

Team 1: Zip Code Group Project  
Version 3.0

Generated by Doxygen 1.9.8



<b>1 Class Index</b>	<b>1</b>
1.1 Class List	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Class Documentation</b>	<b>5</b>
3.1 BlockBuffer Class Reference	5
3.1.1 Detailed Description	6
3.1.2 Constructor & Destructor Documentation	6
3.1.2.1 BlockBuffer()	6
3.1.3 Member Function Documentation	7
3.1.3.1 unpackBlockRecords()	7
3.1.3.2 readBlockMetadata()	8
3.1.3.3 getCurrentRBN()	9
3.1.3.4 getPrevRBN()	9
3.1.3.5 getNextRBN()	10
3.1.3.6 getNumRecordsInBlock()	10
3.1.3.7 readBlock()	10
3.1.3.8 readCurrentBlock()	11
3.1.3.9 readNextBlock()	12
3.1.3.10 calculateBlockAddress()	13
3.1.3.11 moveToBlock()	14
3.2 BlockSearch Class Reference	15
3.2.1 Detailed Description	15
3.2.2 Constructor & Destructor Documentation	16
3.2.2.1 BlockSearch() [1/2]	16
3.2.2.2 BlockSearch() [2/2]	16
3.2.3 Member Function Documentation	16
3.2.3.1 searchForRecord()	16
3.2.3.2 displayRecord()	18
3.3 Dump Class Reference	19
3.3.1 Detailed Description	20
3.3.2 Constructor & Destructor Documentation	21
3.3.2.1 Dump()	21
3.3.3 Member Function Documentation	21
3.3.3.1 dumpLogicalOrder()	21
3.3.3.2 dumpPhysicalOrder()	22
3.3.3.3 printBlock()	22
3.3.3.4 dumpBlockIndex()	23
3.4 HeaderBuffer::Field Struct Reference	24
3.4.1 Detailed Description	25
3.4.2 Member Data Documentation	25

3.4.2.1 zipCode . . . . .	25
3.4.2.2 placeName . . . . .	26
3.4.2.3 state . . . . .	26
3.4.2.4 county . . . . .	26
3.4.2.5 latitude . . . . .	26
3.4.2.6 longitude . . . . .	26
3.5 HeaderBuffer Class Reference . . . . .	27
3.5.1 Detailed Description . . . . .	28
3.5.2 Constructor & Destructor Documentation . . . . .	29
3.5.2.1 HeaderBuffer() [1/2] . . . . .	29
3.5.2.2 HeaderBuffer() [2/2] . . . . .	30
3.5.3 Member Function Documentation . . . . .	30
3.5.3.1 writeHeader() . . . . .	30
3.5.3.2 writeHeaderToFile() . . . . .	32
3.5.3.3 readHeader() . . . . .	33
3.5.3.4 calculateHeaderSize() . . . . .	35
3.5.3.5 setFileStructureType() . . . . .	36
3.5.3.6 setFileStructureVersion() . . . . .	36
3.5.3.7 setHeaderSizeBytes() . . . . .	37
3.5.3.8 setRecordSizeBytes() . . . . .	37
3.5.3.9 setSizeFormatType() . . . . .	37
3.5.3.10 setBlockSize() . . . . .	38
3.5.3.11 setminimumBlockCapacity() . . . . .	38
3.5.3.12 setPrimaryKeyIndexFileName() . . . . .	38
3.5.3.13 setprimaryKeyIndexFileSchema() . . . . .	39
3.5.3.14 setRecordCount() . . . . .	39
3.5.3.15 setBlockCount() . . . . .	39
3.5.3.16 setFieldCount() . . . . .	40
3.5.3.17 setPrimaryKeyFieldIndex() . . . . .	40
3.5.3.18 setRBNA() . . . . .	40
3.5.3.19 setRBNS() . . . . .	40
3.5.3.20 setstaleFlag() . . . . .	41
3.5.3.21 addField() . . . . .	41
3.5.3.22 getFileStructureType() . . . . .	41
3.5.3.23 getFileStructureVersion() . . . . .	41
3.5.3.24 getHeaderSizeBytes() . . . . .	42
3.5.3.25 getRecordSizeBytes() . . . . .	42
3.5.3.26 getSizeFormatType() . . . . .	42
3.5.3.27 getBlockSize() . . . . .	42
3.5.3.28 getMinimumBlockCapacity() . . . . .	42
3.5.3.29 getPrimaryKeyIndexFileName() . . . . .	42
3.5.3.30 getRecordCount() . . . . .	43

3.5.3.31 getBlockCount()	43
3.5.3.32 getFieldCount()	43
3.5.3.33 getPrimaryKeyFieldIndex()	43
3.5.3.34 getRBNA()	44
3.5.3.35 getRBNS()	44
3.5.3.36 getStateFlag()	44
3.5.3.37 getFields()	45
3.6 ZipCodeBuffer Class Reference	45
3.6.1 Detailed Description	46
3.6.2 Constructor & Destructor Documentation	47
3.6.2.1 ZipCodeBuffer()	47
3.6.3 Member Function Documentation	48
3.6.3.1 parseRecord()	48
3.6.3.2 readNextRecord()	49
3.6.3.3 getCurrentPosition()	51
3.6.3.4 setCurrentPosition()	51
3.6.4 Friends And Related Symbol Documentation	51
3.6.4.1 Dump	51
3.6.5 Member Data Documentation	52
3.6.5.1 blockBuffer	52
3.6.5.2 headerBuffer	52
3.7 ZipCodeIndexer Class Reference	52
3.7.1 Detailed Description	53
3.7.2 Constructor & Destructor Documentation	53
3.7.2.1 ZipCodeIndexer()	53
3.7.3 Member Function Documentation	54
3.7.3.1 createIndex()	54
3.7.3.2 writeIndexToFile()	55
3.7.3.3 loadIndexFromRAM()	55
3.7.3.4 getRecordPosition()	56
3.8 ZipCodeRecord Struct Reference	57
3.8.1 Detailed Description	58
3.8.2 Member Data Documentation	58
3.8.2.1 zipCode	58
3.8.2.2 placeName	58
3.8.2.3 state	58
3.8.2.4 county	58
3.8.2.5 latitude	59
3.8.2.6 longitude	59
<b>4 File Documentation</b>	<b>61</b>
4.1 block_idx_gen.cpp File Reference	61

