

ZIP Code Geographic Analysis

Version 1.0

Generated by Doxygen 1.16.1

1 Class Index	2
1.1 Class List	2
2 File Index	2
2.1 File List	2
3 Class Documentation	2
3.1 StateExtremes Struct Reference	2
3.1.1 Detailed Description	3
3.1.2 Constructor & Destructor Documentation	3
3.1.3 Member Data Documentation	4
3.2 ZipCodeBuffer Class Reference	6
3.2.1 Detailed Description	7
3.2.2 Constructor & Destructor Documentation	7
3.2.3 Member Function Documentation	8
3.2.4 Member Data Documentation	15
3.3 ZipCodeRecord Struct Reference	16
3.3.1 Detailed Description	17
3.3.2 Constructor & Destructor Documentation	17
3.3.3 Member Data Documentation	18
4 File Documentation	20
4.1 main.cpp File Reference	20
4.1.1 Detailed Description	20
4.1.2 USAGE	21
4.1.3 OUTPUT	21
4.1.4 Function Documentation	21
4.2 main.cpp	23
4.3 ZipCodeBuffer.cpp File Reference	26
4.3.1 Detailed Description	27
4.4 ZipCodeBuffer.cpp	27
4.5 ZipCodeBuffer.h File Reference	31
4.5.1 Detailed Description	32
4.6 ZipCodeBuffer.h	32
Index	37

1 Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

StateExtremes	
Holds the extreme ZIP codes for a single state	2
ZipCodeBuffer	
A buffer class for reading ZIP code records from CSV files	6
ZipCodeRecord	
Structure to hold a single ZIP code record	16

2 File Index

2.1 File List

Here is a list of all files with brief descriptions:

main.cpp	
Main application program for ZIP code geographic analysis	20
ZipCodeBuffer.cpp	
Implementation of the ZipCodeBuffer class	26
ZipCodeBuffer.h	
Header file for the ZipCodeBuffer class	31

3 Class Documentation

3.1 StateExtremes Struct Reference

Holds the extreme ZIP codes for a single state.

Public Member Functions

- [StateExtremes](#) ()
Constructor initializes all values to sentinel values.

Public Attributes

- int [easternmost](#)
ZIP code with minimum longitude (farthest east).
- int [westernmost](#)
ZIP code with maximum longitude (farthest west).
- int [northernmost](#)
ZIP code with maximum latitude (farthest north).
- int [southernmost](#)
ZIP code with minimum latitude (farthest south).
- double [minLongitude](#)
Minimum longitude value (easternmost point).
- double [maxLongitude](#)
Maximum longitude value (westernmost point).
- double [maxLatitude](#)
Maximum latitude value (northernmost point).
- double [minLatitude](#)
Minimum latitude value (southernmost point).

3.1.1 Detailed Description

Holds the extreme ZIP codes for a single state.

This structure stores the four extreme ZIP codes (by geographic coordinates) for a particular state. It is used to aggregate data during the analysis.

Definition at line 47 of file [main.cpp](#).

3.1.2 Constructor & Destructor Documentation

StateExtremes()

```
StateExtremes::StateExtremes () [inline]
```

Constructor initializes all values to sentinel values.

Uses extreme values so that any real coordinate will replace them during the first comparison.

Definition at line 64 of file [main.cpp](#).

References [easternmost](#), [maxLatitude](#), [maxLongitude](#), [minLatitude](#), [minLongitude](#), [northernmost](#), [southernmost](#), and [westernmost](#).

3.1.3 Member Data Documentation

easternmost

```
int StateExtremes::easternmost
```

ZIP code with minimum longitude (farthest east).

Definition at line 48 of file [main.cpp](#).

Referenced by [calculateStateExtremes\(\)](#), [printStateExtremesTable\(\)](#), and [StateExtremes\(\)](#).

maxLatitude

```
double StateExtremes::maxLatitude
```

Maximum latitude value (northernmost point).

Definition at line 55 of file [main.cpp](#).

Referenced by [calculateStateExtremes\(\)](#), and [StateExtremes\(\)](#).

maxLongitude

```
double StateExtremes::maxLongitude
```

Maximum longitude value (westernmost point).

Definition at line 54 of file [main.cpp](#).

Referenced by [calculateStateExtremes\(\)](#), and [StateExtremes\(\)](#).

minLatitude

```
double StateExtremes::minLatitude
```

Minimum latitude value (southernmost point).

Definition at line 56 of file [main.cpp](#).

Referenced by [calculateStateExtremes\(\)](#), and [StateExtremes\(\)](#).

minLongitude

```
double StateExtremes::minLongitude
```

Minimum longitude value (easternmost point).

Definition at line 53 of file [main.cpp](#).

Referenced by [calculateStateExtremes\(\)](#), and [StateExtremes\(\)](#).

northernmost

```
int StateExtremes::northernmost
```

ZIP code with maximum latitude (farthest north).

Definition at line 50 of file [main.cpp](#).

Referenced by [calculateStateExtremes\(\)](#), [printStateExtremesTable\(\)](#), and [StateExtremes\(\)](#).

southernmost

```
int StateExtremes::southernmost
```

ZIP code with minimum latitude (farthest south).

Definition at line 51 of file [main.cpp](#).

Referenced by [calculateStateExtremes\(\)](#), [printStateExtremesTable\(\)](#), and [StateExtremes\(\)](#).

westernmost

```
int StateExtremes::westernmost
```

ZIP code with maximum longitude (farthest west).

Definition at line 49 of file [main.cpp](#).

Referenced by [calculateStateExtremes\(\)](#), [printStateExtremesTable\(\)](#), and [StateExtremes\(\)](#).

The documentation for this struct was generated from the following file:

- [main.cpp](#)

3.2 ZipCodeBuffer Class Reference

A buffer class for reading ZIP code records from CSV files.

```
#include <ZipCodeBuffer.h>
```

Public Member Functions

- [ZipCodeBuffer](#) ()
Default constructor.
- [ZipCodeBuffer](#) (const string &csvFilename)
Parameterized constructor.
- [~ZipCodeBuffer](#) ()
Destructor.
- bool [open](#) (const string &csvFilename)
Opens a CSV file for reading.
- void [close](#) ()
Closes the currently open file.
- bool [isOpen](#) () const
Checks if a file is currently open.
- bool [readRecord](#) ([ZipCodeRecord](#) &record)
Reads the next ZIP code record from the file.
- vector< [ZipCodeRecord](#) > [gatherAllRecords](#) ()
Reads and gathers all records from the file.
- bool [reset](#) ()
Resets the file position to the beginning (after header).
- long [getRecordCount](#) () const
Gets the total number of records read so far.
- string [getFilename](#) () const
Gets the name of the currently open file.

Private Member Functions

- bool [parseLine](#) (const string &line, [ZipCodeRecord](#) &record)
Parses a CSV line into a [ZipCodeRecord](#).
- vector< string > [splitCSV](#) (const string &line)
Splits a CSV line into individual fields.
- string [trim](#) (const string &str)
Trims whitespace from both ends of a string.

Private Attributes

- ifstream [fileStream](#)
Input file stream for reading CSV data.
- string [filename](#)
Name of the CSV file being read.
- bool [headerSkipped](#)
Flag to track if header row has been skipped.
- long [recordCount](#)
Counter for total records read.

3.2.1 Detailed Description

A buffer class for reading ZIP code records from CSV files.

This class provides an abstraction layer for reading ZIP code data from a comma-separated values (CSV) file. It handles file I/O, parsing, and error checking while maintaining a clean interface for client code.

