4.3.2 `file_exists()`

Description:

Returns whether the file from the given path exists.

Iterates through all files in a given path and checks whether the given file name exists in the given path.

Declaration:

```
bool file_exists(std::string path, std::string file);
```

Parameters:

`path`

The path to the directory containing the files of interest. This is set by the initialization of **NAME_FORM_PATH**.

`file`

The name of the file of interest. This is set by the function `consent_form_file_name()`.

Notes:

This function checks if the given path is valid, including for the default initialization of **NAME_FORM_PATH**, using C++ function `std::filesystem::exists()`. If the given path were not checked for validity, and an invalid path were passed in, the program crashes.

The given path can contain any slash direction as separators.

In the function `MakeCardeaCompatible()`, this function determines whether **FORM_YES** or **FORM_NO** is outputted.

4.3.3 `ten_digit_phone_number()`

Description:

Returns a ten-digit phone number compatible with Cardea.

Concatenates substrings of the input argument with phone-number characters.

Declaration:

```
std::string ten_digit_phone_number(std::string digits);
```

Parameters:

`digits`

The string of the ten digit number that will be made a Cardea-compatible phone number.

Notes:

This function assumes that the input consists exactly of ten numeral characters. It is recommended that the input is paired with the function `remove_spaces_from_this()` to ensure no whitespace characters are passed in.

This function may require alterations if the SHF server undergoes a formatting change.

4.3.4 `consent_form_file_name()`

Description:

Returns consent form file name based on the format of **NAME_FORM**.

Scans through **NAME_FORM**, stopping whenever the character defined by **NAME_FORM_VAR** is reached, to concatenate the part just scanned and one of the function parameters (in the order of the function arguments). These concatenated segments ultimately come together to become the final formatted consent form file name.

Declaration:

```
std::string consent_form_file_name(std::string screeningName, std::string ID, std::string
LN, std::string FN, std::string format);
```

Parameters:

`screeningName`

The name of the screening event. This is set by the initialization of **NAME_EVENT** and replaces the first **NAME_FORM_VAR** character in **NAME_FORM**.

`ID`

This is the content of the entry from the “MSN” column.

`LN`

This is the content of the entry from the “LastName” column.

`FN`

This is the content of the entry from the “FirstName” column.

`format`

The format for the consent form file name. This is set by the definition of **NAME_FORM**.

Notes:

This function will not behave correctly if the variables in **NAME_FORM** are not replaced by the **NAME_FORM_VAR** character. The **NAME_FORM_VAR** character is used as substitute for the variables in **NAME_FORM** and is explicitly set as the delimiter for the C++ function `std::getline()` in this function.

This function is customized to the current default for **NAME_FORM** (“SHF-Consent_#_#_#_#-SIGNED.pdf”). If the variables or their order changes in **NAME_FORM**, then the parameters for this function must also be changed to reflect that.

The current order of the variables is

1) initialization of **NAME_EVENT** (the `screeningName` parameter)
2) entry in “MSN” column (the `ID` parameter)
3) entry in “LastName” column (the `FN` parameter)
4) entry in “FirstName” column (the `LN` parameter)

4.4 Assets for `RemoveDuplicatesFrom()`

4.4.1 `track_duplicates_including_this()`

Description:

Defines duplicates and their strictness criteria by building specific string representations of those rows.

Builds row-wise the assets `potential_duplicate_to_warn` for entries marked `true` by the parameter `to_warn`, `potential_swap` for entries marked `true` by the parameter `for_swap`, and `potential_duplicate_to_remove` for entries marked `true` by the parameter `to_remove`.

String representation of a row: also referred to as *row string*; a special encoding of a row to be tracked for duplicates.

> Row strings serve as a convenient way to represent any particular row in this program so that it can be properly identified and processed for duplicate tracking. The assets that are designated to hold and be processed as row strings are
>
> - `potential_duplicate_to_warn`
> - `potential_swap`
> - `potential_duplicate_to_remove`
>
> A row string consists of the contents of the entries that are tracked by this function, delimited by the **DELIMITER_CLEAN** character. See the descriptions of the parameters below for how the row string of a specific asset is encoded/built.
>
> This term is unique to this program and has no special meaning outside the context of this program.

