

# ZIP Code Geographic Analysis

Version 1.0

Generated by Doxygen 1.16.1

---

<b>1 Class Index</b>	<b>1</b>
1.1 Class List . . . . .	1
<b>2 File Index</b>	<b>2</b>
2.1 File List . . . . .	2
<b>3 Class Documentation</b>	<b>2</b>
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 . . . . .	3
3.2 ZipCodeBuffer Class Reference . . . . .	5
3.2.1 Detailed Description . . . . .	6
3.2.2 Constructor & Destructor Documentation . . . . .	6
3.2.3 Member Function Documentation . . . . .	7
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
<b>4 File Documentation</b>	<b>20</b>
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 . . . . .	26
4.4 ZipCodeBuffer.cpp . . . . .	27
4.5 ZipCodeBuffer.h File Reference . . . . .	31
4.5.1 Detailed Description . . . . .	31
4.5.2 ASSUMPTIONS . . . . .	32
4.6 ZipCodeBuffer.h . . . . .	32
<b>Index</b>	<b>37</b>

## 1 Class Index

### 1.1 Class List

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

<b>StateExtremes</b>	Holds the extreme ZIP codes for a single state	2
<b>ZipCodeBuffer</b>	A buffer class for reading ZIP code records from CSV files	5
<b>ZipCodeRecord</b>	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:

<b>main.cpp</b>	Main application program for ZIP code geographic analysis	20
<b>ZipCodeBuffer.cpp</b>	Implementation of the <b>ZipCodeBuffer</b> class	26
<b>ZipCodeBuffer.h</b>	Header file for the <b>ZipCodeBuffer</b> class and <b>ZipCodeRecordStruct</b>	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 **eastermost**  
*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 (eastermost 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 43 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 60 of file [main.cpp](#).

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

### 3.1.3 Member Data Documentation

#### **eastermost**

```
int StateExtremes::eastermost
```

ZIP code with minimum longitude (farthest east).

Definition at line 44 of file [main.cpp](#).

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

#### **maxLatitude**

```
double StateExtremes::maxLatitude
```

Maximum latitude value (northernmost point).

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

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

#### **maxLongitude**

```
double StateExtremes::maxLongitude
```

Maximum longitude value (westernmost point).

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

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

#### **minLatitude**

```
double StateExtremes::minLatitude
```

Minimum latitude value (southernmost point).

Definition at line 52 of file [main.cpp](#).

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

**minLongitude**

```
double StateExtremes::minLongitude
```

Minimum longitude value (easternmost point).

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

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

**northernmost**

```
int StateExtremes::northernmost
```

ZIP code with maximum latitude (farthest north).

Definition at line [46](#) of file [main.cpp](#).

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

**southernmost**

```
int StateExtremes::southernmost
```

ZIP code with minimum latitude (farthest south).

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

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

**westernmost**

```
int StateExtremes::westernmost
```

ZIP code with maximum longitude (farthest west).

Definition at line [45](#) 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 91 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 52 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 60 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 69 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 122 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 184 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 [246](#) 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 [237](#) 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 136 of file [ZipCodeBuffer.cpp](#).

References [fileStream](#).

**open()**

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

Opens a CSV file for reading.

**Parameters**

<code>csvFilename</code>	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**

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

**Returns**

true if successful, false if file cannot be opened

Definition at line 85 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**

<code>line</code>	The CSV line to parse
<code>record</code>	Reference to <a href="#">ZipCodeRecord</a> to populate

**Returns**

true if parsing was successful, false otherwise

This 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. Fails (returns false) if the CSV line does not match assumptions regarding the number, ordering, and data types of its fields.

**See also**[ZipCodeBuffer.h](#)

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 <a href="#">ZipCodeRecord</a> to populate

**Returns**

true if parsing succeeded, false if format is invalid

Definition at line 262 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 <a href="#">ZipCodeRecord</a> 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 <a href="#">ZipCodeRecord</a> to be populated
---------------	--

**Returns**

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

Definition at line 149 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 209 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 303 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 339 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 94 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 93 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 95 of file [ZipCodeBuffer.h](#).

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

**recordCount**

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

Counter for total records read.

Definition at line 96 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 49 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 29 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 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

### 3.3.3 Member Data Documentation

**county**

```
string ZipCodeRecord::county
```

County name.

Definition at line 53 of file [ZipCodeBuffer.h](#).

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

**latitude**

```
double ZipCodeRecord::latitude
```

Latitude coordinate (decimal degrees).

Definition at line 54 of file [ZipCodeBuffer.h](#).

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

**longitude**

```
double ZipCodeRecord::longitude
```

Longitude coordinate (decimal degrees).

Definition at line 55 of file [ZipCodeBuffer.h](#).

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

**placeName**

```
string ZipCodeRecord::placeName
```

Name of the place/city.

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

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

**state**

```
string ZipCodeRecord::state
```

Two-letter state abbreviation.