The class uses internal buffering to efficiently read data from the file and parse it into structured [ZipCodeRecord](#) objects.

Note

The CSV file must have a header row which is automatically skipped

Expected CSV format: ZipCode,PlaceName,State,County,Lat,Long

Definition at line 73 of file [ZipCodeBuffer.h](#).

3.2.2 Constructor & Destructor Documentation

ZipCodeBuffer() [1/2]

```
ZipCodeBuffer::ZipCodeBuffer ()
```

Default constructor.

Creates an uninitialized [ZipCodeBuffer](#). The [open\(\)](#) method must be called before reading any records.

Default constructor implementation Initializes member variables to safe default values

Definition at line 51 of file [ZipCodeBuffer.cpp](#).

References [filename](#), [headerSkipped](#), and [recordCount](#).

ZipCodeBuffer() [2/2]

```
ZipCodeBuffer::ZipCodeBuffer (  
    const string & csvFilename) [explicit]
```

Parameterized constructor.

Parameters

<code>csvFilename</code>	Path to the CSV file to open
--------------------------	------------------------------

Creates a [ZipCodeBuffer](#) and automatically opens the specified file. The header row is skipped during initialization.

Parameterized constructor implementation Opens the specified file and prepares it for reading

Definition at line 59 of file [ZipCodeBuffer.cpp](#).

References [filename](#), [headerSkipped](#), [open\(\)](#), and [recordCount](#).

~ZipCodeBuffer()

```
ZipCodeBuffer::~ZipCodeBuffer ()
```

Destructor.

Ensures the file stream is properly closed when the object is destroyed, preventing resource leaks.

Destructor implementation Ensures file is properly closed to prevent resource leaks

Definition at line 68 of file [ZipCodeBuffer.cpp](#).

References [close\(\)](#).

3.2.3 Member Function Documentation

close()

```
void ZipCodeBuffer::close ()
```

Closes the currently open file.

Closes the file stream and resets internal state. Safe to call even if no file is open.

Closes the currently open file

Resets all internal state variables and closes the file stream. Safe to call multiple times or when no file is open.

Definition at line 121 of file [ZipCodeBuffer.cpp](#).

References [filename](#), [fileStream](#), [headerSkipped](#), and [recordCount](#).

Referenced by [main\(\)](#), [open\(\)](#), and [~ZipCodeBuffer\(\)](#).

gatherAllRecords()

```
vector< ZipCodeRecord > ZipCodeBuffer::gatherAllRecords ()
```

Reads and gathers all records from the file.

Returns

Vector containing all ZIP code records from the file

Reads the entire CSV file and returns all valid records as a vector. The file position is reset to the beginning (after header) when complete.

Note

This method loads all data into memory - use with caution for very large files

Reads all records from the file into a vector

This method reads the entire file and returns all valid records. The file position is reset to the beginning after reading.

Returns

Vector containing all [ZipCodeRecord](#) objects from the file

Definition at line 183 of file [ZipCodeBuffer.cpp](#).

References [readRecord\(\)](#), and [reset\(\)](#).

Referenced by [main\(\)](#).

getFilename()

```
string ZipCodeBuffer::getFilename () const
```

Gets the name of the currently open file.

Returns

Filename string

Gets the filename of the currently open file

Returns

String containing the filename

Definition at line [245](#) of file [ZipCodeBuffer.cpp](#).

References [filename](#).

getRecordCount()

```
long ZipCodeBuffer::getRecordCount () const
```

Gets the total number of records read so far.

Returns

Count of records read

Gets the count of records read so far

Returns

Number of records successfully read

Definition at line [236](#) of file [ZipCodeBuffer.cpp](#).

References [recordCount](#).

isOpen()

```
bool ZipCodeBuffer::isOpen () const
```

Checks if a file is currently open.

Returns

true if file is open and ready for reading, false otherwise

Checks if file is open and ready for reading

Returns

true if file stream is open, false otherwise

Definition at line 135 of file [ZipCodeBuffer.cpp](#).

References [fileStream](#).

open()

```
bool ZipCodeBuffer::open (  
    const string & csvFilename)
```

Opens a CSV file for reading.

Parameters

<i>csvFilename</i>	Path to the CSV file
--------------------	----------------------

Returns

true if file opened successfully, false otherwise

Opens the specified CSV file and skips the header row. If a file is already open, it is closed first.

Opens a CSV file for reading

This method:

1. Closes any previously open file
2. Opens the new file in input mode
3. Skips the header row
4. Resets the record counter

Parameters

<i>csvFilename</i>	Path to the CSV file to open
--------------------	------------------------------

Returns

true if successful, false if file cannot be opened

Definition at line 84 of file [ZipCodeBuffer.cpp](#).

References [close\(\)](#), [filename](#), [fileStream](#), [headerSkipped](#), and [recordCount](#).

Referenced by [main\(\)](#), and [ZipCodeBuffer\(\)](#).

parseLine()

```
bool ZipCodeBuffer::parseLine (  
    const string & line,  
    ZipCodeRecord & record) [private]
```

Parses a CSV line into a [ZipCodeRecord](#).

Parameters

<i>line</i>	The CSV line to parse
<i>record</i>	Reference to ZipCodeRecord to populate

Returns

true if parsing was successful, false otherwise

This private helper method takes a raw CSV line and extracts the individual fields, converting them to appropriate data types and storing them in the provided [ZipCodeRecord](#) structure.

Handles quoted fields and embedded commas correctly.

Parses a CSV line into a [ZipCodeRecord](#)

This private helper method handles the conversion of string fields to appropriate data types and performs basic validation.

Expected CSV format: ZipCode,PlaceName,State,County,Lat,Long

Parameters

<i>line</i>	The CSV line to parse
-------------	-----------------------

<i>record</i>	Reference to ZipCodeRecord to populate
---------------	--

Returns

true if parsing succeeded, false if format is invalid

Definition at line 261 of file [ZipCodeBuffer.cpp](#).

References [ZipCodeRecord::county](#), [ZipCodeRecord::latitude](#), [ZipCodeRecord::longitude](#), [ZipCodeRecord::placeName](#), [splitCSV\(\)](#), [ZipCodeRecord::state](#), [trim\(\)](#), and [ZipCodeRecord::zipCode](#).

Referenced by [readRecord\(\)](#).

readRecord()

```
bool ZipCodeBuffer::readRecord (  
    ZipCodeRecord & record)
```

Reads the next ZIP code record from the file.

Parameters

<i>record</i>	Reference to ZipCodeRecord to populate
---------------	--

Returns

true if a record was successfully read, false on EOF or error

Reads one line from the CSV file, parses it, and populates the provided [ZipCodeRecord](#) structure. Returns false when end of file is reached or if a parsing error occurs.

Note

Automatically skips the header row on first read

Reads a single record from the CSV file

This method reads one line from the file, parses it into fields, and populates the provided [ZipCodeRecord](#) structure.

Parameters

<i>record</i>	Reference to ZipCodeRecord to be populated
---------------	--

Returns

true if a record was read successfully, false on EOF or error

Definition at line 148 of file [ZipCodeBuffer.cpp](#).

References [fileStream](#), [parseLine\(\)](#), [readRecord\(\)](#), [recordCount](#), and [trim\(\)](#).

Referenced by [gatherAllRecords\(\)](#), and [readRecord\(\)](#).

reset()

```
bool ZipCodeBuffer::reset ()
```

Resets the file position to the beginning (after header).