4.1.1 Detailed Description . . . . .	62
4.1.2 Function Documentation . . . . .	62
4.1.2.1 findZipcode() . . . . .	62
4.1.2.2 main() . . . . .	63
4.2 block_idx_gen.cpp . . . . .	64
4.3 BlockBuffer.cpp File Reference . . . . .	65
4.4 BlockBuffer.cpp . . . . .	66
4.5 BlockBuffer.h File Reference . . . . .	67
4.6 BlockBuffer.h . . . . .	68
4.7 BlockGenerator.cpp File Reference . . . . .	69
4.7.1 Detailed Description . . . . .	69
4.7.2 Function Documentation . . . . .	70
4.7.2.1 main() . . . . .	70
4.8 BlockGenerator.cpp . . . . .	72
4.9 BlockSearch.cpp File Reference . . . . .	74
4.9.1 Function Documentation . . . . .	74
4.9.1.1 findZipcode() . . . . .	74
4.10 BlockSearch.cpp . . . . .	75
4.11 BlockSearch.h File Reference . . . . .	76
4.12 BlockSearch.h . . . . .	77
4.13 CSVConverter.cpp File Reference . . . . .	77
4.13.1 Function Documentation . . . . .	78
4.13.1.1 convertCSV() . . . . .	78
4.14 CSVConverter.cpp . . . . .	78
4.15 Dump.cpp File Reference . . . . .	79
4.15.1 Detailed Description . . . . .	79
4.16 Dump.cpp . . . . .	79
4.17 Dump.h File Reference . . . . .	81
4.18 Dump.h . . . . .	82
4.19 HeaderBuffer.cpp File Reference . . . . .	82
4.19.1 Function Documentation . . . . .	83
4.19.1.1 headerBuffer() . . . . .	83
4.20 HeaderBuffer.cpp . . . . .	84
4.21 HeaderBuffer.h File Reference . . . . .	90
4.22 HeaderBuffer.h . . . . .	90
4.23 headerTester.cpp File Reference . . . . .	91
4.23.1 Function Documentation . . . . .	92
4.23.1.1 main() . . . . .	92
4.24 headerTester.cpp . . . . .	93
4.25 ZipCodeBuffer.cpp File Reference . . . . .	94
4.26 ZipCodeBuffer.cpp . . . . .	95
4.27 ZipCodeBuffer.h File Reference . . . . .	96

4.28 ZipCodeBuffer.h . . . . .	97
4.29 ZipCodeIndexer.cpp File Reference . . . . .	98
4.30 ZipCodeIndexer.cpp . . . . .	99
4.31 ZipCodeIndexer.h File Reference . . . . .	100
4.32 ZipCodeIndexer.h . . . . .	101
4.33 ZipCodeRecordSearch.cpp File Reference . . . . .	101
4.33.1 Function Documentation . . . . .	102
4.33.1.1 isNumber() . . . . .	102
4.33.1.2 displayHelp() . . . . .	103
4.33.1.3 defaultMessage() . . . . .	103
4.33.1.4 searchHelper() . . . . .	104
4.34 ZipCodeRecordSearch.cpp . . . . .	105
4.35 ZipCodeRecordSearch.h File Reference . . . . .	106
4.35.1 Macro Definition Documentation . . . . .	107
4.35.1.1 ZIPCODERECORDSEARCH_H . . . . .	107
4.35.2 Function Documentation . . . . .	107
4.35.2.1 isNumber() . . . . .	107
4.35.2.2 displayHelp() . . . . .	108
4.35.2.3 defaultMessage() . . . . .	109
4.35.2.4 searchHelper() . . . . .	109
4.36 ZipCodeRecordSearch.h . . . . .	111
4.37 ZipCodeTableViewer.cpp File Reference . . . . .	111
4.37.1 Detailed Description . . . . .	112
4.37.2 Function Documentation . . . . .	113
4.37.2.1 main() . . . . .	113
4.38 ZipCodeTableViewer.cpp . . . . .	116





# Chapter 1

## Class Index

### 1.1 Class List

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

<a href="#">BlockBuffer</a>	See <a href="#">BlockBuffer.h</a> for full documentation . . . . .	5
<a href="#">BlockSearch</a>	Implementation of the <a href="#">BlockSearch</a> class for searching for records in the blocked index file . .	15
<a href="#">Dump</a>	Class for dumping and printing block records in logical and physical order and for dumping the simple index for the blocks . . . . .	19
<a href="#">HeaderBuffer::Field</a>	. . . . .	24
<a href="#">HeaderBuffer</a>	Implementation of the <a href="#">HeaderBuffer</a> class for for handling header data . . . . .	27
<a href="#">ZipCodeBuffer</a>	Parses the file one record at a time and returns the fields in a <a href="#">ZipCodeRecord</a> struct . . . . .	45
<a href="#">ZipCodeIndexer</a>	Implementation of the <a href="#">ZipCodeIndexer</a> class for indexing ZIP code records in a file . . . . .	52
<a href="#">ZipCodeRecord</a>	Structure to hold a ZIP Code record . . . . .	57



## Chapter 2

# File Index

### 2.1 File List

Here is a list of all files with brief descriptions:

<a href="#">block_idx_gen.cpp</a>	
This class creates an index file for a blocked data file	61
<a href="#">BlockBuffer.cpp</a>	65
<a href="#">BlockBuffer.h</a>	67
<a href="#">BlockGenerator.cpp</a>	
(Blocked Sequence Set Generator)	69
<a href="#">BlockSearch.cpp</a>	74
<a href="#">BlockSearch.h</a>	76
<a href="#">CSVConverter.cpp</a>	77
<a href="#">Dump.cpp</a>	
Implementation of the <a href="#">Dump</a> class methods for printing and dumping records	79
<a href="#">Dump.h</a>	81
<a href="#">HeaderBuffer.cpp</a>	82
<a href="#">HeaderBuffer.h</a>	90
<a href="#">headerTester.cpp</a>	91
<a href="#">ZipCodeBuffer.cpp</a>	94
<a href="#">ZipCodeBuffer.h</a>	96
<a href="#">ZipCodeIndexer.cpp</a>	98
<a href="#">ZipCodeIndexer.h</a>	100
<a href="#">ZipCodeRecordSearch.cpp</a>	101
<a href="#">ZipCodeRecordSearch.h</a>	106
<a href="#">ZipCodeTableViewer.cpp</a>	
Console program for displaying a table of the most eastern, western, northern, and southern ZIP codes for each state code in a file	111



## Chapter 3

# Class Documentation

### 3.1 BlockBuffer Class Reference

See [BlockBuffer.h](#) for full documentation.

```
#include <BlockBuffer.h>
```

Collaboration diagram for BlockBuffer:

BlockBuffer
<div>+ BlockBuffer() + unpackBlockRecords() + readBlockMetadata() + getCurrentRBN() + getPrevRBN() + getNextRBN() + getNumRecordsInBlock() + readBlock() + readCurrentBlock() + readNextBlock() + calculateBlockAddress() + moveToBlock()</div>

## Public Member Functions

- [BlockBuffer](#) (std::ifstream &file, [HeaderBuffer](#) headerBuffer)  
*Construct a new Block Buffer object.*
- vector< string > [unpackBlockRecords](#) ()  
*Unpacks the length-indicated records from the block into a string vector.*
- void [readBlockMetadata](#) ()  
*Reads the block metadata for the current block.*
- int [getCurrentRBN](#) () const
- int [getPrevRBN](#) () const
- int [getNextRBN](#) () const
- int [getNumRecordsInBlock](#) () const
- vector< string > [readBlock](#) (int relativeBlockNumber)  
*Reads the block at the given Relative Block Number (RBN) and returns it as a vector of records in string form.*
- vector< string > [readCurrentBlock](#) ()  
*Reads the current block after the file pointer and returns it as a vector of records in string form.*
- vector< string > [readNextBlock](#) ()  
*Moves to and reads the next block and returns it as a vector of records in string form.*
- int [calculateBlockAddress](#) (int relativeBlockNumber)  
*Calculates the address of a Relative Block Number (RBN) within the file.*
- void [moveToBlock](#) (int relativeBlockNumber)  
*Moves the file pointer to the address of the block at the given Relative Block Number (RBN).*

### 3.1.1 Detailed Description

See [BlockBuffer.h](#) for full documentation.

Reads blocks of length-indicated records from a blocked file.

#### Author

Kent Biernath  
Andrew Clayton

#### Date

2023-11-14

#### Version

1.0

Definition at line 48 of file [BlockBuffer.h](#).