Definition at line 52 of file [ZipCodeBuffer.h](#).

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

**zipCode**

```
int ZipCodeRecord::zipCode
```

The 5-digit ZIP code.

Definition at line 50 of file [ZipCodeBuffer.h](#).

Referenced by [ZipCodeBuffer::parseLine\(\)](#), 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 <limits>
```

#### Classes

- struct [StateExtremes](#)

*Holds the extreme ZIP codes for a single state.*

#### Functions

- static bool [smallerZipWins](#) (int candidate, int current)  
*Tie-break helper: choose smaller ZIP if coordinate value ties.*
- 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 (primary contributor)

Dristi Barnwal, Ethan Jackson, Marcus Julius, Natoli Mayu (reviewers)

##### 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.

### 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

#### Note

Some records can have identical latitude/longitude. To ensure output is identical regardless of CSV row ordering, ties are broken deterministically by choosing the smaller ZIP code.

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

<code>records</code>	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

If multiple records in the same state tie for an extreme coordinate value, the record with smallest ZIP is chosen.

Definition at line 96 of file [main.cpp](#).

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

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

**main()**

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

Main program entry point.

Definition at line 195 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**

<code>stateMap</code>	Map containing <a href="#">StateExtremes</a> for each state
-----------------------	---

This function generates a formatted table with:

- A header row with column labels
- One row per state, alphabetically sorted
- ZIP codes formatted as 5-digit numbers (leading zeros preserved in output)

Definition at line 160 of file [main.cpp](#).

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

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

**smallerZipWins()**

```
bool smallerZipWins (
    int candidate,
    int current) [static]
```

Tie-break helper: choose smaller ZIP if coordinate value ties.

**Parameters**

<i>candidate</i>	Candidate ZIP code
<i>current</i>	Current chosen ZIP code

**Returns**

true if candidate should replace current when tied on coordinate

Definition at line 75 of file [main.cpp](#).

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

**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 (primary contributor)
00005 * @author Dristi Barnwal, Ethan Jackson, Marcus Julius, Natoli Mayu (reviewers)
00006 * @date February 2026
00007 *
00008 * This program analyzes ZIP code data from a CSV file and generates a report
00009 * showing the extreme geographic coordinates (Easternmost, Westernmost,
00010 * Northernmost, and Southernmost ZIP codes) for each state.
00011 *
00012 * @section USAGE
00013 * Usage: ./zip_analysis <csv_filename>
00014 *
00015 * @section OUTPUT
00016 * The program outputs a formatted table to stdout with the following columns:
00017 * - State: Two-letter state abbreviation
00018 * - Easternmost: ZIP code with least (most negative) longitude
00019 * - Westernmost: ZIP code with greatest (most positive) longitude
00020 * - Northernmost: ZIP code with greatest (most positive) latitude
00021 * - Southernmost: ZIP code with least (most negative) latitude
00022 *
00023 * @note Some records can have identical latitude/longitude. To ensure output is
00024 * identical regardless of CSV row ordering, ties are broken deterministically
00025 * by choosing the smaller ZIP code.
00026 */
00027
00028 #include "ZipCodeBuffer.h"
00029 #include <iostream>
00030 #include <map>
00031 #include <iomanip>
00032 #include <limits>
00033
00034 using namespace std;
00035
00036 /**
00037 * @struct StateExtremes
00038 * @brief Holds the extreme ZIP codes for a single state
```

```

00039  *
00040  * This structure stores the four extreme ZIP codes (by geographic coordinates)
00041  * for a particular state. It is used to aggregate data during the analysis.
00042  */
00043 struct StateExtremes {
00044     int easternmost;      ///< ZIP code with minimum longitude (farthest east)
00045     int westernmost;      ///< ZIP code with maximum longitude (farthest west)
00046     int northernmost;    ///< ZIP code with maximum latitude (farthest north)
00047     int southernmost;    ///< ZIP code with minimum latitude (farthest south)
00048
00049     double minLongitude; // Minimum longitude value (easternmost point)
00050     double maxLongitude; // Maximum longitude value (westernmost point)
00051     double maxLatitude; // Maximum latitude value (northernmost point)
00052     double minLatitude; // Minimum latitude value (southernmost point)
00053
00054 /**
00055  * @brief Constructor initializes all values to sentinel values
00056  *
00057  * Uses extreme values so that any real coordinate will replace them
00058  * during the first comparison.
00059  */
00060 StateExtremes()
00061     : easternmost(0), westernmost(0), northernmost(0), southernmost(0),
00062         minLongitude(numeric_limits<double>::max()),
00063         maxLongitude(numeric_limits<double>::lowest()),
00064         maxLatitude(numeric_limits<double>::lowest()),
00065         minLatitude(numeric_limits<double>::max()) {
00066 }
00067 };
00068
00069 /**
00070  * @brief Tie-break helper: choose smaller ZIP if coordinate value ties
00071  * @param candidate Candidate ZIP code
00072  * @param current Current chosen ZIP code
00073  * @return true if candidate should replace current when tied on coordinate
00074 */
00075 static bool smallerZipWins(int candidate, int current) {
00076     // If current is 0 (uninitialized), candidate should win
00077     if (current == 0) return true;
00078     return candidate < current;
00079 }
00080
00081 /**
00082  * @brief Processes all ZIP code records and determines state extremes
00083  * @param records Vector of all ZIP code records
00084  * @return Map of state abbreviations to their StateExtremes data
00085  *
00086  * This function iterates through all records and maintains running
00087  * extremes for each state. For each record:
00088  * - If the longitude is less than current minimum, update easternmost
00089  * - If the longitude is greater than current maximum, update westernmost
00090  * - If the latitude is greater than current maximum, update northernmost
00091  * - If the latitude is less than current minimum, update southernmost
00092  *
00093  * If multiple records in the same state tie for an extreme coordinate value,
00094  * the record with smallest ZIP is chosen.
00095 */
00096 map<string, StateExtremes> calculateStateExtremes(const vector<ZipCodeRecord>& records) {
00097     map<string, StateExtremes> stateMap;
00098
00099     for (const auto& record : records) {
00100         const string& state = record.state;
00101         StateExtremes& extremes = stateMap[state]; // creates entry if not exist
00102
00103         // EASTERNMOST (minimum longitude)
00104         if (record.longitude < extremes.minLongitude) {
00105             extremes.minLongitude = record.longitude;
00106             extremes.easternmost = record.zipCode;
00107         } else if (record.longitude == extremes.minLongitude) {
00108             // Tie on longitude -> choose smaller ZIP deterministically
00109             if (smallerZipWins(record.zipCode, extremes.easternmost)) {
00110                 extremes.easternmost = record.zipCode;
00111             }
00112         }
00113
00114         // WESTERNMOST (maximum longitude)
00115         if (record.longitude > extremes.maxLongitude) {
00116             extremes.maxLongitude = record.longitude;
00117             extremes.westernmost = record.zipCode;
00118         } else if (record.longitude == extremes.maxLongitude) {
00119             // Tie on longitude -> choose smaller ZIP deterministically