Returns

true if reset was successful, false otherwise

Seeks back to the start of the file and skips the header row again, allowing the file to be re-read without closing and reopening.

Resets the file position to the beginning (after header)

This allows the file to be re-read without closing and reopening.

Returns

true if reset was successful, false otherwise

Definition at line 208 of file [ZipCodeBuffer.cpp](#).

References [fileStream](#), and [recordCount](#).

Referenced by [gatherAllRecords\(\)](#).

splitCSV()

```
vector< string > ZipCodeBuffer::splitCSV (
    const string & line) [private]
```

Splits a CSV line into individual fields.

Parameters

<i>line</i>	The CSV line to split
-------------	-----------------------

Returns

Vector of strings containing individual fields

This utility function correctly handles CSV formatting including:

- Quoted fields
- Embedded commas within quotes
- Leading/trailing whitespace

Splits a CSV line into individual fields

This method correctly handles:

- Regular comma-separated fields
- Quoted fields containing commas
- Embedded quotes (escaped as "")

Algorithm:

1. Iterate through each character
2. Track whether we're inside quotes
3. Split on commas that are not inside quotes

Parameters

<i>line</i>	The CSV line to split
-------------	-----------------------

Returns

Vector of field strings

Definition at line [302](#) of file [ZipCodeBuffer.cpp](#).

Referenced by [parseLine\(\)](#).

trim()

```
string ZipCodeBuffer::trim (  
    const string & str) [private]
```

Trims whitespace from both ends of a string.

Parameters

<i>str</i>	The string to trim
------------	--------------------

Returns

Trimmed string

Trims leading and trailing whitespace from a string

This utility function removes spaces, tabs, newlines, and other whitespace characters from both ends of the string.

Parameters

<i>str</i>	The string to trim
------------	--------------------

Returns

Trimmed string

Definition at line 338 of file [ZipCodeBuffer.cpp](#).

Referenced by [parseLine\(\)](#), and [readRecord\(\)](#).

3.2.4 Member Data Documentation**filename**

```
string ZipCodeBuffer::filename [private]
```

Name of the CSV file being read.

Definition at line 76 of file [ZipCodeBuffer.h](#).

Referenced by [close\(\)](#), [getFilename\(\)](#), [open\(\)](#), [ZipCodeBuffer\(\)](#), and [ZipCodeBuffer\(\)](#).

fileStream

```
ifstream ZipCodeBuffer::fileStream [private]
```

Input file stream for reading CSV data.

Definition at line 75 of file [ZipCodeBuffer.h](#).

Referenced by [close\(\)](#), [isOpen\(\)](#), [open\(\)](#), [readRecord\(\)](#), and [reset\(\)](#).

headerSkipped

```
bool ZipCodeBuffer::headerSkipped [private]
```

Flag to track if header row has been skipped.

Definition at line 77 of file [ZipCodeBuffer.h](#).

Referenced by [close\(\)](#), [open\(\)](#), [ZipCodeBuffer\(\)](#), and [ZipCodeBuffer\(\)](#).

recordCount

```
long ZipCodeBuffer::recordCount [private]
```

Counter for total records read.

Definition at line 78 of file [ZipCodeBuffer.h](#).

Referenced by [close\(\)](#), [getRecordCount\(\)](#), [open\(\)](#), [readRecord\(\)](#), [reset\(\)](#), [ZipCodeBuffer\(\)](#), and [ZipCodeBuffer\(\)](#).

The documentation for this class was generated from the following files:

- [ZipCodeBuffer.h](#)
- [ZipCodeBuffer.cpp](#)

3.3 ZipCodeRecord Struct Reference

Structure to hold a single ZIP code record.

```
#include <ZipCodeBuffer.h>
```

Public Member Functions

- [ZipCodeRecord](#) ()
Default constructor.
- [ZipCodeRecord](#) (int zip, const string &place, const string &st, const string &cnty, double lat, double lon)
Parameterized constructor.

Public Attributes

- int [zipCode](#)
The 5-digit ZIP code.
- string [placeName](#)
Name of the place/city.
- string [state](#)
Two-letter state abbreviation.
- string [county](#)
County name.
- double [latitude](#)
Latitude coordinate (decimal degrees).
- double [longitude](#)
Longitude coordinate (decimal degrees).

3.3.1 Detailed Description

Structure to hold a single ZIP code record.

This structure represents one row from the ZIP code CSV file, containing all relevant geographic and administrative information for a specific ZIP code.

Definition at line 30 of file [ZipCodeBuffer.h](#).

3.3.2 Constructor & Destructor Documentation

ZipCodeRecord() [1/2]

```
ZipCodeRecord::ZipCodeRecord ()
```

Default constructor.

Initializes all numeric fields to zero and strings to empty.

Default constructor implementation Initializes all fields to default values

Definition at line 28 of file [ZipCodeBuffer.cpp](#).

References [county](#), [latitude](#), [longitude](#), [placeName](#), [state](#), and [zipCode](#).

ZipCodeRecord() [2/2]

```
ZipCodeRecord::ZipCodeRecord (
    int zip,
    const string & place,
    const string & st,
    const string & cnty,
    double lat,
    double lon)
```

Parameterized constructor.

Parameters

<i>zip</i>	ZIP code number
<i>place</i>	Place name
<i>st</i>	State abbreviation
<i>cnty</i>	County name
<i>lat</i>	Latitude
<i>lon</i>	Longitude

Parameterized constructor implementation Initializes all fields with provided values

Definition at line 36 of file [ZipCodeBuffer.cpp](#).

References [county](#), [latitude](#), [longitude](#), [placeName](#), [state](#), and [zipCode](#).

3.3.3 Member Data Documentation**county**

```
string ZipCodeRecord::county
```

County name.

Definition at line 34 of file [ZipCodeBuffer.h](#).

Referenced by [ZipCodeBuffer::parseLine\(\)](#), [ZipCodeRecord\(\)](#), and [ZipCodeRecord\(\)](#).

latitude

```
double ZipCodeRecord::latitude
```

Latitude coordinate (decimal degrees).

Definition at line 35 of file [ZipCodeBuffer.h](#).

Referenced by [ZipCodeBuffer::parseLine\(\)](#), [ZipCodeRecord\(\)](#), and [ZipCodeRecord\(\)](#).

longitude

```
double ZipCodeRecord::longitude
```

Longitude coordinate (decimal degrees).

Definition at line 36 of file [ZipCodeBuffer.h](#).

Referenced by [ZipCodeBuffer::parseLine\(\)](#), [ZipCodeRecord\(\)](#), and [ZipCodeRecord\(\)](#).

placeName

```
string ZipCodeRecord::placeName
```

Name of the place/city.

Definition at line 32 of file [ZipCodeBuffer.h](#).

Referenced by [ZipCodeBuffer::parseLine\(\)](#), [ZipCodeRecord\(\)](#), and [ZipCodeRecord\(\)](#).

state

```
string ZipCodeRecord::state
```

Two-letter state abbreviation.

Definition at line 33 of file [ZipCodeBuffer.h](#).

Referenced by [ZipCodeBuffer::parseLine\(\)](#), [ZipCodeRecord\(\)](#), and [ZipCodeRecord\(\)](#).

zipCode

```
int ZipCodeRecord::zipCode
```

The 5-digit ZIP code.

Definition at line 31 of file [ZipCodeBuffer.h](#).

Referenced by [ZipCodeBuffer::parseLine\(\)](#), [ZipCodeRecord\(\)](#), and [ZipCodeRecord\(\)](#).

