Documentation - fast-cpp-csv-parser

The libary provides two classes:

- `LineReader`: A class to efficiently read large files line by line.
- `CSVReader`: A class that efficiently reads large CSV files.

Note that everything is contained in the `io` namespace.

`LineReader`

```
class LineReader{
public:
  // Constructors
  LineReader(some_string_type file_name);
  LineReader(some_string_type file_name, std::FILE*file);

  // Reading
  char*next_line();

  // File Location
  void set_file_line(unsigned);
  unsigned get_file_line(unsigned)const;
  void set_file_name(some_string_type file_name);
  const char*get_truncated_file_name()const;
};
```

The constructor takes a file name and optionally a `stdio.h` file handle. If no file handle is provided the class tries to open the file and throws an `error::can_not_open_file exception` on failure. If a file handle is provided then the file name is only used to format error messages. The library will call `std::fclose` on the file handle. `some_string_type` can be a `std::string` or a `char*`.

Lines are read by calling the `next_line` function. It returns a pointer to a null terminated C-string that contains the line. If the end of file is reached a null pointer is returned. The newline character is not included in the string. You may modify the string as long as you do not write past the null terminator. The string stays valid until the destructor is called or until next_line is called again. Windows and `*`nix newlines are handled transparently. UTF-8 BOMs are automatically ignored and missing newlines at the end of the file are no problem.

Important: There is a limit of 2^24-1 characters per line. If this limit is exceeded a `error::line_length_limit_exceeded` exception is thrown.

Looping over all the lines in a file can be done in the following way.

```
LineReader in(...);
while(char*line = in.next_line()){
  ...
}
```

The remaining functions are mainly used used to format error messages. The file line indicates the current position in the file, i.e., after the first `next_line` call it is 1 and after the second 2. Before the first call it is 0. The file name is truncated as internally C-strings are used to avoid `std::bad_alloc` exceptions during error reporting.

Note: It is not possible to exchange the line termination character.

`CSVReader`

`CSVReader` uses policies. These are classes with only static members to allow core functionality to be exchanged in an efficient way.

```
template<
  unsigned column_count,
  class trim_policy = trim_chars<' ', '\t'>,
  class quote_policy = no_quote_escape<','>,
  class overflow_policy = throw_on_overflow,
  class comment_policy = no_comment
>
class CSVReader{
public:
  // Constructors
  CSVReader(some_string_type file_name);
  CSVReader(some_string_type file_name, std::FILE*file);

  // Parsing Header
  void read_header(ignore_column ignore_policy, some_string_type col_name1, some_string_type col_name2, ...);
  void set_header(some_string_type col_name1, some_string_type col_name2, ...);
  bool has_column(some_string_type col_name)const;

  // Read
  bool read_row(ColType1&col1, ColType2&col2, ...);

  // File Location
  void set_file_line(unsigned);
  unsigned get_file_line(unsigned)const;
  void set_file_name(some_string_type file_name);
  const char*get_truncated_file_name()const;
};
```

The `column_count` template parameter indicates how many columns you want to read from the CSV file. This must not necessarily coincide with the actual number of columns in the file. The three policies govern various aspects of the parsing.

The trim policy indicates what characters should be ignored at the begin and the end of every column. The default ignores spaces and tabs. This makes sure that

```
a,b,c
1,2,3
```

is interpreted in the same way as

```
  a, b,   c
1  , 2,   3
```

The trim_chars can take any number of template parameters. For example `trim_chars<' ', '\t', '_'>` is also valid. If no character should be trimmed use `trim_chars<>`.

The quote policy indicates how string should be escaped. It also specifies the column separator. The predefined policies are:

- `no_quote_escape<sep>` : Strings are not escaped. "`sep`" is used as column separator.
- `double_quote_escape<sep, quote>` : Strings are escaped using quotes. Quotes are escaped using two consecutive quotes. "`sep`" is used as column separator and "`quote`" as quoting character.

Important: Quoting can be quite expensive. Disable it if you do not need it.

The overflow policy indicates what should be done if the integers in the input are too large to fit into the variables. There following policies are predefined:

- `throw_on_overflow` : Throw an `error::integer_overflow` or `error::integer_underflow` exception.
- `ignore_overflow` : Do nothing and let the overflow happen.
- `set_to_max_on_overflow` : Set the value to `numeric_limits<...>::max()` (or to the min-pendant).