### 3.1.2 Constructor & Destructor Documentation

#### 3.1.2.1 BlockBuffer()

```
BlockBuffer::BlockBuffer (
    std::ifstream & file,
    HeaderBuffer headerBuffer = HeaderBuffer("blocked_postal_codes.txt") )
```

Construct a new Block Buffer object.

## Parameters

<i>file</i>	The file to read.
<i>headerBuffer</i>	A <a href="#">HeaderBuffer</a> object for the file.

## Precondition

: The block is a string.

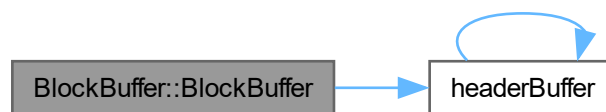
## Postcondition

: A new Block Buffer object is created.

Definition at line 15 of file [BlockBuffer.cpp](#).

```
00015 : file(file) { // TODO remove HeaderBuffer filename once it allows generic constructor
00016     headerBuffer.readHeader();
00017     headerSize = headerBuffer.getHeaderSizeBytes();
00018     blockSize = headerBuffer.getBlockSize();
00019     nextRBN = headerBuffer.getRBNS();
00020 }
```

Here is the call graph for this function:



### 3.1.3 Member Function Documentation

#### 3.1.3.1 unpackBlockRecords()

```
vector< string > BlockBuffer::unpackBlockRecords ( )
```

Unpacks the length-indicated records from the block into a string vector.

## Returns

A vector of strings, the records within a block.

## Precondition

: The file pointer is at the start of the records within the block after the block metadata was read.

**Postcondition**

: The block is unpacked into individual strings for each record in the block, returned as a vector.

Definition at line 23 of file [BlockBuffer.cpp](#).

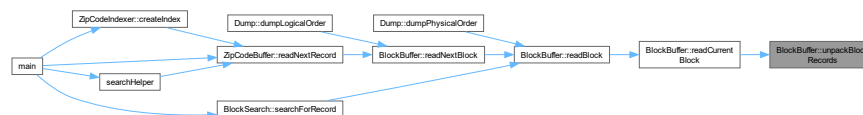
```

00023                                     {
00024     // This will convert a block to a vector of records
00025     size_t idx = 0;
00026     vector<string> records;
00027
00028     for (size_t i = 0; i < getNumRecordsInBlock(); i++)
00029     {
00030         // Reads the length and retrieves that many characters for the record
00031         std::string recordString;
00032         int numCharactersToRead = 0;
00033         file » numCharactersToRead; // Read the length indicator, the first field in each record
00034         file.ignore(1);           // Skip the comma after the length field
00035         recordString.resize(numCharactersToRead);
00036         file.read(&recordString[0], numCharactersToRead);
00037         records.push_back(recordString);
00038     }
00039
00040     return records;
00041 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

**3.1.3.2 readBlockMetadata()**

```
void BlockBuffer::readBlockMetadata ( )
```

Reads the block metadata for the current block.

**Precondition**

The file pointer is at the start of the block before the 5 metadata fields.



**Postcondition**

The 5 metadata fields have been read into the member variables and the file pointer is after the metadata.

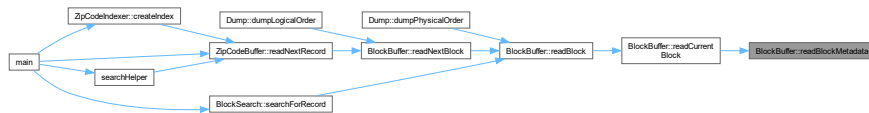
Definition at line 46 of file [BlockBuffer.cpp](#).

```

00046 {
00047     int metadataRecordLength = -1;
00048     int newRelativeBlockNumber = -1;
00049     int newNumRecordsInBlock = -1;
00050     int newPrevRBN = -1;
00051     int newNextRBN = -1;
00052
00053     file » metadataRecordLength;
00054     file.ignore(1); // Ignore the commas separating the fields
00055     file » newRelativeBlockNumber;
00056     file.ignore(1);
00057     file » newNumRecordsInBlock;
00058     file.ignore(1);
00059     file » newPrevRBN;
00060     file.ignore(1);
00061     file » newNextRBN;
00062     file.ignore(1); // Skip the comma after the last metadata field
00063
00064     // TODO throw exception if any of these reads failed or the values are invalid
00065
00066     currentRBN = newRelativeBlockNumber;
00067     numRecordsInBlock = newNumRecordsInBlock;
00068     prevRBN = newPrevRBN;
00069     nextRBN = newNextRBN;
00070 }

```

Here is the caller graph for this function:

**3.1.3.3 getCurrentRBN()**

```
int BlockBuffer::getCurrentRBN ( ) const [inline]
```

Definition at line 87 of file [BlockBuffer.h](#).

```
00087 { return currentRBN; }
```

**3.1.3.4 getPrevRBN()**

```
int BlockBuffer::getPrevRBN ( ) const [inline]
```

Definition at line 88 of file [BlockBuffer.h](#).

```
00088 { return prevRBN; }
```

Here is the caller graph for this function:



### 3.1.3.5 getNextRBN()

```
int BlockBuffer::getNextRBN ( ) const [inline]
```

Definition at line 89 of file [BlockBuffer.h](#).

```
00089 { return nextRBN; }
```

Here is the caller graph for this function:



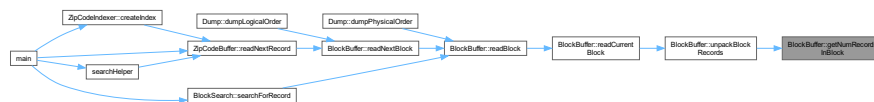
### 3.1.3.6 getNumRecordsInBlock()

```
int BlockBuffer::getNumRecordsInBlock ( ) const [inline]
```

Definition at line 90 of file [BlockBuffer.h](#).

```
00090 { return numRecordsInBlock; }
```

Here is the caller graph for this function:



### 3.1.3.7 readBlock()

```
vector< string > BlockBuffer::readBlock (
    int relativeBlockNumber )
```

Reads the block at the given Relative Block Number (RBN) and returns it as a vector of records in string form.

#### Returns

A vector of strings where each string represents a record within the block.  
The length indication field for each record is read but not returned in the string.

#### Precondition

: The file is open and in a blocked length-indicated file format.

**Postcondition**

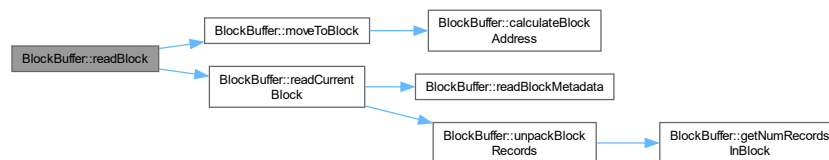
: The block is broken down into records and the file pointer is after the records in the block.

Definition at line 90 of file [BlockBuffer.cpp](#).

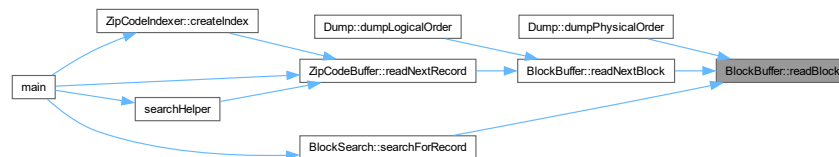
```

00090                                     {
00091     vector<string> recordStrings;
00092     std::string line;
00093
00094     // If the RBN is -1, the end of the chain has been reached.
00095     if (relativeBlockNumber == -1)
00096     {
00097         currentRBN = -1;
00098         return recordStrings;
00099     }
00100
00101     moveToBlock(relativeBlockNumber);           // Move to the next block
00102     return readCurrentBlock();                  // Read the metadata and the records
00103 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

**3.1.3.8 readCurrentBlock()**

```
vector< string > BlockBuffer::readCurrentBlock ( )
```

Reads the current block after the file pointer and returns it as a vector of records in string form.