```

```

00120         if (smallerZipWins(record.zipCode, extremes.westernmost)) {
00121             extremes.westernmost = record.zipCode;
00122         }
00123     }
00124
00125     // NORTHERNMOST (maximum latitude)
00126     if (record.latitude > extremes.maxLatitude) {
00127         extremes.maxLatitude = record.latitude;
00128         extremes.northernmost = record.zipCode;
00129     } else if (record.latitude == extremes.maxLatitude) {
00130         // Tie on latitude -> choose smaller ZIP deterministically
00131         if (smallerZipWins(record.zipCode, extremes.northernmost)) {
00132             extremes.northernmost = record.zipCode;
00133         }
00134     }
00135
00136     // SOUTHERNMOST (minimum latitude)
00137     if (record.latitude < extremes.minLatitude) {
00138         extremes.minLatitude = record.latitude;
00139         extremes.southernmost = record.zipCode;
00140     } else if (record.latitude == extremes.minLatitude) {
00141         // Tie on latitude -> choose smaller ZIP deterministically
00142         if (smallerZipWins(record.zipCode, extremes.southernmost)) {
00143             extremes.southernmost = record.zipCode;
00144         }
00145     }
00146 }
00147
00148 return stateMap;
00149 }
00150
00151 /**
00152 * @brief Prints a formatted table of state extremes to stdout
00153 * @param stateMap Map containing StateExtremes for each state
00154 *
00155 * This function generates a formatted table with:
00156 * - A header row with column labels
00157 * - One row per state, alphabetically sorted
00158 * - ZIP codes formatted as 5-digit numbers (leading zeros preserved in output)
00159 */
00160 void printStateExtremesTable(const map<string, StateExtremes>& stateMap) {
00161     cout << left;
00162     cout << setw(8) << "State"
00163         << setw(15) << "Easternmost"
00164         << setw(15) << "Westernmost"
00165         << setw(15) << "Northernmost"
00166         << setw(15) << "Southernmost"
00167         << endl;
00168
00169     cout << string(68, '-') << endl;
00170
00171     for (const auto& entry : stateMap) {
00172         const string& state = entry.first;
00173         const StateExtremes& extremes = entry.second;
00174
00175         cout << setw(8) << state;
00176
00177         // Print ZIPs as 5 digits (leading zeros)
00178         cout << setfill('0') << setw(5) << extremes.easternmost
00179             << setfill(' ') << setw(10) << " ";
00180
00181         cout << setfill('0') << setw(5) << extremes.westernmost
00182             << setfill(' ') << setw(10) << " ";
00183
00184         cout << setfill('0') << setw(5) << extremes.northernmost
00185             << setfill(' ') << setw(10) << " ";
00186
00187         cout << setfill('0') << setw(5) << extremes.southernmost
00188             << setfill(' ') << endl;
00189     }
00190 }
00191
00192 /**
00193 * @brief Main program entry point
00194 */
00195 int main(int argc, char* argv[]) {
00196     if (argc != 2) {
00197         cerr << "Usage: " << argv[0] << " <csv_filename>" << endl;
00198         cerr << "Example: " << argv[0] << " us_postal_codes.csv" << endl;
00199         return 1;
00200     }