Matching entries with mismatching capitalization are still considered identical, but those with mismatching whitespace characters are not.

Declaration:

```
void track_duplicates_including_this(std::string entry, bool to_warn = true, bool for_swap
= false, bool to_remove = false);
```

Parameters:

`entry`

> Entry of a column to include in duplicate definition.

`to_warn`

> If all entries of a row marked `true` by this parameter are identical, then that row is given a warning of being a duplicate. The more columns marked `true`, the stricter the criteria (all entries of a row under these columns must match to trigger this action). Builds asset `potential_duplicate_to_warn` by separating entries *entry-first* with **DELIMITER_CLEAN**.

`for_swap`

> Entries of a column marked `true` by this parameter is tracked for swapped contents. If the first two entries of a row marked `true` with this parameter are swapped, then that row is given a

warning of being a duplicate. Builds asset `potential_swap` by separating entries *entry-first* with **DELIMITER_CLEAN**.

`to_remove`

If all entries of a row marked `true` by this parameter are identical, then that row is excluded from the output file. The more columns marked `true`, the stricter the criteria (all entries of a row under these columns must match to trigger this action). Builds asset `potential_duplicate_to_remove` by separating entries *delimiter-first* with **DELIMITER_CLEAN**.

Notes:

Consider swaps to be a special case of duplicates, even though this manual, in many instances, may refer to duplicates and swaps as separate entities.

Although this function technically marks one entry, the function `RemoveDuplicatesFrom()` iterates column-wise through every row. Thus, marking an entry with this function is equivalent to marking the whole column containing that entry.

It is advised that at some point the entry input has its whitespace characters checked since mismatching whitespace characters for matching entries are not considered identical. This program removed spaces from the input entries using `remove_spaces_from_this()` in the parser function `MakeCardeaCompatible()` (during the Cardea-compatible conversion stage).

This program tracks columns “LastName” and “FirstName” to warn of duplicates and for swaps; in addition to those columns, “PGPhone” and “Birthdate” are tracked to remove duplicates.

Although it is possible to mark more than two columns `true` for the parameter `for_swap`, only the first two columns are tracked for swapping.

Unexpected behavior may occur if the columns marked for `to_warn` are not a subset of the columns marked for `to_remove`.

By default, marking an entry without explicitly specifying the parameters of this function marks that entry for the parameter `to_warn` only.

Having the entries and delimiter orders be different between the assets `potential_duplicate_to_warn` and `potential_duplicate_to_remove` is important for proper duplicate identification for logging purposes at the end of the parser function `RemoveDuplicatesFrom()`.

The function `swapped()` depends on this function.

4.4.2 `swapped()`

Description:

Swaps the first two entries of the string representation of a row tracked for duplicates and returns that swapped version of the string. Intended for the asset `potential_swap`.

Takes entry before first delimiter and swaps it with the entry before the second delimiter in the string of the tracked row.

Declaration:

```
std::string swapped(std::string potential_duplicate);
```

Parameters:

`potential_duplicate`

The string representation of a row tracked for duplicates to have its first two entries swapped.

Notes:

This function is intended for the asset `potential_swap` and assumes that it was built entry-first.

This is certainly the most contrived function in this program based on the highly artificial nature of its definition (this is what happens when using globally defined variables!). Modify with care.

This function depends on the function `track_duplicates_including_this()`.

4.4.3 `potential_duplicate_to_warn`

Description:

The string representation of a row with its entries tracked for duplicates to warn of. Entries are separated entry-first by the delimiter **DELIMITER_CLEAN**.

These strings are compared with strings of other rows to determine if a match exists (a duplicate) and takes the corresponding action (logs a warning of the duplicate).

Data Type:

`std::string`

Notes:

This asset is built by the function `track_duplicates_including_this()` in the parser function `RemoveDuplicatesFrom()`.

The entry-delimiter order for this asset must be different from the asset `potential_duplicate_to_remove` for proper duplicate identification for logging purposes at the end of the parser function `RemoveDuplicatesFrom()`. Being entry-first indicates that a row was a duplicate to warn of.

4.4.4 `potential_duplicate_to_remove`

Description:

The string representation of a row with its entries tracked for duplicates to remove. Entries are separated delimiter-first by the delimiter **DELIMITER_CLEAN**.