The documentation for this struct was generated from the following files:

- [ZipCodeBuffer.h](#)
- [ZipCodeBuffer.cpp](#)

4 File Documentation

4.1 main.cpp File Reference

Main application program for ZIP code geographic analysis.

```
#include "ZipCodeBuffer.h"
#include <iostream>
#include <map>
#include <iomanip>
#include <algorithm>
#include <limits>
```

Classes

- struct [StateExtremes](#)
Holds the extreme ZIP codes for a single state.

Functions

- `map< string, StateExtremes > calculateStateExtremes` (const vector< [ZipCodeRecord](#) > &records)
Processes all ZIP code records and determines state extremes.
- void `printStateExtremesTable` (const map< string, [StateExtremes](#) > &stateMap)
Prints a formatted table of state extremes to stdout.
- int `main` (int argc, char *argv[])
Main program entry point.

4.1.1 Detailed Description

Main application program for ZIP code geographic analysis.

Author

Teagen Lee, ADD NAMES

Date

February 2026

This program analyzes ZIP code data from a CSV file and generates a report showing the extreme geographic coordinates (Easternmost, Westernmost, Northernmost, and Southernmost ZIP codes) for each state.

The program demonstrates:

- Use of the [ZipCodeBuffer](#) class for CSV parsing
- Data aggregation and analysis
- Formatted console output
- Proper resource management

4.1.2 USAGE

Usage: ./zip_analysis <csv_filename>

4.1.3 OUTPUT

The program outputs a formatted table to stdout with the following columns:

- State: Two-letter state abbreviation
- Easternmost: ZIP code with least (most negative) longitude
- Westernmost: ZIP code with greatest (most positive) longitude
- Northernmost: ZIP code with greatest (most positive) latitude
- Southernmost: ZIP code with least (most negative) latitude

Definition in file [main.cpp](#).

4.1.4 Function Documentation

calculateStateExtremes()

```
map< string, StateExtremes > calculateStateExtremes (  
    const vector< ZipCodeRecord > & records)
```

Processes all ZIP code records and determines state extremes.

Parameters

<i>records</i>	Vector of all ZIP code records
----------------	--------------------------------

Returns

Map of state abbreviations to their [StateExtremes](#) data

This function iterates through all records and maintains running extremes for each state. For each record:

- If the longitude is less than current minimum, update easternmost
- If the longitude is greater than current maximum, update westernmost
- If the latitude is greater than current maximum, update northernmost
- If the latitude is less than current minimum, update southernmost

Note

Longitude in the US is negative (west of Prime Meridian) So MINIMUM longitude = EASTERNMOST point And MAXIMUM longitude = WESTERNMOST point

Latitude is positive in Northern Hemisphere So MAXIMUM latitude = NORTHERNMOST point And MINIMUM latitude = SOUTHERNMOST point

Definition at line 92 of file [main.cpp](#).

References [StateExtremes::easternmost](#), [StateExtremes::maxLatitude](#), [StateExtremes::maxLongitude](#), [StateExtremes::minLatitude](#), [StateExtremes::minLongitude](#), [StateExtremes::northernmost](#), [StateExtremes::southernmost](#), and [StateExtremes::westernmost](#).

Referenced by [main\(\)](#).

main()

```
int main (  
    int argc,  
    char * argv[])
```

Main program entry point.

Parameters

<i>argc</i>	Number of command line arguments
<i>argv</i>	Array of command line argument strings

Returns

0 on success, non-zero on error

Program flow:

1. Validate command line arguments
2. Open CSV file using [ZipCodeBuffer](#)
3. Read all records into memory
4. Calculate extreme coordinates for each state
5. Display formatted results
6. Clean up and exit

Definition at line 213 of file [main.cpp](#).

References [calculateStateExtremes\(\)](#), [ZipCodeBuffer::close\(\)](#), [ZipCodeBuffer::gatherAllRecords\(\)](#), [ZipCodeBuffer::open\(\)](#), and [printStateExtremesTable\(\)](#).

printStateExtremesTable()

```
void printStateExtremesTable (
    const map< string, StateExtremes > & stateMap)
```

Prints a formatted table of state extremes to stdout.

Parameters

<i>stateMap</i>	Map containing StateExtremes for each state
-----------------	---

This function generates a nicely formatted table with:

- A header row with column labels
- One row per state, alphabetically sorted
- Aligned columns for readability
- ZIP codes formatted as 5-digit numbers

The output format is designed to be clear and professional, suitable for reports or further processing.

Definition at line [155](#) of file [main.cpp](#).

References [StateExtremes::easternmost](#), [StateExtremes::northernmost](#), [StateExtremes::southernmost](#), and [StateExtremes::westernmost](#)

Referenced by [main\(\)](#).

4.2 main.cpp

[Go to the documentation of this file.](#)

```
00001 /**
00002  * @file main.cpp
00003  * @brief Main application program for ZIP code geographic analysis
00004  * @author Teagen Lee, ADD NAMES
00005  * @date February 2026
00006  *
00007  * This program analyzes ZIP code data from a CSV file and generates a report
00008  * showing the extreme geographic coordinates (Easternmost, Westernmost,
00009  * Northernmost, and Southernmost ZIP codes) for each state.
00010  *
00011  * The program demonstrates:
00012  * - Use of the ZipCodeBuffer class for CSV parsing
00013  * - Data aggregation and analysis
00014  * - Formatted console output
00015  * - Proper resource management
00016  *
00017  * @section USAGE
00018  * Usage: ./zip_analysis <csv_filename>
00019  *
00020  * @section OUTPUT
00021  * The program outputs a formatted table to stdout with the following columns:
00022  * - State: Two-letter state abbreviation
00023  * - Easternmost: ZIP code with least (most negative) longitude
00024  * - Westernmost: ZIP code with greatest (most positive) longitude
00025  * - Northernmost: ZIP code with greatest (most positive) latitude
00026  * - Southernmost: ZIP code with least (most negative) latitude
00027  */
00028
00029 #include "ZipCodeBuffer.h"
```



```

00030 #include <iostream>
00031 #include <map>
00032 #include <iomanip>
00033 #include <algorithm>
00034 #include <limits>
00035
00036 #include "ZipCodeBuffer.h"
00037
00038 using namespace std;
00039
00040 /**
00041  * @struct StateExtremes
00042  * @brief Holds the extreme ZIP codes for a single state
00043  *
00044  * This structure stores the four extreme ZIP codes (by geographic coordinates)
00045  * for a particular state. It is used to aggregate data during the analysis.
00046  */
00047 struct StateExtremes {
00048     int easternmost;    ///< ZIP code with minimum longitude (farthest east)
00049     int westernmost;    ///< ZIP code with maximum longitude (farthest west)
00050     int northernmost;   ///< ZIP code with maximum latitude (farthest north)
00051     int southernmost;   ///< ZIP code with minimum latitude (farthest south)
00052
00053     double minLongitude; ///< Minimum longitude value (easternmost point)
00054     double maxLongitude; ///< Maximum longitude value (westernmost point)
00055     double maxLatitude;  ///< Maximum latitude value (northernmost point)
00056     double minLatitude;  ///< Minimum latitude value (southernmost point)
00057
00058     /**
00059      * @brief Constructor initializes all values to sentinel values
00060      *
00061      * Uses extreme values so that any real coordinate will replace them
00062      * during the first comparison.
00063      */
00064     StateExtremes()
00065         : easternmost(0), westernmost(0), northernmost(0), southernmost(0),
00066           minLongitude(numeric_limits<double>::max()),
00067           maxLongitude(numeric_limits<double>::lowest()),
00068           maxLatitude(numeric_limits<double>::lowest()),
00069           minLatitude(numeric_limits<double>::max()) {
00070     }
00071 };
00072
00073 /**
00074  * @brief Processes all ZIP code records and determines state extremes
00075  * @param records Vector of all ZIP code records
00076  * @return Map of state abbreviations to their StateExtremes data
00077  *
00078  * This function iterates through all records and maintains running
00079  * extremes for each state. For each record:
00080  * - If the longitude is less than current minimum, update easternmost
00081  * - If the longitude is greater than current maximum, update westernmost
00082  * - If the latitude is greater than current maximum, update northernmost
00083  * - If the latitude is less than current minimum, update southernmost
00084  *
00085  * @note Longitude in the US is negative (west of Prime Meridian)
00086  *       So MINIMUM longitude = EASTERNMOST point
00087  *       And MAXIMUM longitude = WESTERNMOST point
00088  * @note Latitude is positive in Northern Hemisphere
00089  *       So MAXIMUM latitude = NORTHERNMOST point
00090  *       And MINIMUM latitude = SOUTHERNMOST point
00091  */
00092 map<string, StateExtremes> calculateStateExtremes(
00093     const vector<ZipCodeRecord>& records) {
00094
00095     map<string, StateExtremes> stateMap;
00096
00097     // Iterate through all records
00098     for (const auto& record : records) {
00099         const string& state = record.state;
00100         StateExtremes& extremes = stateMap[state]; // Creates entry if doesn't exist
00101
00102         /*
00103          * Check and update EASTERNMOST (minimum longitude)
00104          * Longitude in US is negative, so more negative = farther east
00105          */
00106         if (record.longitude < extremes.minLongitude) {
00107             extremes.minLongitude = record.longitude;
00108             extremes.easternmost = record.zipCode;
00109         }
00110     }