```

```
00201     string filename = argv[1];
00202
00203     ZipCodeBuffer buffer;
00204     if (!buffer.open(filename)) {
00205         cerr << "Error: Could not open file '" << filename << "'" << endl;
00206         cerr << "Please check that the file exists and is readable." << endl;
00207         return 2;
00208     }
00209
00210     cout << "Reading ZIP code data from: " << filename << endl;
00211     cout << "Processing records..." << endl << endl;
00212
00213     vector<ZipCodeRecord> allRecords = buffer.gatherAllRecords();
00214
00215     if (allRecords.empty()) {
00216         cerr << "Error: No valid records found in file." << endl;
00217         buffer.close();
00218         return 3;
00219     }
00220
00221     cout << "Total records read: " << allRecords.size() << endl << endl;
00222
00223     map<string, StateExtremes> stateExtremes = calculateStateExtremes(allRecords);
00224
00225     cout << "Analysis Results:" << endl;
00226     cout << "======" << endl << endl;
00227
00228     printStateExtremesTable(stateExtremes);
00229
00230     cout << endl;
00231     cout << "Total states/territories: " << stateExtremes.size() << endl;
00232
00233     buffer.close();
00234
00235     return 0;
00236 }
```

## 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 (primary contributor)

Dristi Barnwal, Ethan Jackson, Marcus Julius, Natoli Mayu (reviewers)

#### 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 (primary contributor)
00005 * @author Dristi Barnwal, Ethan Jackson, Marcus Julius, Natoli Mayu (reviewers)
00006 * @date February 2026
00007 *
00008 * This file contains the implementation of all methods declared in
00009 * ZipCodeBuffer.h. The class provides robust CSV parsing with proper
00010 * error handling and memory management.
00011 */
00012
00013 #include "ZipCodeBuffer.h"
00014 #include <algorithm>
00015 #include <cctype>
00016
00017 using namespace std;
00018
00019 /**
00020 * =====
00021 * ZipCodeRecord Implementation
00022 * =====
00023 */
00024
00025 /**
00026 * Default constructor implementation
00027 * Initializes all fields to default values
00028 */
00029 ZipCodeRecord::ZipCodeRecord()
00030     : zipCode(0), placeName(""), state(""), county(""), latitude(0.0), longitude(0.0) {
00031 }
00032
00033 /**
00034 * Parameterized constructor implementation
00035 * Initializes all fields with provided values
00036 */
00037 ZipCodeRecord::ZipCodeRecord(int zip, const string& place, const string& st,
00038                             const string& cnty, double lat, double lon)
00039     : zipCode(zip), placeName(place), state(st), county(cnty), latitude(lat), longitude(lon) {
00040 }
00041
00042 /**
00043 * =====
00044 * ZipCodeBuffer Implementation
00045 * =====
00046 */
00047
00048 /**
00049 * Default constructor implementation
00050 * Initializes member variables to safe default values
00051 */
00052 ZipCodeBuffer::ZipCodeBuffer()
00053     : filename(""), headerSkipped(false), recordCount(0) {
00054 }
00055
00056 /**
00057 * Parameterized constructor implementation
00058 * Opens the specified file and prepares it for reading
00059 */
00060 ZipCodeBuffer::ZipCodeBuffer(const string& csvFilename)
00061     : filename(""), headerSkipped(false), recordCount(0) {
00062     open(csvFilename);
00063 }
00064
00065 /**
00066 * Destructor implementation
00067 * Ensures file is properly closed to prevent resource leaks
00068 */
00069 ZipCodeBuffer::~ZipCodeBuffer() {
00070     close();
00071 }
00072
00073 /**
00074 * Opens a CSV file for reading
00075 *
00076 * This method:
```