These strings are compared with strings of other rows to determine if a match exists (a duplicate) and takes the corresponding action (excludes row of the earliest duplicate).

Data Type:

`std::string`

Notes:

This asset is built by the function `track_duplicates_including_this()` in the parser function `RemoveDuplicatesFrom()`.

The entry-delimiter order for this asset must be different from the asset `potential_duplicate_to_warn` for proper duplicate identification for logging purposes at the end of the parser function `RemoveDuplicatesFrom()`. Being duplicate-first indicates that a row was a duplicate to remove.

4.4.5 `potential_swap`

Description:

The string representation of a row with its entries tracked for swaps. Entries are separated entry-first by the delimiter **DELIMITER_CLEAN**.

These strings are compared with the swapped (using the function `swapped()`) version of strings of other rows to determine if a match exists (a swap) and takes the corresponding action (logs a warning of the swap).

Data Type:

`std::string`

Notes:

This asset is built by the function `track_duplicates_including_this()` in the parser function `RemoveDuplicatesFrom()`.

The entry-delimiter order for this asset must be entry-first so that it behaves properly with the function `swapped()`.

4.4.6 `rows_scanned_for_duplicates`

Description:

Holds unique string representations of rows that are to be checked against for any duplicates to remove and/or to warn of.

While this program first iterates through each row of the file, it will check if the current row matches any row already stored in this asset. If a match is NOT found, the current row is NOT a duplicate, and the string representation of the current row is then added to this asset. This way, a match will be found if a duplicate of the current row is encountered later on.

Data Type:

`std::unordered_set<std::string>`

Notes:

There are no string representations that are identical to each other in this asset.

This asset is built during the first iteration of the parser function `RemoveDuplicatesFrom()`.

This asset is never explicitly emptied.

This asset only appears in the first iteration of the parser function `RemoveDuplicatesFrom()`.

4.4.7 `rows_scanned_for_swaps`

Description:

Holds unique string representations of rows that are to be checked against for swapped entries.

While this program first iterates through each row of the file, it will check if the current row matches any row already stored in this asset. If a match is NOT found, the current row is NOT a swap, and the swapped (using the function `swapped()`) version of the string representation of the current row is then added to this asset. This way, a match will be found if a swapped version of the current row is encountered later on.

Data Type:

`std::unordered_set<std::string>`

Notes:

There are no string representations that are identical to each other in this asset.

This asset is built during the first iteration of the parser function `RemoveDuplicatesFrom()`.

This asset is never explicitly emptied.

This asset only appears in the first iteration of the parser function `RemoveDuplicatesFrom()`.

4.4.8 `list_of_duplicates`

Description:

Holds all string representations of rows that are confirmed to be duplicates to remove and/or to warn of.

While this program first iterates through each row of the file, it will check if the current row matches any row already stored in `rows_scanned_for_duplicates`. If a match is found, the current row is a duplicate, and the string representation of the current row is then added to this asset, even if there are already previous occurrences of the same row string in this asset. This way, every duplicate is matched to a row string from this asset.

Note that this asset does not include matches for the reference duplicates (see Functions and Assets; `RemoveDuplicatesFrom()`).

Data Type:

`std::unordered_multiset<std::string>`

Notes:

There can be multiple string representations that are identical to each other in this asset.

This asset is built during the first iteration of the parser function `RemoveDuplicatesFrom()`.

This asset is explicitly emptied during the second iteration of the parser function `RemoveDuplicatesFrom()`.

Although the name of this asset may suggest that it is a C++ `std::list` container, it is actually a C++ `std::unordered_multiset` container.

An arguably more space-efficient choice of container for how this asset is used could have been a C++ `std::unordered_map<std::string, int>` implementation where a count of the strings is stored instead of multiple instances of strings. However, the decision for the current implementation was made for simplicity purposes (fewer numbers for someone trying to maintain the source code to keep track of), conceptualization purposes (intuitive to match the row string to a row string than to match the row string to an integer count of row strings), and diversification purposes (a beginner to C++ can be exposed to more types of C++ containers for learning).