```

```

00111     /*
00112     * Check and update WESTERNMOST (maximum longitude)
00113     * Less negative (closer to 0) = farther west
00114     */
00115     if (record.longitude > extremes.maxLongitude) {
00116         extremes.maxLongitude = record.longitude;
00117         extremes.westernmost = record.zipCode;
00118     }
00119
00120     /*
00121     * Check and update NORTHERNMOST (maximum latitude)
00122     * Higher latitude = farther north
00123     */
00124     if (record.latitude > extremes.maxLatitude) {
00125         extremes.maxLatitude = record.latitude;
00126         extremes.northernmost = record.zipCode;
00127     }
00128
00129     /*
00130     * Check and update SOUTHERNMOST (minimum latitude)
00131     * Lower latitude = farther south
00132     */
00133     if (record.latitude < extremes.minLatitude) {
00134         extremes.minLatitude = record.latitude;
00135         extremes.southernmost = record.zipCode;
00136     }
00137 }
00138
00139 return stateMap;
00140 }
00141
00142 /**
00143  * @brief Prints a formatted table of state extremes to stdout
00144  * @param stateMap Map containing StateExtremes for each state
00145  *
00146  * This function generates a nicely formatted table with:
00147  * - A header row with column labels
00148  * - One row per state, alphabetically sorted
00149  * - Aligned columns for readability
00150  * - ZIP codes formatted as 5-digit numbers
00151  *
00152  * The output format is designed to be clear and professional,
00153  * suitable for reports or further processing.
00154  */
00155 void printStateExtremesTable(const map<string, StateExtremes>& stateMap) {
00156     // Print header row with column labels
00157     cout << left; // Left-align text
00158         cout << setw(8) << "State"
00159             << setw(15) << "Easternmost"
00160             << setw(15) << "Westernmost"
00161             << setw(15) << "Northernmost"
00162             << setw(15) << "Southernmost"
00163             << endl;
00164
00165     // Print separator line for visual clarity
00166     cout << string(68, '-') << endl;
00167
00168     /*
00169     * Print data rows
00170     * The map automatically keeps states in alphabetical order by key
00171     * because map maintains sorted order
00172     */
00173     for (const auto& entry : stateMap) {
00174         const string& state = entry.first;
00175         const StateExtremes& extremes = entry.second;
00176
00177         // Print state abbreviation and ZIP codes
00178         // Format each ZIP code separately with proper padding
00179         cout << setw(8) << state;
00180
00181         // Easternmost ZIP code
00182         cout << setfill('0') << setw(5) << extremes.easternmost
00183             << setfill(' ') << setw(10) << " ";
00184
00185         // Westernmost ZIP code
00186         cout << setfill('0') << setw(5) << extremes.westernmost
00187             << setfill(' ') << setw(10) << " ";
00188
00189         // Northernmost ZIP code
00190         cout << setfill('0') << setw(5) << extremes.northernmost
00191             << setfill(' ') << setw(10) << " ";

```

```

00192
00193         // Southernmost ZIP code
00194         cout << setfill('0') << setw(5) << extremes.southernmost
00195             << setfill(' ') << endl;
00196     }
00197 }
00198
00199 /**
00200  * @brief Main program entry point
00201  * @param argc Number of command line arguments
00202  * @param argv Array of command line argument strings
00203  * @return 0 on success, non-zero on error
00204  *
00205  * Program flow:
00206  * 1. Validate command line arguments
00207  * 2. Open CSV file using ZipCodeBuffer
00208  * 3. Read all records into memory
00209  * 4. Calculate extreme coordinates for each state
00210  * 5. Display formatted results
00211  * 6. Clean up and exit
00212  */
00213 int main(int argc, char* argv[]) {
00214     // Check command line arguments
00215     if (argc != 2) {
00216         cerr << "Usage: " << argv[0] << " <csv_filename>" << endl;
00217         cerr << "Example: " << argv[0] << " us_postal_codes.csv" << endl;
00218         return 1;
00219     }
00220
00221     // Get filename from command line
00222     string filename = argv[1];
00223
00224     // Create ZipCodeBuffer object
00225     ZipCodeBuffer buffer;
00226
00227     // Attempt to open the CSV file
00228     if (!buffer.open(filename)) {
00229         cerr << "Error: Could not open file '" << filename << "'" << endl;
00230         cerr << "Please check that the file exists and is readable." << endl;
00231         return 2;
00232     }
00233
00234     cout << "Reading ZIP code data from: " << filename << endl;
00235     cout << "Processing records..." << endl << endl;
00236
00237     // Read all records from the file
00238     vector<ZipCodeRecord> allRecords = buffer.gatherAllRecords();
00239
00240     // Check if we got any records
00241     if (allRecords.empty()) {
00242         cerr << "Error: No valid records found in file." << endl;
00243         buffer.close();
00244         return 3;
00245     }
00246
00247     cout << "Total records read: " << allRecords.size() << endl << endl;
00248
00249     // Calculate state extremes
00250     map<string, StateExtremes> stateExtremes = calculateStateExtremes(allRecords);
00251
00252     cout << "Analysis Results:" << endl;
00253     cout << "======" << endl << endl;
00254
00255     // Print the results table
00256     printStateExtremesTable(stateExtremes);
00257
00258     cout << endl;
00259     cout << "Total states/territories: " << stateExtremes.size() << endl;
00260
00261     // Close the file (destructor would do this automatically, but being explicit)
00262     buffer.close();
00263
00264     return 0; // Success
00265 }

```

4.3 ZipCodeBuffer.cpp File Reference

Implementation of the [ZipCodeBuffer](#) class.

```
#include "ZipCodeBuffer.h"
#include <algorithm>
#include <cctype>
```

4.3.1 Detailed Description

Implementation of the [ZipCodeBuffer](#) class.