```

00077 * 1. Closes any previously open file
00078 * 2. Opens the new file in input mode
00079 * 3. Skips the header row
00080 * 4. Resets the record counter
00081 *
00082 * @param csvFilename Path to the CSV file to open
00083 * @return true if successful, false if file cannot be opened
00084 */
00085 bool ZipCodeBuffer::open(const string& csvFilename) {
00086     // Close any currently open file
00087     close();
00088
00089     // Store the filename
00090     filename = csvFilename;
00091
00092     // Open the file
00093     fileStream.open(filename);
00094
00095     // Check if file opened successfully
00096     if (!fileStream.is_open()) {
00097         return false;
00098     }
00099
00100    // Skip the header row
00101    string headerLine;
00102    if (getline(fileStream, headerLine)) {
00103        headerSkipped = true;
00104    } else {
00105        // File is empty or unreadable
00106        close();
00107        return false;
00108    }
00109
00110    // Reset record counter
00111    recordCount = 0;
00112
00113    return true;
00114 }
00115
00116 /**
00117 * Closes the currently open file
00118 *
00119 * Resets all internal state variables and closes the file stream.
00120 * Safe to call multiple times or when no file is open.
00121 */
00122 void ZipCodeBuffer::close() {
00123     if (fileStream.is_open()) {
00124         fileStream.close();
00125     }
00126     filename = "";
00127     headerSkipped = false;
00128     recordCount = 0;
00129 }
00130
00131 /**
00132 * Checks if file is open and ready for reading
00133 *
00134 * @return true if file stream is open, false otherwise
00135 */
00136 bool ZipCodeBuffer::isOpen() const {
00137     return fileStream.is_open();
00138 }
00139
00140 /**
00141 * Reads a single record from the CSV file
00142 *
00143 * This method reads one line from the file, parses it into fields,
00144 * and populates the provided ZipCodeRecord structure.
00145 *
00146 * @param record Reference to ZipCodeRecord to be populated
00147 * @return true if a record was read successfully, false on EOF or error
00148 */
00149 bool ZipCodeBuffer::readRecord(ZipCodeRecord& record) {
00150     // Check if file is open
00151     if (!fileStream.is_open()) {
00152         return false;
00153     }
00154
00155     // Read a line from the file
00156     string line;
00157     if (getline(fileStream, line)) {

```

```
00158     // Skip empty lines
00159     if (line.empty() || trim(line).empty()) {
00160         return readRecord(record); // Recursively read next non-empty line
00161     }
00162
00163     // Parse the line into a record
00164     if (parseLine(line, record)) {
00165         recordCount++;
00166         return true;
00167     } else {
00168         return false; // Parse error
00169     }
00170 }
00171
00172 // End of file reached
00173 return false;
00174 }
00175
00176 /**
00177 * Reads all records from the file into a vector
00178 *
00179 * This method reads the entire file and returns all valid records.
00180 * The file position is reset to the beginning after reading.
00181 *
00182 * @return Vector containing all ZipCodeRecord objects from the file
00183 */
00184 vector<ZipCodeRecord> ZipCodeBuffer::gatherAllRecords() {
00185     vector<ZipCodeRecord> records;
00186
00187     // Reset to beginning of file (after header)
00188     reset();
00189
00190     // Read all records
00191     ZipCodeRecord record;
00192     while (readRecord(record)) {
00193         records.push_back(record);
00194     }
00195
00196     // Reset again so file can be re-read if needed
00197     reset();
00198
00199     return records;
00200 }
00201
00202 /**
00203 * Resets the file position to the beginning (after header)
00204 *
00205 * This allows the file to be re-read without closing and reopening.
00206 *
00207 * @return true if reset was successful, false otherwise
00208 */
00209 bool ZipCodeBuffer::reset() {
00210     if (!fileStream.is_open()) {
00211         return false;
00212     }
00213
00214     // Clear any error flags
00215     fileStream.clear();
00216
00217     // Seek to beginning of file
00218     fileStream.seekg(0, ios::beg);
00219
00220     // Skip header row again
00221     string headerLine;
00222     if (!getline(fileStream, headerLine)) {
00223         return false;
00224     }
00225
00226     // Reset record counter
00227     recordCount = 0;
00228
00229     return true;
00230 }
00231
00232 /**
00233 * Gets the count of records read so far
00234 *
00235 * @return Number of records successfully read
00236 */
00237 long ZipCodeBuffer::getRecordCount() const {
00238     return recordCount;
```