4.4.9 `list_of_swaps`

Description:

Holds all string representations of rows that are confirmed to have swapped entries.

While this program first iterates through each row of the file, it will check if the current row matches any row already stored in `rows_scanned_for_swaps`. If a match is found, the current row is a swap, and the string representation of the current row is then added to this asset, even if there are already previous occurrences of the same row string in this asset. This way, every swap is matched to a row string from this asset.

Note that this asset does not include matches for the reference swaps (see Functions and Assets; `RemoveDuplicatesFrom()`).

Data Type:

`std::unordered_multiset<std::string>`

Notes:

There can be multiple string representations that are identical to each other in this asset.

This asset is built during the first iteration of the parser function `RemoveDuplicatesFrom()`.

This asset is explicitly emptied during the second iteration of the parser function `RemoveDuplicatesFrom()`.

Although the name of this asset may suggest that it is a C++ `std::list` container, it is actually a C++ `std::unordered_multiset` container.

An arguably more space-efficient choice of container for how this asset is used could have been a C++ `std::unordered_map<std::string, int>` implementation where a count of the strings is stored instead of multiple instances of strings. However, the decision for the current implementation was made for simplicity purposes (fewer numbers for someone trying to maintain the source code to keep track of), conceptualization purposes (intuitive to match the row string to a row string than to match the row string to an integer count of row strings), and diversification purposes (a beginner to C++ can be exposed to more types of C++ containers for learning).

4.4.10 `list_of_duplicates_found`

Description:

Holds unique string representations of rows that are confirmed to have duplicates to remove and/or to warn of (these row strings match to the reference duplicates: see Functions and Assets; `RemoveDuplicatesFrom()`) along with the locations of each of those rows and their duplicates.

While this program first iterates through each row of the file, it will check if the current row matches any row already stored in `rows_scanned_for_duplicates`. If a match is found, the current row is a duplicate, and the string representation of the current row is then added to this asset, replacing any previous occurrences. Row locations are then recorded when this program iterates through the second time.

Data Type:

`std::unordered_map<std::string, std::vector<int>>`

Notes:

`.first` in the source code represents the unique row string. `.second` represents the container holding the row numbers.

There are no string representations that are identical to each other in this asset.

This asset is built during the first iteration of the parser function `RemoveDuplicatesFrom()`.

This asset is never explicitly emptied.

The row location of the reference duplicate is made to be the last number stored in the C++ `std::vector<int>` data structure by the parser function `RemoveDuplicatesFrom()`.

Although the name of this asset may suggest that it is a C++ `std::list` container, it is actually a C++ `std::unordered_map` container.

4.4.11 `list_of_swaps_found`

Description:

Holds unique string representations of rows that are confirmed to have swapped entries (these row strings match to the reference swaps: see Functions and Assets; `RemoveDuplicatesFrom()`) along with the locations of each of those rows and their swaps.

While this program first iterates through each row of the file, it will check if the current row matches any row already stored in `rows_scanned_for_swaps`. If a match is found, the current row is a swap, and the swapped (using the function `swapped()`) version of the string representation of the current row is then added to this asset, replacing any previous occurrences. This way, the row locations in this asset are represented by their respective reference swaps. Row locations are then recorded when this program iterates through the second time.

Data Type:

```
std::unordered_map<std::string, std::vector<int>>
```

Notes:

`.first` in the source code represents the unique row string. `.second` represents the container holding the row numbers.

There are no string representations that are identical to each other in this asset.

This asset is built during the first iteration of the parser function `RemoveDuplicatesFrom()`.

This asset is never explicitly emptied.

The row location of the reference swap is made to be the first number stored in the C++ `std::vector<int>` data structure by the parser function `RemoveDuplicatesFrom()`.

Although the name of this asset may suggest that it is a C++ `std::list` container, it is actually a C++ `std::unordered_map` container.

4.5 Assets for `CreateLogFile()`

4.5.1 `add_log()`

Description:

Returns inputted log message itself while recording the input to be appended to the log file later on.

Stores inputted log in the asset `logs`.

Declaration:

```
std::string add_log(std::string log);
```

Parameters:

`log`

Message to append to the log file.

Notes:

TIP: This function can be treated as a C++ `std::string` object in itself. For example, printing to the console directly with `std::cout` using this function allows for simultaneous printing to the console and log recording in the asset `logs`.