Author

Teagen Lee, ADD NAMES

Date

February 2026

This file contains the implementation of all methods declared in [ZipCodeBuffer.h](#). The class provides robust CSV parsing with proper error handling and memory management.

Definition in file [ZipCodeBuffer.cpp](#).

4.4 ZipCodeBuffer.cpp

[Go to the documentation of this file.](#)

```
00001 /**
00002  * @file ZipCodeBuffer.cpp
00003  * @brief Implementation of the ZipCodeBuffer class
00004  * @author Teagen Lee, ADD NAMES
00005  * @date February 2026
00006  *
00007  * This file contains the implementation of all methods declared in
00008  * ZipCodeBuffer.h. The class provides robust CSV parsing with proper
00009  * error handling and memory management.
00010  */
00011
00012 #include "ZipCodeBuffer.h"
00013 #include <algorithm>
00014 #include <cctype>
00015
00016 using namespace std;
00017
00018 /*
00019  * =====
00020  * ZipCodeRecord Implementation
00021  * =====
00022  */
00023
00024 /**
00025  * Default constructor implementation
00026  * Initializes all fields to default values
00027  */
00028 ZipCodeRecord::ZipCodeRecord()
00029     : zipCode(0), placeName(""), state(""), county(""), latitude(0.0), longitude(0.0) {
00030 }
00031
00032 /**
00033  * Parameterized constructor implementation
00034  * Initializes all fields with provided values
00035  */
00036 ZipCodeRecord::ZipCodeRecord(int zip, const string& place, const string& st,
```

```

00037         const string& cnty, double lat, double lon)
00038     : zipCode(zip), placeName(place), state(st), county(cnty), latitude(lat), longitude(lon) {
00039 }
00040
00041 /**
00042  * =====
00043  * ZipCodeBuffer Implementation
00044  * =====
00045  */
00046
00047 /**
00048  * Default constructor implementation
00049  * Initializes member variables to safe default values
00050  */
00051 ZipCodeBuffer::ZipCodeBuffer()
00052     : filename(""), headerSkipped(false), recordCount(0) {
00053 }
00054
00055 /**
00056  * Parameterized constructor implementation
00057  * Opens the specified file and prepares it for reading
00058  */
00059 ZipCodeBuffer::ZipCodeBuffer(const string& csvFilename)
00060     : filename(""), headerSkipped(false), recordCount(0) {
00061     open(csvFilename);
00062 }
00063
00064 /**
00065  * Destructor implementation
00066  * Ensures file is properly closed to prevent resource leaks
00067  */
00068 ZipCodeBuffer::~ZipCodeBuffer() {
00069     close();
00070 }
00071
00072 /**
00073  * Opens a CSV file for reading
00074  *
00075  * This method:
00076  * 1. Closes any previously open file
00077  * 2. Opens the new file in input mode
00078  * 3. Skips the header row
00079  * 4. Resets the record counter
00080  *
00081  * @param csvFilename Path to the CSV file to open
00082  * @return true if successful, false if file cannot be opened
00083  */
00084 bool ZipCodeBuffer::open(const string& csvFilename) {
00085     // Close any currently open file
00086     close();
00087
00088     // Store the filename
00089     filename = csvFilename;
00090
00091     // Open the file
00092     fileStream.open(filename);
00093
00094     // Check if file opened successfully
00095     if (!fileStream.is_open()) {
00096         return false;
00097     }
00098
00099     // Skip the header row
00100     string headerLine;
00101     if (getline(fileStream, headerLine)) {
00102         headerSkipped = true;
00103     } else {
00104         // File is empty or unreadable
00105         close();
00106         return false;
00107     }
00108
00109     // Reset record counter
00110     recordCount = 0;
00111
00112     return true;
00113 }
00114
00115 /**
00116  * Closes the currently open file
00117  *

```

```

00118 * Resets all internal state variables and closes the file stream.
00119 * Safe to call multiple times or when no file is open.
00120 */
00121 void ZipCodeBuffer::close() {
00122     if (fileStream.is_open()) {
00123         fileStream.close();
00124     }
00125     filename = "";
00126     headerSkipped = false;
00127     recordCount = 0;
00128 }
00129
00130 /**
00131 * Checks if file is open and ready for reading
00132 *
00133 * @return true if file stream is open, false otherwise
00134 */
00135 bool ZipCodeBuffer::isOpen() const {
00136     return fileStream.is_open();
00137 }
00138
00139 /**
00140 * Reads a single record from the CSV file
00141 *
00142 * This method reads one line from the file, parses it into fields,
00143 * and populates the provided ZipCodeRecord structure.
00144 *
00145 * @param record Reference to ZipCodeRecord to be populated
00146 * @return true if a record was read successfully, false on EOF or error
00147 */
00148 bool ZipCodeBuffer::readRecord(ZipCodeRecord& record) {
00149     // Check if file is open
00150     if (!fileStream.is_open()) {
00151         return false;
00152     }
00153
00154     // Read a line from the file
00155     string line;
00156     if (getline(fileStream, line)) {
00157         // Skip empty lines
00158         if (line.empty() || trim(line).empty()) {
00159             return readRecord(record); // Recursively read next non-empty line
00160         }
00161
00162         // Parse the line into a record
00163         if (parseLine(line, record)) {
00164             recordCount++;
00165             return true;
00166         } else {
00167             return false; // Parse error
00168         }
00169     }
00170
00171     // End of file reached
00172     return false;
00173 }
00174
00175 /**
00176 * Reads all records from the file into a vector
00177 *
00178 * This method reads the entire file and returns all valid records.
00179 * The file position is reset to the beginning after reading.
00180 *
00181 * @return Vector containing all ZipCodeRecord objects from the file
00182 */
00183 vector<ZipCodeRecord> ZipCodeBuffer::gatherAllRecords() {
00184     vector<ZipCodeRecord> records;
00185
00186     // Reset to beginning of file (after header)
00187     reset();
00188
00189     // Read all records
00190     ZipCodeRecord record;
00191     while (readRecord(record)) {
00192         records.push_back(record);
00193     }
00194
00195     // Reset again so file can be re-read if needed
00196     reset();
00197
00198     return records;