Reads the current block and returns it as a vector of records in string form.

**Returns**

A vector of strings where each string represents a record within the block.

The length indication field for each record is read but not returned in the string.

**Precondition**

: The file pointer is at the start of the block.

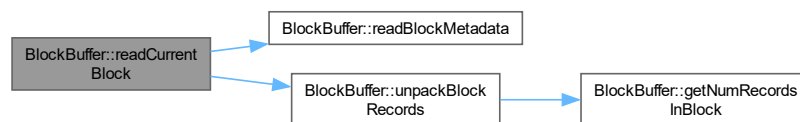
**Postcondition**

: The block is broken down into records and the file pointer is after the records in the block.

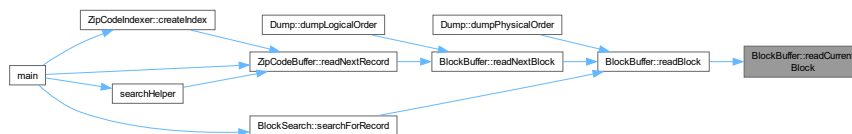
Definition at line 108 of file [BlockBuffer.cpp](#).

```
00108 {
00109     readBlockMetadata();           // Read the metadata for the block
00110     return unpackBlockRecords();   // Read the length-indicated records into strings and return
    them
00111 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

**3.1.3.9 readNextBlock()**

```
vector< string > BlockBuffer::readNextBlock ( )
```

Moves to and reads the next block and returns it as a vector of records in string form.

**Returns**

A vector of strings where each string represents a record within the block.  
The length indication field for each record is read but not returned in the string.

**Precondition**

: The file is open and in a blocked length-indicated file format.

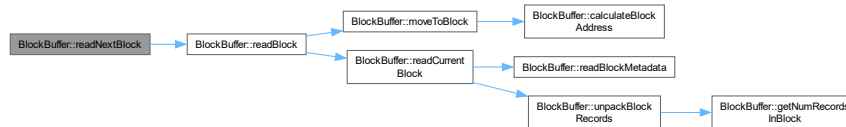
**Postcondition**

: The block is broken down into records and the file pointer is after the records in the block.

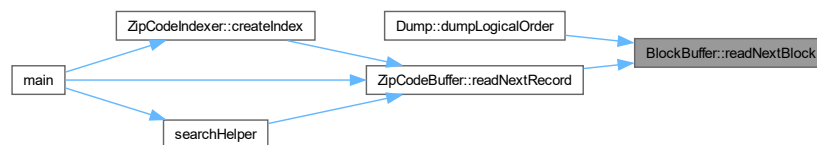
Definition at line 114 of file [BlockBuffer.cpp](#).

```
00114 {
00115     return readBlock(nextRBN);
00116 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

**3.1.3.10 calculateBlockAddress()**

```
int BlockBuffer::calculateBlockAddress (
    int relativeBlockNumber )
```

Calculates the address of a Relative Block Number (RBN) within the file.

**Returns**

The address of the RBN.

**Precondition**

The file metadata has been read.

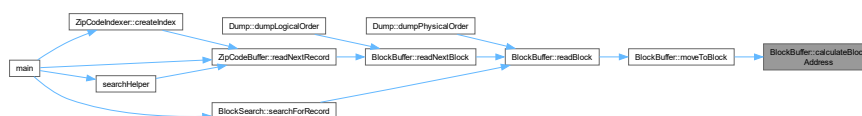
**Postcondition**

The calculation results have been returned.

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

```
00075 {
00076     return headerSize + relativeBlockNumber*blockSize;
00077 }
```

Here is the caller graph for this function:



### 3.1.3.11 moveToBlock()

```
void BlockBuffer::moveToBlock (
    int relativeBlockNumber )
```

Moves the file pointer to the address of the block at the given Relative Block Number (RBN).

#### Precondition

The file is open.

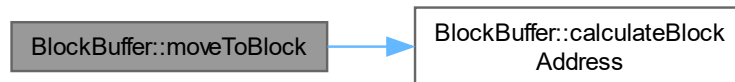
#### Postcondition

The file pointer is moved to the start of the block at the given RBN.

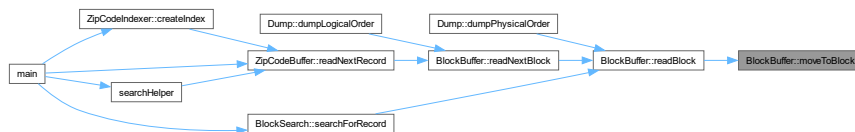
Definition at line 82 of file [BlockBuffer.cpp](#).

```
00082     {
00083     int address = calculateBlockAddress(relativeBlockNumber);
00084     file.seekg(address);
00085 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



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

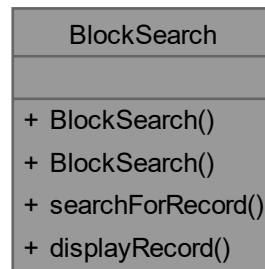
- [BlockBuffer.h](#)
- [BlockBuffer.cpp](#)

## 3.2 BlockSearch Class Reference

Implementation of the [BlockSearch](#) class for searching for records in the blocked index file.

```
#include <BlockSearch.h>
```

Collaboration diagram for BlockSearch:



### Public Member Functions

- [BlockSearch](#) ()  
*Construct a new Block Search object.*
- [BlockSearch](#) (string idxFile)  
*Constructor that takes in a blocked index file.*
- string [searchForRecord](#) (int target)  
*Searches for a record in the blocked index file by key (zipcode).*
- void [displayRecord](#) (string record)  
*Displays the record to the console.*

### 3.2.1 Detailed Description

Implementation of the [BlockSearch](#) class for searching for records in the blocked index file.

Uses the blocked index file to be able to search for specific records in the file.

See [BlockSearch.h](#) for full documentation.

#### Author

Andrew Clayton

#### Date

2023-11-14

#### Version

1.0

Definition at line 26 of file [BlockSearch.h](#).

## 3.2.2 Constructor & Destructor Documentation

### 3.2.2.1 BlockSearch() [1/2]

```
BlockSearch::BlockSearch ( ) [inline]
```

Construct a new Block Search object.

#### Precondition

: none

#### Postcondition

: A new [BlockSearch](#) object is created

Definition at line 39 of file [BlockSearch.h](#).

```
00039 { indexFile = "blocked_Index.txt"; }
```

### 3.2.2.2 BlockSearch() [2/2]

```
BlockSearch::BlockSearch (
    string idxFile )
```

Constructor that takes in a blocked index file.

#### Parameters

<i>indexFile</i>	The file to open
------------------	------------------

#### Precondition

: A blocked index file exists

#### Postcondition

: A new [BlockSearch](#) object is created

Definition at line 18 of file [BlockSearch.cpp](#).