Although the message is copied verbatim to the asset `logs`, a line break is appended to each message by the function `CreateLogFile()` later on.

4.5.2 `logs`

Description:

Holds record of log entries throughout this program added by the function `add_log()`.

Data Type:

`std::vector<std::string>`

Notes:

This asset is built by the function `add_log()` and is accessed by the function `CreateLogFile()`.

---

5 Advanced Setup

Below are some alternative ways to run this program.

5.1 Modes

When this program is launched, a greeting is first displayed followed by a prompt to press ENTER to start this program. Instead of just pressing ENTER, one can instead enter in a specific keyword to run this program in an alternative mode. This program does not continue until a valid keyword is entered. The modes that can be run are described below. Ignore the quotation marks.

NOTE: Capitalization does not matter.

5.1.1 Express Mode

Description:

Advanced parameters are automatically initialized to their program defaults.

This mode is implicitly set by default if no keyword is entered at the beginning of this program.

Keyword:

"EXPRESS"

Notes:

Having an explicit keyword for this mode is helpful when running this program from a text file containing the pre-initialized program parameters (see Advanced Setup; Accelerated Parameter Initialization) since it allows for visible indication that the default of this program is to be run instead of just having a blank line at the top of the document.

5.1.2 Custom Mode

Description:

Gives access to initializing the advanced parameters from the program interface. The program-parameter name will also be displayed beside each initialization prompt.

Keyword:

"CUSTOM"

Notes:

See Program Parameters; Custom-Only (Advanced) Parameters to understand the implications of changing a particular advanced parameter from its default value.

5.1.3 Retain Mode

Description:

Gives access to initializing the advanced parameters from the program interface (same as Custom Mode) but also retains the intermediate files produced by this program after the sanitation and Cardea-compatible conversion stages. The program-parameter name will also be displayed beside each initialization prompt.

Keyword:

"RETAIN"

Notes:

Normally, these intermediate files are deleted automatically by the program by the function `Parse()`. This mode prevents that from happening. This mode might be useful for debugging purposes.

5.1.4 Other Keywords

Here are some Easter eggs to reward you for reading this manual. Although not modes per se, try entering any of the following keywords and see what happens!

- "HEART"
- "SHF"

5.2 Accelerated Parameter Initialization

While one can manually initialize each parameter line by line, this process can be accelerated by pre-initializing each expected parameter in a separate text document and then copy and pasting that into this program.

1) Create an empty text document.

   This can be done by opening any desired location in File Explorer, right-clicking in any empty area of that location, and selecting "New" and then the "Text Document" option. Give the text document any name, such as "parameters.txt" (".txt" might already be appended).

2) Enter into the document the desired mode to run this program in (see Advanced Setup; Modes). Press ENTER once to move to the next line.

   TIP: For readability, enter the mode keyword in all caps. That way, one can recognize the mode and distinguish it from the parameter initializations from a glance of the document.

3) Enter into the document line by line the initialization of each parameter in the order as if running this program itself.

   Press ENTER only once after each line, including after the final line. If one wishes to use the default value of a parameter (see Program Parameters), simply leave that line empty.

   TIP: For readability, type "EXIT" on the line just after the final line instead of just leaving that line blank. That way, one can recognize when this program would exit from a glance of the document.

4) If one would like to have this program exit immediately upon completion and skip seeing the event logs printed to the console altogether, add one additional empty line to the end of the document (simulates pressing ENTER to exit the program). The event logs are still appended to the log file.

5) Select all the contents of the document (Ctrl+A), copy its contents (Ctrl+C), run this program, and then simply paste the contents into the console (Ctrl+V).

This program interprets the end of each line as if ENTER were pressed and initializes each parameter with the contents of each line in the same order.

One advantage to this approach is that it allows for fast runs and reruns of this program. This is especially true if this program has to be run or rerun at a later time when one may not immediately recall how to initialize the parameters.

---

6 Troubleshooting

NOTE: Some solutions may require running the custom mode of this program to initialize more advanced parameters (see Advanced Setup; Custom Initialization).