```

```

00199 }
00200
00201 /**
00202  * Resets the file position to the beginning (after header)
00203  *
00204  * This allows the file to be re-read without closing and reopening.
00205  *
00206  * @return true if reset was successful, false otherwise
00207  */
00208 bool ZipCodeBuffer::reset() {
00209     if (!fileStream.is_open()) {
00210         return false;
00211     }
00212
00213     // Clear any error flags
00214     fileStream.clear();
00215
00216     // Seek to beginning of file
00217     fileStream.seekg(0, ios::beg);
00218
00219     // Skip header row again
00220     string headerLine;
00221     if (!getline(fileStream, headerLine)) {
00222         return false;
00223     }
00224
00225     // Reset record counter
00226     recordCount = 0;
00227
00228     return true;
00229 }
00230
00231 /**
00232  * Gets the count of records read so far
00233  *
00234  * @return Number of records successfully read
00235  */
00236 long ZipCodeBuffer::getRecordCount() const {
00237     return recordCount;
00238 }
00239
00240 /**
00241  * Gets the filename of the currently open file
00242  *
00243  * @return String containing the filename
00244  */
00245 string ZipCodeBuffer::getFilename() const {
00246     return filename;
00247 }
00248
00249 /**
00250  * Parses a CSV line into a ZipCodeRecord
00251  *
00252  * This private helper method handles the conversion of string fields
00253  * to appropriate data types and performs basic validation.
00254  *
00255  * Expected CSV format: ZipCode,PlaceName,State,County,Lat,Long
00256  *
00257  * @param line The CSV line to parse
00258  * @param record Reference to ZipCodeRecord to populate
00259  * @return true if parsing succeeded, false if format is invalid
00260  */
00261 bool ZipCodeBuffer::parseLine(const string& line, ZipCodeRecord& record) {
00262     // Split the line into fields
00263     vector<string> fields = splitCSV(line);
00264
00265     // Verify we have the correct number of fields
00266     if (fields.size() != 6) {
00267         return false;
00268     }
00269
00270     try {
00271         // Parse each field with appropriate type conversion
00272         record.zipCode = stoi(trim(fields[0]));
00273         record.placeName = trim(fields[1]);
00274         record.state = trim(fields[2]);
00275         record.county = trim(fields[3]);
00276         record.latitude = stod(trim(fields[4]));
00277         record.longitude = stod(trim(fields[5]));
00278
00279         return true;

```

```

00280     } catch (const exception& e) {
00281         // Conversion failed - invalid data format
00282         return false;
00283     }
00284 }
00285
00286 /**
00287  * Splits a CSV line into individual fields
00288  *
00289  * This method correctly handles:
00290  * - Regular comma-separated fields
00291  * - Quoted fields containing commas
00292  * - Embedded quotes (escaped as "")
00293  *
00294  * Algorithm:
00295  * 1. Iterate through each character
00296  * 2. Track whether we're inside quotes
00297  * 3. Split on commas that are not inside quotes
00298  *
00299  * @param line The CSV line to split
00300  * @return Vector of field strings
00301  */
00302 vector<string> ZipCodeBuffer::splitCSV(const string& line) {
00303     vector<string> fields;
00304     string currentField;
00305     bool inQuotes = false;
00306
00307     for (size_t i = 0; i < line.length(); i++) {
00308         char c = line[i];
00309
00310         if (c == '"') {
00311             // Toggle quote state
00312             inQuotes = !inQuotes;
00313         } else if (c == ',' && !inQuotes) {
00314             // Field separator found (not inside quotes)
00315             fields.push_back(currentField);
00316             currentField.clear();
00317         } else {
00318             // Regular character - add to current field
00319             currentField += c;
00320         }
00321     }
00322
00323     // Add the last field
00324     fields.push_back(currentField);
00325
00326     return fields;
00327 }
00328
00329 /**
00330  * Trims leading and trailing whitespace from a string
00331  *
00332  * This utility function removes spaces, tabs, newlines, and other
00333  * whitespace characters from both ends of the string.
00334  *
00335  * @param str The string to trim
00336  * @return Trimmed string
00337  */
00338 string ZipCodeBuffer::trim(const string& str) {
00339     // Find first non-whitespace character
00340     size_t start = str.find_first_not_of(" \t\r\n");
00341     if (start == string::npos) {
00342         return ""; // String is all whitespace
00343     }
00344
00345     // Find last non-whitespace character
00346     size_t end = str.find_last_not_of(" \t\r\n");
00347
00348     // Extract substring
00349     return str.substr(start, end - start + 1);
00350 }

```

4.5 ZipCodeBuffer.h File Reference

Header file for the [ZipCodeBuffer](#) class.


```
#include <string>
#include <fstream>
#include <sstream>
#include <vector>
#include "ZipCodeBuffer.cpp"
```

Classes

- struct [ZipCodeRecord](#)
Structure to hold a single ZIP code record.
- class [ZipCodeBuffer](#)
A buffer class for reading ZIP code records from CSV files.

4.5.1 Detailed Description

Header file for the [ZipCodeBuffer](#) class.

Author

Teagen Lee, ADD NAMES

Date

February 2026

This file contains the declaration of the [ZipCodeBuffer](#) class which provides functionality to read and parse ZIP code records from a CSV file. The class implements buffered reading for efficient file I/O operations.

Definition in file [ZipCodeBuffer.h](#).

4.6 ZipCodeBuffer.h

[Go to the documentation of this file.](#)

```
00001 /**
00002  * @file ZipCodeBuffer.h
00003  * @brief Header file for the ZipCodeBuffer class
00004  * @author Teagen Lee, ADD NAMES
00005  * @date February 2026
00006  *
00007  * This file contains the declaration of the ZipCodeBuffer class which provides
00008  * functionality to read and parse ZIP code records from a CSV file. The class
00009  * implements buffered reading for efficient file I/O operations.
00010  */
00011
00012 #ifndef ZIPCODEBUFFER_H
00013 #define ZIPCODEBUFFER_H
00014
00015 #include <string>
00016 #include <fstream>
00017 #include <sstream>
00018 #include <vector>
00019
00020 using namespace std;
```

```

00021
00022 /**
00023  * @struct ZipCodeRecord
00024  * @brief Structure to hold a single ZIP code record
00025  *
00026  * This structure represents one row from the ZIP code CSV file,
00027  * containing all relevant geographic and administrative information
00028  * for a specific ZIP code.
00029  */
00030 struct ZipCodeRecord {
00031     int zipCode;          ///< The 5-digit ZIP code
00032     string placeName;     ///< Name of the place/city
00033     string state;         ///< Two-letter state abbreviation
00034     string county;        ///< County name
00035     double latitude;      ///< Latitude coordinate (decimal degrees)
00036     double longitude;     ///< Longitude coordinate (decimal degrees)
00037
00038     /**
00039      * @brief Default constructor
00040      *
00041      * Initializes all numeric fields to zero and strings to empty.
00042      */
00043     ZipCodeRecord();
00044
00045     /**
00046      * @brief Parameterized constructor
00047      * @param zip ZIP code number
00048      * @param place Place name
00049      * @param st State abbreviation
00050      * @param cnty County name
00051      * @param lat Latitude
00052      * @param lon Longitude
00053      */
00054     ZipCodeRecord(int zip, const string& place, const string& st,
00055                   const string& cnty, double lat, double lon);
00056 };
00057
00058 /**
00059  * @class ZipCodeBuffer
00060  * @brief A buffer class for reading ZIP code records from CSV files
00061  *
00062  * This class provides an abstraction layer for reading ZIP code data
00063  * from a comma-separated values (CSV) file. It handles file I/O,
00064  * parsing, and error checking while maintaining a clean interface
00065  * for client code.
00066  *
00067  * The class uses internal buffering to efficiently read data from
00068  * the file and parse it into structured ZipCodeRecord objects.
00069  *
00070  * @note The CSV file must have a header row which is automatically skipped
00071  * @note Expected CSV format: ZipCode,PlaceName,State,County,Lat,Long
00072  */
00073 class ZipCodeBuffer {
00074 private:
00075     ifstream fileStream; ///< Input file stream for reading CSV data
00076     string filename;      ///< Name of the CSV file being read
00077     bool headerSkipped;   ///< Flag to track if header row has been skipped
00078     long recordCount;     ///< Counter for total records read
00079
00080     /**
00081      * @brief Parses a CSV line into a ZipCodeRecord
00082      * @param line The CSV line to parse
00083      * @param record Reference to ZipCodeRecord to populate
00084      * @return true if parsing was successful, false otherwise
00085      *
00086      * This private helper method takes a raw CSV line and extracts
00087      * the individual fields, converting them to appropriate data types
00088      * and storing them in the provided ZipCodeRecord structure.
00089      *
00090      * Handles quoted fields and embedded commas correctly.
00091      */
00092     bool parseLine(const string& line, ZipCodeRecord& record);
00093
00094     /**
00095      * @brief Splits a CSV line into individual fields
00096      * @param line The CSV line to split
00097      * @return Vector of strings containing individual fields
00098      *
00099      * This utility function correctly handles CSV formatting including:
00100      * - Quoted fields
00101      * - Embedded commas within quotes