```
00018 {
00019     indexFile = idxFile;
00020 }
```

## 3.2.3 Member Function Documentation

### 3.2.3.1 searchForRecord()

```
string BlockSearch::searchForRecord (
    int target )
```

Searches for a record in the blocked index file by key (zipcode).



## Parameters

<i>target</i>	The zipcode to search for
---------------	---------------------------

## Precondition

: A blocked index file exists

## Postcondition

: The record is either found or not found

## Returns

: The record if it is found, or a -1 if it is not found

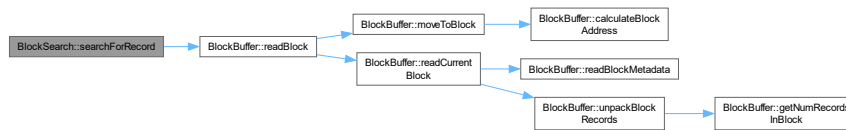
Definition at line 35 of file [BlockSearch.cpp](#).

```

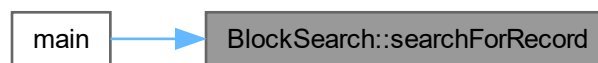
00035                                     {
00036     // Open the index file
00037     ifstream readFile(indexFile);
00038     string line;
00039
00040     // Read through the file until we find target <= greatestKeyInBlock
00041
00042     // Iterate through each line of the file
00043     while (getline(readFile, line)) {
00044         int commaIdx = line.find(',');
00045         int rbn = 0;
00046         try {
00047             rbn = stoi(line.substr(0, commaIdx));
00048         } catch (invalid_argument& e) {
00049             cerr << "Error parsing RBN: " << e.what() << endl;
00050             // return "-1";
00051         }
00052
00053         int greatestKeyInBlock;
00054         try {
00055             greatestKeyInBlock = stoi(line.substr(commaIdx+1));
00056         } catch (invalid_argument& e) {
00057             cerr << "Error parsing greatest key in block: " << e.what() << endl;
00058             // return "-1";
00059         }
00060
00061         if (target <= greatestKeyInBlock) {
00062             // We have found the block that contains the record we are looking for
00063             // now we need to actually access the block itself, which we should be able to do with
00064             BlockBuffer
00065
00066             ifstream dataFile("us_postal_codes_blocked.txt", std::ios::app);
00067             HeaderBuffer headerBuffer2("us_postal_codes_blocked.txt");
00068             BlockBuffer blockbuffer(dataFile, headerBuffer2);
00069
00070             // We break down all the block into a vector of records
00071
00072             vector<string> records = blockbuffer.readBlock(rbn);
00073
00074             for (string record : records) { // Check if each record is the target record
00075                 int commaIdx = record.find(',');
00076                 int zipcode = stoi(record.substr(0, commaIdx));
00077
00078                 if (zipcode == target) {
00079                     return record;
00080                 }
00081             }
00082             break; // not found in this block, and next blocks have greater keys
00083         }
00084         // We could not find the block that contains the record we are looking for
00085         return "-1";
00086     }

```

Here is the call graph for this function:



Here is the caller graph for this function:



### 3.2.3.2 displayRecord()

```
void BlockSearch::displayRecord (
    string record )
```

Displays the record to the console.

#### Parameters

<i>record</i>	The record to display, in its raw data form
---------------	---

#### Precondition

: A record exists

#### Postcondition

: The record is displayed to the console

Definition at line 88 of file [BlockSearch.cpp](#).

```

00088     {
00089         // The format of a record is: zipcode,town,state,county,latitude,longitude
00090         vector<string> fields;
00091         stringstream ss(record);
00092         string field;
00093
00094         while (getline(ss, field, ',')) {
00095             fields.push_back(field);
00096         }
00097
00098         if (fields.size() == 6) { // Ensure there are exactly 6 fields: zipcode, town, state, county,
latitude, longitude
00099             cout << "Zipcode: " << fields[0] << endl;
  
```

```

00100         cout << "Town: " << fields[1] << endl;
00101         cout << "State: " << fields[2] << endl;
00102         cout << "County: " << fields[3] << endl;
00103         cout << "Latitude: " << fields[4] << endl;
00104         cout << "Longitude: " << fields[5] << "\n\n";
00105     } else {
00106         cerr << "Zipcode not found." << endl;
00107     }
00108
00109 }

```

Here is the caller graph for this function:



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

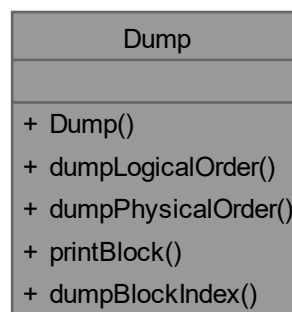
- [BlockSearch.h](#)
- [BlockSearch.cpp](#)

### 3.3 Dump Class Reference

Class for dumping and printing block records in logical and physical order and for dumping the simple index for the blocks.

```
#include <Dump.h>
```

Collaboration diagram for Dump:



## Public Member Functions

- [Dump](#) ([ZipCodeBuffer](#) &recordBuffer)
- void [dumpLogicalOrder](#) ()  
*Dump records in logical order.*
- void [dumpPhysicalOrder](#) ()  
*Dump records in physical order.*
- void [printBlock](#) (std::vector< std::string > records)  
*Print a block of records.*
- void [dumpBlockIndex](#) (const std::string &filename)  
*Dump the block index from a file.*

### 3.3.1 Detailed Description

Class for dumping and printing block records in logical and physical order and for dumping the simple index for the blocks.

The [Dump](#) class provides methods to dump and print records in logical and physical order. It takes a [ZipCodeBuffer](#) object during initialization and uses it to access and process ZIP code records.

The constructor takes a reference to a [ZipCodeBuffer](#) object, initializing the [BlockBuffer](#) and [HeaderBuffer](#) for further use. It also reads the header information from the file.

The class includes methods for dumping records in logical order ([dumpLogicalOrder](#)), dumping records in physical order ([dumpPhysicalOrder](#)), and printing a block index from a specified file ([dumpBlockIndex](#)).

The logical order dump iterates through the records in a file using the `readNextBlock` method of [BlockBuffer](#).

The physical order dump iterates through the blocks in a file using the `readBlock` method of [BlockBuffer](#).

The `printBlock` method is used internally to print the records within a block.

The class assumes that the ZIP code records are stored in a file and follow a specific format. It also assumes that the file has header information and contains a specific structure of blocks.

#### Author

Kent Biernath  
Tristan Adams

#### Date

2023-11-19

#### Version

1.0

Definition at line 56 of file [Dump.h](#).

### 3.3.2 Constructor & Destructor Documentation

#### 3.3.2.1 Dump()

```
Dump::Dump (
    ZipCodeBuffer & recordBuffer )
```

Definition at line 18 of file [Dump.cpp](#).

```
00018                                     : recordBuffer(recordBuffer),
    blockBuffer(recordBuffer.blockBuffer), headerBuffer(recordBuffer.headerBuffer) {
00019     // The constructor takes an ifstream and HeaderBuffer, initializing the BlockBuffer
00020     // with the provided file and headerBuffer.
00021     headerBuffer.readHeader();
00022 }
```

Here is the call graph for this function:



### 3.3.3 Member Function Documentation

#### 3.3.3.1 dumpLogicalOrder()

```
void Dump::dumpLogicalOrder ( )
```

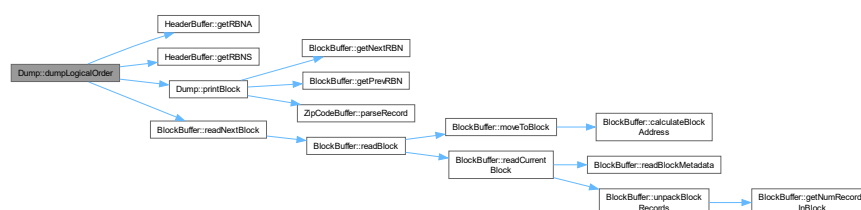
[Dump](#) records in logical order.

Prints the List Head and Avail Head Relative Block Numbers (RBNS and RBNA) and iterates through each block in logical order, printing its contents.

Definition at line 27 of file [Dump.cpp](#).