```

00239 }
00240
00241 /**
00242 * Gets the filename of the currently open file
00243 *
00244 * @return String containing the filename
00245 */
00246 string ZipCodeBuffer::getFilename() const {
00247     return filename;
00248 }
00249
00250 /**
00251 * Parses a CSV line into a ZipCodeRecord
00252 *
00253 * This private helper method handles the conversion of string fields
00254 * to appropriate data types and performs basic validation.
00255 *
00256 * Expected CSV format: ZipCode,PlaceName,State,County,Lat,Long
00257 *
00258 * @param line The CSV line to parse
00259 * @param record Reference to ZipCodeRecord to populate
00260 * @return true if parsing succeeded, false if format is invalid
00261 */
00262 bool ZipCodeBuffer::parseLine(const string& line, ZipCodeRecord& record) {
00263     // Split the line into fields
00264     vector<string> fields = splitCSV(line);
00265
00266     // Verify we have the correct number of fields
00267     if (fields.size() != 6) {
00268         return false;
00269     }
00270
00271     try {
00272         // Parse each field with appropriate type conversion
00273         record.zipCode = stoi(trim(fields[0]));
00274         record.placeName = trim(fields[1]);
00275         record.state = trim(fields[2]);
00276         record.county = trim(fields[3]);
00277         record.latitude = stod(trim(fields[4]));
00278         record.longitude = stod(trim(fields[5]));
00279
00280         return true;
00281     } catch (const exception& e) {
00282         // Conversion failed - invalid data format
00283         return false;
00284     }
00285 }
00286
00287 /**
00288 * Splits a CSV line into individual fields
00289 *
00290 * This method correctly handles:
00291 * - Regular comma-separated fields
00292 * - Quoted fields containing commas
00293 * - Embedded quotes (escaped as "")
00294 *
00295 * Algorithm:
00296 * 1. Iterate through each character
00297 * 2. Track whether we're inside quotes
00298 * 3. Split on commas that are not inside quotes
00299 *
00300 * @param line The CSV line to split
00301 * @return Vector of field strings
00302 */
00303 vector<string> ZipCodeBuffer::splitCSV(const string& line) {
00304     vector<string> fields;
00305     string currentField;
00306     bool inQuotes = false;
00307
00308     for (size_t i = 0; i < line.length(); i++) {
00309         char c = line[i];
00310
00311         if (c == '\"') {
00312             // Toggle quote state
00313             inQuotes = !inQuotes;
00314         } else if (c == ',' && !inQuotes) {
00315             // Field separator found (not inside quotes)
00316             fields.push_back(currentField);
00317             currentField.clear();
00318         } else {
00319             // Regular character - add to current field

```

```
00320         currentField += c;
00321     }
00322 }
00323
00324 // Add the last field
00325 fields.push_back(currentField);
00326
00327 return fields;
00328 }
00329
00330 /**
00331 * Trims leading and trailing whitespace from a string
00332 *
00333 * This utility function removes spaces, tabs, newlines, and other
00334 * whitespace characters from both ends of the string.
00335 *
00336 * @param str The string to trim
00337 * @return Trimmed string
00338 */
00339 string ZipCodeBuffer::trim(const string& str) {
00340     // Find first non-whitespace character
00341     size_t start = str.find_first_not_of(" \t\r\n");
00342     if (start == string::npos) {
00343         return ""; // String is all whitespace
00344     }
00345
00346     // Find last non-whitespace character
00347     size_t end = str.find_last_not_of(" \t\r\n");
00348
00349     // Extract substring
00350     return str.substr(start, end - start + 1);
00351 }
```

## 4.5 ZipCodeBuffer.h File Reference

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

```
#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 and [ZipCodeRecordStruct](#).

**Author**

Teagen Lee (primary contributor)  
Ethan Jackson (additional comments and formatting)  
Dristi Barnwal, Marcus Julius, Natoli Mayu (reviewers)

**Date**

February 2026

This file contains the declarations of [ZipCodeBuffer](#), a class which provides functionality to read and parse ZIP code records from a CSV file, and of [ZipCodeRecord](#), a struct designed to contain one full row of data from the input file. The class [ZipCodeBuffer](#) implements buffered reading for efficient I/O operations.

#### 4.5.2 ASSUMPTIONS

This program makes the following working assumptions about the nature of the input file:

1. The first line of the CSV file is a header row, containing the names of each field but no actual data.
2. The CSV file has six fields per line, which are the zip code, name of the city or town, state or U.S. territory code, name of the county, latitude, and longitude, in that order.
3. The latitude and longitude fields are always numerical values.
4. The zip code field contains an integer between 0 and 99999, and uniquely identifies the row it is in.
5. The state code consists of two characters.

Violating these assumptions should not cause the program to crash, but may result in misformatted, incomplete, or otherwise incorrect output.

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 and ZipCodeRecordStruct
00004 * @author Teagen Lee (primary contributor)
00005 * @author Ethan Jackson (additional comments and formatting)
00006 * @author Dristi Barnwal, Marcus Julius, Natoli Mayu (reviewers)
00007 * @date February 2026
00008 *
00009 * This file contains the declarations of ZipCodeBuffer, a class which provides
00010 * functionality to read and parse ZIP code records from a CSV file, and of
00011 * ZipCodeRecord, a struct designed to contain one full row of data from the
00012 * input file. The class ZipCodeBuffer implements buffered reading for efficient
00013 * I/O operations.
00014 *
00015 * @section ASSUMPTIONS
00016 * This program makes the following working assumptions about the nature of the
00017 * input file:
00018 * -# The first line of the CSV file is a header row, containing the names of
```