```

```

00102     * - Leading/trailing whitespace
00103     */
00104     vector<string> splitCSV(const string& line);
00105
00106     /**
00107     * @brief Trims whitespace from both ends of a string
00108     * @param str The string to trim
00109     * @return Trimmed string
00110     */
00111     string trim(const string& str);
00112
00113 public:
00114     /**
00115     * @brief Default constructor
00116     *
00117     * Creates an uninitialized ZipCodeBuffer. The open() method
00118     * must be called before reading any records.
00119     */
00120     ZipCodeBuffer();
00121
00122     /**
00123     * @brief Parameterized constructor
00124     * @param csvFilename Path to the CSV file to open
00125     *
00126     * Creates a ZipCodeBuffer and automatically opens the specified file.
00127     * The header row is skipped during initialization.
00128     */
00129     explicit ZipCodeBuffer(const string& csvFilename);
00130
00131     /**
00132     * @brief Destructor
00133     *
00134     * Ensures the file stream is properly closed when the object
00135     * is destroyed, preventing resource leaks.
00136     */
00137     ~ZipCodeBuffer();
00138
00139     /**
00140     * @brief Opens a CSV file for reading
00141     * @param csvFilename Path to the CSV file
00142     * @return true if file opened successfully, false otherwise
00143     *
00144     * Opens the specified CSV file and skips the header row.
00145     * If a file is already open, it is closed first.
00146     */
00147     bool open(const string& csvFilename);
00148
00149     /**
00150     * @brief Closes the currently open file
00151     *
00152     * Closes the file stream and resets internal state.
00153     * Safe to call even if no file is open.
00154     */
00155     void close();
00156
00157     /**
00158     * @brief Checks if a file is currently open
00159     * @return true if file is open and ready for reading, false otherwise
00160     */
00161     bool isOpen() const;
00162
00163     /**
00164     * @brief Reads the next ZIP code record from the file
00165     * @param record Reference to ZipCodeRecord to populate
00166     * @return true if a record was successfully read, false on EOF or error
00167     *
00168     * Reads one line from the CSV file, parses it, and populates the
00169     * provided ZipCodeRecord structure. Returns false when end of file
00170     * is reached or if a parsing error occurs.
00171     *
00172     * @note Automatically skips the header row on first read
00173     */
00174     bool readRecord(ZipCodeRecord& record);
00175
00176     /**
00177     * @brief Reads and gathers all records from the file
00178     * @return Vector containing all ZIP code records from the file
00179     *
00180     * Reads the entire CSV file and returns all valid records as a vector.
00181     * The file position is reset to the beginning (after header) when complete.
00182     */

```

```
00183     * @note This method loads all data into memory - use with caution for very large files
00184     */
00185     vector<ZipCodeRecord> gatherAllRecords();
00186
00187     /**
00188     * @brief Resets the file position to the beginning (after header)
00189     * @return true if reset was successful, false otherwise
00190     *
00191     * Seeks back to the start of the file and skips the header row again,
00192     * allowing the file to be re-read without closing and reopening.
00193     */
00194     bool reset();
00195
00196     /**
00197     * @brief Gets the total number of records read so far
00198     * @return Count of records read
00199     */
00200     long getRecordCount() const;
00201
00202     /**
00203     * @brief Gets the name of the currently open file
00204     * @return Filename string
00205     */
00206     string getFilename() const;
00207 };
00208
00209 #include "ZipCodeBuffer.cpp"
00210
00211 #endif // ZIPCODEBUFFER_H
```


Index

- ~ZipCodeBuffer
 - ZipCodeBuffer, [6](#)
- calculateStateExtremes
 - main.cpp, [20](#)
- close
 - ZipCodeBuffer, [7](#)
- county
 - ZipCodeRecord, [17](#)
- easternmost
 - StateExtremes, [3](#)
- filename
 - ZipCodeBuffer, [14](#)
- fileStream
 - ZipCodeBuffer, [14](#)
- gatherAllRecords
 - ZipCodeBuffer, [7](#)
- getFilename
 - ZipCodeBuffer, [7](#)
- getRecordCount
 - ZipCodeBuffer, [8](#)
- headerSkipped
 - ZipCodeBuffer, [15](#)
- isOpen
 - ZipCodeBuffer, [8](#)
- latitude
 - ZipCodeRecord, [17](#)
- longitude
 - ZipCodeRecord, [17](#)
- main
 - main.cpp, [21](#)
- main.cpp, [19](#)
 - calculateStateExtremes, [20](#)
 - main, [21](#)
 - printStateExtremesTable, [21](#)
- maxLatitude
 - StateExtremes, [3](#)
- maxLongitude
 - StateExtremes, [3](#)
- minLatitude
 - StateExtremes, [3](#)
- minLongitude
 - StateExtremes, [3](#)
- northernmost
 - StateExtremes, [4](#)
- open
 - ZipCodeBuffer, [9](#)
- parseLine
 - ZipCodeBuffer, [10](#)
- placeName
 - ZipCodeRecord, [18](#)
- printStateExtremesTable
 - main.cpp, [21](#)
- readRecord
 - ZipCodeBuffer, [11](#)
- recordCount
 - ZipCodeBuffer, [15](#)
- reset
 - ZipCodeBuffer, [11](#)
- southernmost
 - StateExtremes, [4](#)
- splitCSV
 - ZipCodeBuffer, [12](#)
- state
 - ZipCodeRecord, [18](#)
- StateExtremes, [1](#)
 - easternmost, [3](#)
 - maxLatitude, [3](#)
 - maxLongitude, [3](#)
 - minLatitude, [3](#)
 - minLongitude, [3](#)
 - northernmost, [4](#)
 - southernmost, [4](#)
 - StateExtremes, [2](#)
 - westernmost, [4](#)
- trim
 - ZipCodeBuffer, [13](#)
- westernmost
 - StateExtremes, [4](#)
- zipCode
 - ZipCodeRecord, [18](#)
- ZipCodeBuffer, [5](#)
 - ~ZipCodeBuffer, [6](#)
 - close, [7](#)
 - filename, [14](#)
 - fileStream, [14](#)
 - gatherAllRecords, [7](#)
 - getFilename, [7](#)
 - getRecordCount, [8](#)
 - headerSkipped, [15](#)
 - isOpen, [8](#)

- open, [9](#)
- parseLine, [10](#)
- readRecord, [11](#)
- recordCount, [15](#)
- reset, [11](#)
- splitCSV, [12](#)
- trim, [13](#)
- ZipCodeBuffer, [6](#)
- ZipCodeBuffer.cpp, [25](#)
- ZipCodeBuffer.h, [30](#)
- ZipCodeRecord, [15](#)
 - county, [17](#)
 - latitude, [17](#)
 - longitude, [17](#)
 - placeName, [18](#)
 - state, [18](#)
 - zipCode, [18](#)
- ZipCodeRecord, [16](#)