```
00027     {
00028     // Display the list head Relative Block Numbers
00029     std::cout << "List Head RBN: " << headerBuffer.getRBNS() << std::endl;
00030     std::cout << "Avail Head RBN: " << headerBuffer.getRBNA() << std::endl;
00031
00032     while (true)
00033     {
00034         std::vector<std::string> records = blockBuffer.readNextBlock();
00035         if (records.size() == 0) // TODO could change to when it reads -1 as next RBN to skip a step
00036         {
00037             // When it receives an empty vector, end the loop
00038             break;
00039         }
00040
00041         printBlock(records);
00042     }
00043 }
```

Here is the call graph for this function:



### 3.3.3.2 dumpPhysicalOrder()

```
void Dump::dumpPhysicalOrder ( )
```

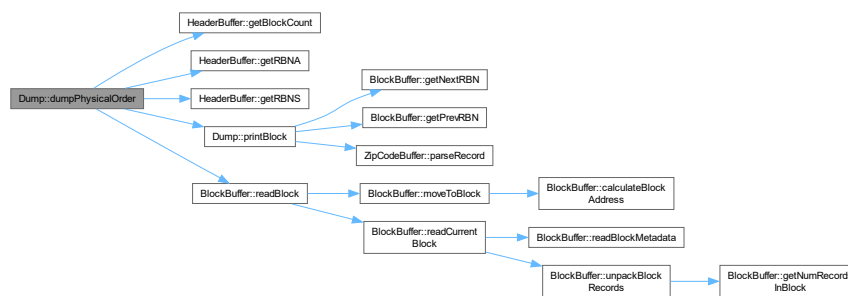
**Dump** records in physical order.

Prints the List Head and Avail Head Relative Block Numbers (RBNS and RBNA) and iterates through each block in physical order, printing its contents.

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

```
00048 {
00049     // Display the list head Relative Block Numbers
00050     std::cout << "List Head RBN: " << headerBuffer.getRBNS() << std::endl;
00051     std::cout << "Avail Head RBN: " << headerBuffer.getRBNA() << std::endl;
00052
00053     int i = 0;
00054     int endpoint = headerBuffer.getBlockCount();
00055     while (i < endpoint)
00056     {
00057         // Read the number of blocks listed in the file metadata
00058         std::vector<std::string> records = blockBuffer.readBlock(i++);
00059         printBlock(records);
00060     }
00061 }
```

Here is the call graph for this function:



### 3.3.3.3 printBlock()

```
void Dump::printBlock (
    std::vector< std::string > records )
```

Print a block of records.

Prints the Previous Relative Block Number (PrevRBN), the Zip Codes from the records in the block, and the Next Relative Block Number (NextRBN).

#### Parameters

<i>records</i>	A vector of strings representing the records in the block.
----------------	--

Definition at line 66 of file [Dump.cpp](#).

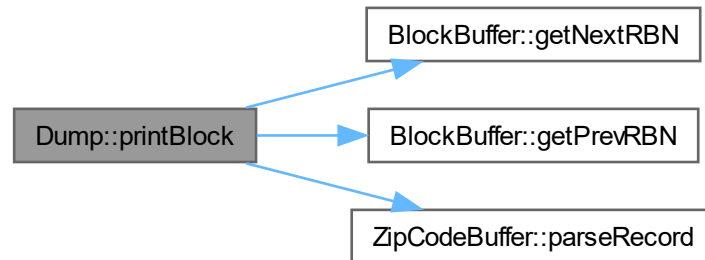
```
00066 {
00067
00068     std::cout << std::left << std::setw(6) << blockBuffer.getPrevRBN(); // Display preceding RBN
```

```

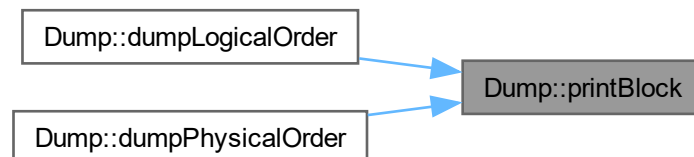
00069     for (const std::string& recordString : records) {
00070         ZipCodeRecord record = recordBuffer.parseRecord(recordString);
00071         std::cout << std::setw(6) << record.zipCode;           // Display the key (zipCode) for
        each record in the block
00072     }
00073     std::cout << std::left << std::setw(6) << blockBuffer.getNextRBN(); // Display succeeding RBN
00074     std::cout << std::endl;
00075 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



#### 3.3.3.4 dumpBlockIndex()

```

void Dump::dumpBlockIndex (
    const std::string & filename )

```

**Dump** the block index from a file.

Reads and prints the Relative Block Numbers (RBN) and Primary Key values from each line of the specified file.

##### Parameters

<i>filename</i>	The name of the file containing the block index.
-----------------	--

Definition at line 80 of file [Dump.cpp](#).

```

00080                                     {
00081     std::ifstream mainFile(filename);
00082     if (!mainFile.is_open()) {
00083         std::cerr << "Error opening index file." << std::endl;
00084         return;
00085     }
00086
00087     std::string line;
00088     while (std::getline(mainFile, line)) {
00089         std::istringstream ss(line);
00090         std::vector<std::string> tokens;
00091
00092         // Split the line by comma and store tokens in the vector
00093         while (std::getline(ss, line, ',')) {
00094             tokens.push_back(line);
00095         }
00096         std::cout << "RBN: ";
00097         int key = 0;
00098         // Print each value in the tokens vector
00099         for (const auto& token : tokens) {
00100             std::cout << token << " ";
00101             if(key == 0){
00102                 std::cout << "Primary Key: ";
00103                 key = 1;
00104             }else{
00105                 key = 0;
00106             }
00107         }
00108
00109         std::cout << "\n";
00110     }
00111 }
00112 }
```

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

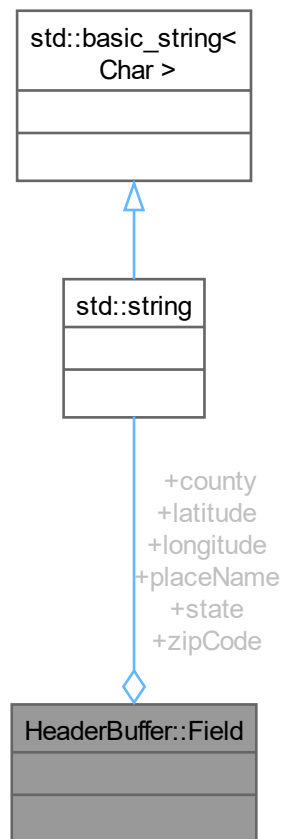
- [Dump.h](#)
- [Dump.cpp](#)

## 3.4 HeaderBuffer::Field Struct Reference

```
#include <HeaderBuffer.h>
```



Collaboration diagram for HeaderBuffer::Field:



#### Public Attributes

- `std::string` [zipCode](#)
- `std::string` [placeName](#)
- `std::string` [state](#)
- `std::string` [county](#)
- `std::string` [latitude](#)
- `std::string` [longitude](#)

### 3.4.1 Detailed Description

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

### 3.4.2 Member Data Documentation

#### 3.4.2.1 zipCode

`std::string HeaderBuffer::Field::zipCode`

Definition at line 56 of file [HeaderBuffer.h](#).