```

00019 * each field but no actual data.
00020 * -# The CSV file has six fields per line, which are the zip code, name of the
00021 * city or town, state or U.S. territory code, name of the county, latitude, and
00022 * longitude, in that order.
00023 * -# The latitude and longitude fields are always numerical values.
00024 * -# The zip code field contains an integer between 0 and 99999, and uniquely
00025 * identifies the row it is in.
00026 * -# The state code consists of two characters.
00027 *
00028 * Violating these assumptions should not cause the program to crash, but may
00029 * result in misformatted, incomplete, or otherwise incorrect output.
00030 */
00031
00032 #ifndef ZIPCODEBUFFER_H
00033 #define ZIPCODEBUFFER_H
00034
00035 #include <string>
00036 #include <fstream>
00037 #include <sstream>
00038 #include <vector>
00039
00040 using namespace std;
00041
00042 /**
00043 * @struct ZipCodeRecord
00044 * @brief Structure to hold a single ZIP code record
00045 *
00046 * This structure represents one row from the ZIP code CSV file, containing all
00047 * relevant geographic and administrative information for a specific ZIP code.
00048 */
00049 struct ZipCodeRecord {
00050     int zipCode;           ///< The 5-digit ZIP code
00051     string placeName;      ///< Name of the place/city
00052     string state;          ///< Two-letter state abbreviation
00053     string county;         ///< County name
00054     double latitude;       ///< Latitude coordinate (decimal degrees)
00055     double longitude;      ///< Longitude coordinate (decimal degrees)
00056
00057 /**
00058 * @brief Default constructor
00059 *
00060 * Initializes all numeric fields to zero and strings to empty.
00061 */
00062 ZipCodeRecord();
00063
00064 /**
00065 * @brief Parameterized constructor
00066 * @param zip ZIP code number
00067 * @param place Place name
00068 * @param st State abbreviation
00069 * @param cnty County name
00070 * @param lat Latitude
00071 * @param lon Longitude
00072 */
00073 ZipCodeRecord(int zip, const string& place, const string& st, const
00074                 string& cnty, double lat, double lon);
00075 };
00076
00077 /**
00078 * @class ZipCodeBuffer
00079 * @brief A buffer class for reading ZIP code records from CSV files
00080 *
00081 * This class provides an abstraction layer for reading ZIP code data from a
00082 * comma-separated values (CSV) file. It handles file I/O, parsing, and error
00083 * checking while maintaining a clean interface for client code.
00084 *
00085 * The class uses internal buffering to efficiently read data from the file and
00086 * parse it into structured ZipCodeRecord objects.
00087 *
00088 * @note The CSV file must have a header row which is automatically skipped
00089 * @note Expected CSV format: ZipCode,PlaceName,State,County,Lat,Long
00090 */
00091 class ZipCodeBuffer {
00092 private:
00093     ifstream fileStream;      ///< Input file stream for reading CSV data
00094     string filename;          ///< Name of the CSV file being read
00095     bool headerSkipped;      ///< Flag to track if header row has been skipped
00096     long recordCount;        ///< Counter for total records read
00097
00098 /**
00099 * @brief Parses a CSV line into a ZipCodeRecord

```

```

00100     * @param line The CSV line to parse
00101     * @param record Reference to ZipCodeRecord to populate
00102     * @return true if parsing was successful, false otherwise
00103     *
00104     * This helper method takes a raw CSV line and extracts the individual
00105     * fields, converting them to appropriate data types and storing them in
00106     * the provided ZipCodeRecord structure.
00107     *
00108     * Handles quoted fields and embedded commas correctly. Fails (returns
00109     * false) if the CSV line does not match assumptions regarding the number,
00110     * ordering, and data types of its fields.
00111     *
00112     * @see ZipCodeBuffer.h
00113     */
00114     bool parseLine(const string& line, ZipCodeRecord& record);
00115
00116 /**
00117     * @brief Splits a CSV line into individual fields
00118     * @param line The CSV line to split
00119     * @return Vector of strings containing individual fields
00120     *
00121     * This utility function correctly handles CSV formatting including:
00122     * - Quoted fields
00123     * - Embedded commas within quotes
00124     * - Leading/trailing whitespace
00125     */
00126     vector<string> splitCSV(const string& line);
00127
00128 /**
00129     * @brief Trims whitespace from both ends of a string
00130     * @param str The string to trim
00131     * @return Trimmed string
00132     */
00133     string trim(const string& str);
00134
00135 public:
00136 /**
00137     * @brief Default constructor
00138     *
00139     * Creates an uninitialized ZipCodeBuffer. The open() method must be called
00140     * before reading any records.
00141     */
00142     ZipCodeBuffer();
00143
00144 /**
00145     * @brief Parameterized constructor
00146     * @param csvFilename Path to the CSV file to open
00147     *
00148     * Creates a ZipCodeBuffer and automatically opens the specified file. The
00149     * header row is skipped during initialization.
00150     */
00151     explicit ZipCodeBuffer(const string& csvFilename);
00152
00153 /**
00154     * @brief Destructor
00155     *
00156     * Ensures the file stream is properly closed when the object
00157     * is destroyed, preventing resource leaks.
00158     */
00159     ~ZipCodeBuffer();
00160
00161 /**
00162     * @brief Opens a CSV file for reading
00163     * @param csvFilename Path to the CSV file
00164     * @return true if file opened successfully, false otherwise
00165     *
00166     * Opens the specified CSV file and skips the header row. If a file is
00167     * already open, it is closed first.
00168     */
00169     bool open(const string& csvFilename);
00170
00171 /**
00172     * @brief Closes the currently open file
00173     *
00174     * Closes the file stream and resets internal state. Safe to call even if
00175     * no file is open.
00176     */
00177     void close();
00178
00179 /**
00180     * @brief Checks if a file is currently open

```