The comment policy allows to skip lines based on some criteria. Valid predefined policies are:

- `no_comment` : Do not ignore any line.
- `empty_line_comment` : Ignore all lines that are empty or only contains spaces and tabs.
- `single_line_comment<com1, com2, ...>` : Ignore all lines that start with com1 or com2 or ... as the first character. There may not be any space between the beginning of the line and the comment character.
- `single_and_empty_line_comment<com1, com2, ...>` : Ignore all empty lines and single line comments.

Examples:

- `CSVReader<4, trim_chars<' '>, double_quote_escape<',','\"'> >` reads 4 columns from a normal CSV file with

string escaping enabled.
- `CSVReader<3, trim_chars<' '>, no_quote_escape<'\t'>, single_line_comment<'#'> >` reads 3 columns from a tab separated file with string escaping disabled. Lines starting with a # are ignored.

The constructors and the file location functions are exactly the same as for `LineReader`. See its documentation for details.

There are three methods that deal with headers. The `read_header` methods reads a line from the file and rearranges the columns to match that order. It also checks whether all necessary columns are present. The `set_header` method does **not** read any input. Use it if the file does not have any header. Obviously it is impossible to rearrange columns or check for their availability when using it. The order in the file and in the program must match when using `set_header`. The `has_column` method checks whether a column is present in the file. The first argument of `read_header` is a bitfield that determines how the function should react to column mismatches. The default behavior is to throw an `error::extra_column_in_header` exception if the file contains more columns than expected and an `error::missing_column_in_header` when there are not enough. This behavior can be altered using the following flags.

- `ignore_no_column`: The default behavior, no flags are set
- `ignore_extra_column`: If a column with a name is in the file but not in the argument list, then it is silently ignored.
- `ignore_missing_column`: If a column with a name is not in the file but is in the argument list, then `read_row` will not modify the corresponding variable.

When using `ignore_column_missing` it is a good idea to initialize the variables passed to `read_row` with a default value, for example:

```
// The file only contains column "a"
CSVReader<2>in(...);
in.read_header(ignore_missing_column, "a", "b");
int a,b = 42;
while(in.read_row(a,b)){
  // a contains the value from the file
  // b is left unchanged by read_row, i.e., it is 42
}
```

If only some columns are optional or their default value depends on other columns you have to use `has_column`, for example:

```
// The file only contains the columns "a" and "b"
CSVReader<2>in(...);
in.read_header(ignore_missing_column, "a", "b", "sum");
if(!in.has_column("a") || !in.has_column("b"))
  throw my_neat_error_class();
bool has_sum = in.has_column("sum");
int a,b,sum;
while(in.read_row(a,b,sum)){
  if(!has_sum)
    sum = a+b;
}
```

Important: Do not call `has_column` from within the read-loop. It would work correctly but significantly slowdown processing.

If two columns have the same name an error::duplicated_column_in_header exception is thrown. If `read_header` is called but the file is empty a `error::header_missing` exception is thrown.

The `read_row` function reads a line, splits it into the columns and arranges them correctly. It trims the entries and unescapes them. If requested the content is interpreted as integer or as floating point. The variables passed to read_row may be of the following types.

- builtin signed integer: These are `signed char`, `short`, `int`, `long` and `long long`. The input must be encoded as a base 10 ASCII number optionally preceded by a + or -. The function detects whether the integer is too large would overflow (or underflow) and behaves as indicated by overflow_policy.
- builtin unsigned integer: Just as the signed counterparts except that a leading + or - is not allowed.
- builtin floating point: These are `float`, `double` and `long double`. The input may have a leading + or -. The number must be base 10 encoded. The decimal point may either be a dot or a comma. (Note that a comma will only work if it is not also used as column separator or the number is escaped.) A base 10 exponent may be specified using the "1e10" syntax. The "e" may be lower- or uppercase. Examples for valid floating points are "1", "-42.42" and "+123.456E789". The input is rounded to the next floating point or infinity if it is too large or small.

- `char`: The column content must be a single character.
- `std::string`: The column content is assigned to the string. The std::string is filled with the trimmed and unescaped version.
- `char*`: A pointer directly into the buffer. The string is trimmed and unescaped and null terminated. This pointer stays valid until read_row is called again or the CSVReader is destroyed. Use this for user defined types.

Note that there is no inherent overhead to using `char*` and then interpreting it compared to using one of the parsers directly build into `CSVReader`. The builtin number parsers are pure convenience. If you need a slightly different syntax then use `char*` and do the parsing yourself.