#### 3.4.2.2 placeName

```
std::string HeaderBuffer::Field::placeName
```

Definition at line 57 of file [HeaderBuffer.h](#).

#### 3.4.2.3 state

```
std::string HeaderBuffer::Field::state
```

Definition at line 58 of file [HeaderBuffer.h](#).

#### 3.4.2.4 county

```
std::string HeaderBuffer::Field::county
```

Definition at line 59 of file [HeaderBuffer.h](#).

#### 3.4.2.5 latitude

```
std::string HeaderBuffer::Field::latitude
```

Definition at line 60 of file [HeaderBuffer.h](#).

#### 3.4.2.6 longitude

```
std::string HeaderBuffer::Field::longitude
```

Definition at line 61 of file [HeaderBuffer.h](#).

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

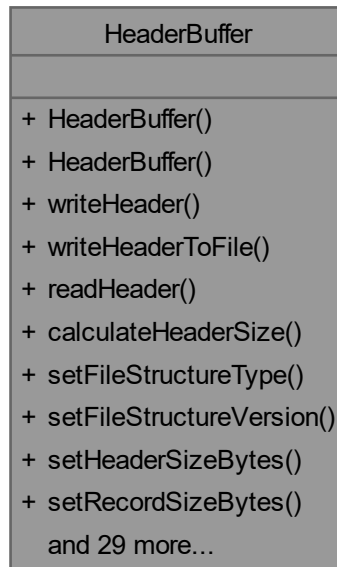
- [HeaderBuffer.h](#)

## 3.5 HeaderBuffer Class Reference

Implementation of the [HeaderBuffer](#) class for for handling header data.

```
#include <HeaderBuffer.h>
```

Collaboration diagram for HeaderBuffer:



### Classes

- struct [Field](#)

### Public Member Functions

- [HeaderBuffer](#) ()  
*Constructor to initialize [HeaderBuffer](#) without a filename.*
- [HeaderBuffer](#) (const std::string &filename)  
*Constructor to initialize [HeaderBuffer](#) with a filename.*
- void [writeHeader](#) ()  
*Write the header data to a file held the by object.*
- void [writeHeaderToFile](#) (const std::string &filename)  
*Write the header data to a file passed to the object.*
- void [readHeader](#) ()  
*Read header data from a file.*
- int [calculateHeaderSize](#) () const  
*calculates the size of the header in bytes*
- void [setFileStructureType](#) (const std::string &fileStructureType)

*Setters for various header fields.*

- void [setFileStructureVersion](#) (const std::string &fileStructureVersion)
- void [setHeaderSizeBytes](#) (int headerSizeBytes)
- void [setRecordSizeBytes](#) (int recordSizeBytes)
- void [setSizeFormatType](#) (const std::string &sizeFormatType)
- void [setBlockSize](#) (int blockSize)
- void [setminimumBlockCapacity](#) (int minimumBlockCapacity)
- void [setPrimaryKeyIndexFileName](#) (const std::string &primaryKeyIndexFileName)
- void [setprimaryKeyIndexFileSchema](#) (const std::string &primaryKeyIndexFileSchema)
- void [setRecordCount](#) (int recordCount)
- void [setBlockCount](#) (int blockCount)
- void [setFieldCount](#) (int fieldCount)
- void [setPrimaryKeyFieldIndex](#) (int primaryKeyFieldIndex)
- void [setRBNA](#) (int RBNA)
- void [setRBNS](#) (int RBNS)
- void [setstaleFlag](#) (int staleFlag)
- void [addField](#) (const [Field](#) &field)

*Add a field to the header.*

- std::string [getFileStructureType](#) () const

*Getters for header fields.*

- std::string [getFileStructureVersion](#) () const
- int [getHeaderSizeBytes](#) () const
- int [getRecordSizeBytes](#) () const
- std::string [getSizeFormatType](#) () const
- int [getBlockSize](#) () const
- int [getMinimumBlockCapacity](#) () const
- std::string [getPrimaryIndexFileName](#) () const
- int [getRecordCount](#) () const
- int [getBlockCount](#) () const
- int [getFieldCount](#) () const
- int [getPrimaryKeyFieldIndex](#) () const
- int [getRBNA](#) () const
- int [getRBNS](#) () const
- int [getStaleFlag](#) () const
- const std::vector< [Field](#) > & [getFields](#) () const

### 3.5.1 Detailed Description

Implementation of the [HeaderBuffer](#) class for for handling header data.

Represents a class for handling header data.

The [HeaderBuffer](#) class is responsible for reading and writing header data to and from a file.

Header Fields:

- File Structure Type (string)
- File Structure Version (string)
- Header Size (bytes) (int)
- Record Size (bytes) (int)
- Size Format Type (string)
- Primary Key Index File (string)
- Record Count (int)
- [Field](#) Count (int)

- Primary Key [Field](#) Index (int)
- Fields
- Zip Code (string)
- Place Name (string)
- State (string)
- County (string)
- Latitude (double)
- Longitude (double)

Whenever `readHeader` is called, it reads the header data from the file specified in the constructor.

The name of the header file to be opened is passed to the class constructor as a string.

Assumptions:

- The header file is in the same directory as the program.
- The header file format follows a specific structure, as described in the code.

See [HeaderBuffer.h](#) for the class declaration and documentation.

#### Author

Emma Hoffmann  
Kent Biernath  
Tristan Adams

#### Date

2023-10-16

#### Version

2.0

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

## 3.5.2 Constructor & Destructor Documentation

### 3.5.2.1 HeaderBuffer() [1/2]

```
HeaderBuffer::HeaderBuffer ( )
```

Constructor to initialize [HeaderBuffer](#) without a filename.

Constructor to initialize [HeaderBuffer](#) with a filename.

#### Parameters

<i>none.</i>	
<i>filename</i>	The name of the header file to be opened as a string.

Definition at line 15 of file [HeaderBuffer.cpp](#).

```

00015         {
00016             // Set default values for member variables
00017             fileStructureType_ = "DefaultType";
00018             fileStructureVersion_ = "0.0";
00019             headerSizeBytes_ = 0;
00020             recordSizeBytes_ = 0;
00021             sizeFormatType_ = "ASCII";
00022             blockSize_ = 0;
00023             minimumBlockCapacity_ = 0;
00024             primaryKeyIndexFileName_ = "default_index.txt";
00025             primaryKeyIndexFileSchema_ = "default_schema";
00026             recordCount_ = 0;
00027             blockCount_ = 0;
00028             fieldCount_ = 0;
00029             primaryKeyFieldIndex_ = 0;
00030             RBNA_ = 0;
00031             RBNS_ = 0;
00032             staleFlag_ = 0;
00033
00034             // Add some default fields
00035             Field defaultField;
00036             defaultField.zipCode = "default_zip";
00037             defaultField.placeName = "default_place";
00038             defaultField.state = "default_state";
00039             defaultField.county = "default_county";
00040             defaultField.latitude = "default_latitude";
00041             defaultField.longitude = "default_longitude";
00042
00043             fields_.push_back(defaultField);
00044         }

```

### 3.5.2.2 HeaderBuffer() [2/2]

```

HeaderBuffer::HeaderBuffer (
    const std::string & filename )

```

Constructor to initialize [HeaderBuffer](#) with a filename.