```
00181     * @return true if file is open and ready for reading, false otherwise
00182     */
00183     bool isOpen() const;
00184
00185     /**
00186     * @brief Reads the next ZIP code record from the file
00187     * @param record Reference to ZipCodeRecord to populate
00188     * @return true if a record was successfully read, false on EOF or error
00189     *
00190     * Reads one line from the CSV file, parses it, and populates the provided
00191     * ZipCodeRecord structure. Returns false when end of file is reached or if
00192     * a parsing error occurs.
00193     *
00194     * @note Automatically skips the header row on first read
00195     */
00196     bool readRecord(ZipCodeRecord& record);
00197
00198     /**
00199     * @brief Reads and gathers all records from the file
00200     * @return Vector containing all ZIP code records from the file
00201     *
00202     * Reads the entire CSV file and returns all valid records as a vector.
00203     * The file position is reset to the beginning (after header) when complete.
00204     *
00205     * @note This method loads all data into memory - use with caution for very
00206     * large files
00207     */
00208     vector<ZipCodeRecord> gatherAllRecords();
00209
00210     /**
00211     * @brief Resets the file position to the beginning (after header)
00212     * @return true if reset was successful, false otherwise
00213     *
00214     * Seeks back to the start of the file and skips the header row again,
00215     * allowing the file to be re-read without closing and reopening.
00216     */
00217     bool reset();
00218
00219     /**
00220     * @brief Gets the total number of records read so far
00221     * @return Count of records read
00222     */
00223     long getRecordCount() const;
00224
00225     /**
00226     * @brief Gets the name of the currently open file
00227     * @return Filename string
00228     */
00229     string getFilename() const;
00230 };
00231
00232 #include "ZipCodeBuffer.cpp"
00233
00234 #endif // ZIPCODEBUFFER_H
```



# Index

~ZipCodeBuffer  
    ZipCodeBuffer, 7

calculateStateExtremes  
    main.cpp, 21

close  
    ZipCodeBuffer, 7

county  
    ZipCodeRecord, 18

eastermost  
    StateExtremes, 3

filename  
    ZipCodeBuffer, 15

fileStream  
    ZipCodeBuffer, 15

gatherAllRecords  
    ZipCodeBuffer, 7

getFilename  
    ZipCodeBuffer, 8

getRecordCount  
    ZipCodeBuffer, 8

headerSkipped  
    ZipCodeBuffer, 16

isOpen  
    ZipCodeBuffer, 9

latitude  
    ZipCodeRecord, 18

longitude  
    ZipCodeRecord, 18

main  
    main.cpp, 21

main.cpp, 20  
    calculateStateExtremes, 21  
    main, 21  
    printStateExtremesTable, 22  
    smallerZipWins, 22

maxLatitude  
    StateExtremes, 3

maxLongitude  
    StateExtremes, 4

minLatitude  
    StateExtremes, 4

minLongitude  
    StateExtremes, 4

northernmost

open  
    ZipCodeBuffer, 9

parseLine  
    ZipCodeBuffer, 10

placeName  
    ZipCodeRecord, 19

printStateExtremesTable  
    main.cpp, 22

readRecord  
    ZipCodeBuffer, 11

recordCount  
    ZipCodeBuffer, 16

reset  
    ZipCodeBuffer, 12

smallerZipWins  
    main.cpp, 22

southernmost  
    StateExtremes, 4

splitCSV  
    ZipCodeBuffer, 13

state  
    ZipCodeRecord, 19

StateExtremes, 2  
    eastermost, 3  
    maxLatitude, 3  
    maxLongitude, 4  
    minLatitude, 4  
    minLongitude, 4  
    northernmost, 4  
    southernmost, 4  
    StateExtremes, 3  
    westernmost, 5

trim  
    ZipCodeBuffer, 14

westernmost  
    StateExtremes, 5

zipCode  
    ZipCodeRecord, 19

ZipCodeBuffer, 5  
    ~ZipCodeBuffer, 7  
    close, 7  
    filename, 15  
    fileStream, 15  
    gatherAllRecords, 7

getFilename, 8  
getRecordCount, 8  
headerSkipped, 16  
isOpen, 9  
open, 9  
parseLine, 10  
readRecord, 11  
recordCount, 16  
reset, 12  
splitCSV, 13  
trim, 14  
ZipCodeBuffer, 6  
ZipCodeBuffer.cpp, 26  
ZipCodeBuffer.h, 31  
ZipCodeRecord, 16  
  county, 18  
  latitude, 18  
  longitude, 18  
  placeName, 19  
  state, 19  
zipCode, 19  
ZipCodeRecord, 17