Below are some of the issues that may arise from this program. If an issue is not addressed, report it to the SHF tech administrator. All these suggestions are assuming this program was compiled or run on a computer using Windows 10.

6.1 [ERROR] ... could not be opened or does not exist.

First, be sure the input CSV is in the same folder as this program. If so, make sure there are no typos when inputting the input CSV name (case matters). Remember that ".csv" is automatically applied to the input

name by this program, so be sure that the input name does not redundantly include ".csv" when inputting the input CSV name (input "inputForCardea" instead of "inputForCardea.csv").

If the error persists, this program might be placed in an access-restricted location. Try moving this program to another location (such as Desktop). If the error still persists, then the input CSV itself might have access restrictions. Contact the administrator if this is the case. Another possible reason is that this program itself might have been access-restricted upon installation (possibly by antivirus software). Try granting exceptions to this program.

6.2 [ERROR] Could not open sanitized input file.

Make sure no other file in the same folder as this program has the same name as whatever the parameter **NAME_INPUT_CLEAN** was initialized to.

Otherwise, this error occurred most likely due to the program being placed in an access-restricted location. Try moving this program to another location (such as Desktop). If the error persists, this program itself might have been access-restricted upon installation (possibly by antivirus software). Try granting exceptions to this program.

6.3 [ERROR] Could not open output with duplicates file.

Make sure no other file in the same folder as this program has the same name as whatever the parameter **NAME_OUTPUT_DUPLICATES** was initialized to. Also, be sure that the parameter **NAME_INPUT_CLEAN** was not initialized with the same name as **NAME_OUTPUT_DUPLICATES**.

6.4 The final Cardea-compatible output file is not appearing.

If the log file exists, check the most recent log events for any errors (most recent logs are appended to the bottom of the file). If no error is present, check that the parameter **NAME_OUTPUT** is not initialized with the same names as the parameters **NAME_INPUT_CLEAN** or **NAME_OUTPUT_DUPLICATES**.

If the log file does not exist, run this program line by line to check the log events for any errors that are printed on the console before the program exits. If no error is present, check that the parameter **NAME_OUTPUT** is not initialized with the same names as the parameters **NAME_INPUT_CLEAN** or **NAME_OUTPUT_DUPLICATES**. Also, be sure this is the case for parameter **NAME_LOG**.

If none of these solutions work, contact the administrator.

6.5 Significant portions of the final output file are blank.

If the final output file is completely blank, be sure that the raw input CSV file is also not empty or that parameter **NAME_INPUT** is initialized to the correct file name. Also, be sure that parameter **NAME_OUTPUT** is not initialized with the same name as **NAME_INPUT_CLEAN**.

If the final output file is partially blank, check that parameter **NAME_ENGLISH** is initialized properly and consistent with what the raw input CSV file requires.

6.6 The final output file is disfigured.

Be sure that the parameters **DELIMITER_CLEAN** and **DELIMITER_CSV** are initialized properly. This means making sure that the character for **DELIMITER_CLEAN** was not used anywhere in the raw input CSV file (a patient may have used and submitted it). If so, initialize **DELIMITER_CLEAN** to an alternative character not present in the raw input CSV file (see Advanced Setup; Custom Mode). Otherwise, check that the raw input CSV file is delimited by the character set by **DELIMITER_CSV** and initialize accordingly. One way to check is to open the raw input CSV file with a text editor like Notepad (can be done by right-clicking on the file and selecting "Open with" or by temporarily changing the file extension from ".csv" to ".txt" and opening that file again).

If none of these solutions work, contact the administrator, as the SHF server may have undergone a formatting change.

6.7 Path name is invalid.

If on Windows, be sure that any folders that have spaces in its name when entered in the path are not surrounded by quotation marks.

TIP: To be absolutely sure that the path entered is formatted correctly, try copy and pasting the location listed in the folder properties. This can be done by right-clicking on the folder containing the consent forms and selecting the "Properties" option. Under the "General" tab should be a listing named "Location:" followed by the path to copy and paste. If the path is long, make sure to copy and paste the whole path, as some parts may be cut off from view.

---

7 Other Resources

Be sure to have the updated contact information of the SHF tech administrator!

One may also contact Bryan Jiang by email at bryanjiang@ucla.edu.