#### Parameters

<i>filename</i>	The name of the header file to be opened as a string.
-----------------	---

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

```

00048                                     : filename_(filename) {
00049     }

```

## 3.5.3 Member Function Documentation

### 3.5.3.1 writeHeader()

```

void HeaderBuffer::writeHeader ( )

```

Write the header data to a file held the by object.

Write the header data to a file.

#### Precondition

The file must be successfully opened for writing.

Used for updating the file in the object

## Precondition

The file must be successfully opened for writing.

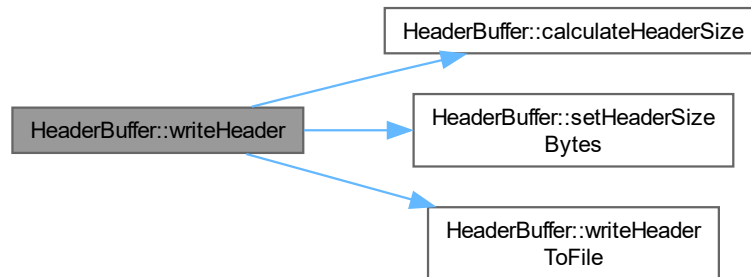
Definition at line 53 of file [HeaderBuffer.cpp](#).

```

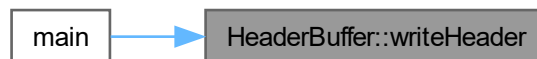
00053         {
00054             const std::string tempFilename = "tempfile.txt";
00055
00056             // Step 1: Write the data portion to the temporary file
00057             std::ofstream tempFile(tempFilename);
00058
00059             if (!tempFile.is_open()) {
00060                 std::cerr << "Error creating temporary file." << std::endl;
00061                 return;
00062             }
00063
00064             // Open the main file
00065             std::ifstream mainFile(filename_);
00066
00067             if (!mainFile.is_open()) {
00068                 std::cerr << "Error opening main file." << std::endl;
00069                 tempFile.close();
00070                 return;
00071             }
00072
00073             // Write your data to the temporary file here
00074             std::string line;
00075             bool copyStarted = false;
00076
00077             while (std::getline(mainFile, line)) {
00078                 if (copyStarted) {
00079                     tempFile << line << std::endl;
00080                 } else if (line.find("Data:") != std::string::npos) {
00081                     copyStarted = true;
00082                 }
00083             }
00084
00085             // Close the main file and the temporary file
00086             mainFile.close();
00087             tempFile.close();
00088
00089             // Step 2: Overwrite the main file with the header
00090             this->setHeaderSizeBytes(calculateHeaderSize());
00091             writeHeaderToFile(filename_);
00092
00093             // Step 3: Append the data from the temporary file to the main file
00094             std::ifstream tempFileReader(tempFilename);
00095             std::ofstream mainFileWriter(filename_, std::ios::app); // Open the file in append mode
00096
00097             if (!tempFileReader.is_open() || !mainFileWriter.is_open()) {
00098                 std::cerr << "Error opening files." << std::endl;
00099                 tempFileReader.close();
00100                 mainFileWriter.close();
00101                 return;
00102             }
00103
00104             mainFileWriter << tempFileReader.rdbuf();
00105
00106             // Close files and remove the temporary file
00107             tempFileReader.close();
00108             mainFileWriter.close();
00109             std::remove(tempFilename.c_str());
00110         }

```

Here is the call graph for this function:



Here is the caller graph for this function:



### 3.5.3.2 writeHeaderToFile()

```
void HeaderBuffer::writeHeaderToFile (
    const std::string & filename )
```

Write the header data to a file passed to the object.

Write the header data to a file.

#### Precondition

The file must be successfully opened for writing.

Used for writing to a file different than the one in the object



**Precondition**

filename the name of the file to be written to.

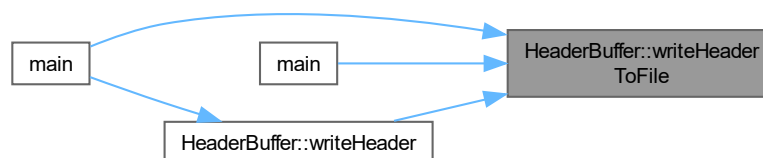
Definition at line 116 of file [HeaderBuffer.cpp](#).

```

00116                                     {
00117         std::ofstream file(filename);
00118
00119         if (!file.is_open()) {
00120             // Print an error mesage if the file cannot be opened
00121
00122             std::cerr << "Error opening the file(writeHeaderToFile)." << std::endl;
00123             return;
00124         }
00125
00126         //version for seeing all the stuff
00127         file << "Header:" << std::endl;
00128         file << " - File structure type: " << fileStructureType_ << std::endl;
00129         file << " - File structure version: " << fileStructureVersion_ << std::endl;
00130         file << " - Header Size (bytes): " << headerSizeBytes_ << std::endl;
00131         file << " - Record Size (bytes): " << recordSizeBytes_ << std::endl;
00132         file << " - Size Format Type: " << sizeFormatType_ << std::endl;
00133         file << " - Block Size: " << blockSize_ << std::endl;
00134         file << " - Minimum Block Capacity: " << minimumBlockCapacity_ << std::endl;
00135         file << " - Primary Key Index File: " << primaryKeyIndexFileName_ << std::endl;
00136         file << " - Primary Key Index File Schema: " << primaryKeyIndexFileSchema_ << std::endl;
00137         file << " - Record Count: " << recordCount_ << std::endl;
00138         file << " - Block Count: " << blockCount_ << std::endl;
00139         file << " - Field Count: " << fieldCount_ << std::endl;
00140         file << " - Primary Key: " << primaryKeyFieldIndex_ << std::endl;
00141         file << " - RBN link for Avail List: " << RBNA_ << std::endl;
00142         file << " - RBN link for active sequence set List: " << RBNS_ << std::endl;
00143         file << " - Stale Flag: " << staleFlag_ << std::endl;
00144
00145         for (const Field& field : fields_) {
00146             file << std::endl;
00147             file << "Fields:" << std::endl;
00148             file << " - Zip Code: " << field.zipCode << std::endl;
00149             file << " - Place Name: " << field.placeName << std::endl;
00150             file << " - State: " << field.state << std::endl;
00151             file << " - County: " << field.county << std::endl;
00152             file << " - Latitude: " << field.latitude << std::endl;
00153             file << " - Longitude: " << field.longitude << std::endl;
00154         }
00155
00156         file << std::endl;
00157         file << "Data:" << std::endl;
00158
00159         file.close();
00160     }

```

Here is the caller graph for this function:

**3.5.3.3 readHeader()**

```
void HeaderBuffer::readHeader ( )
```

Read header data from a file.

Reader header data from a file.

## Precondition

The file must be successfully opened for reading.

Definition at line 164 of file [HeaderBuffer.cpp](#).

```

00164         {
00165             std::ifstream file(filename_);
00166
00167             if (!file.is_open()) {
00168                 // Print an error message if the file cannot be opened
00169                 std::cerr << "Error opening the file(readHeader)." << std::endl;
00170                 return;
00171             }
00172
00173             std::string line;
00174
00175             while (std::getline(file, line)) {
00176                 if (line.find(" - File structure type: ") != std::string::npos) {
00177                     fileStructureType_ = line.substr(line.find(": ") + 2);
00178                 }
00179                 else if (line.find(" - File structure version: ") != std::string::npos) {
00180                     fileStructureVersion_ = line.substr(line.find(": ") + 2);
00181                 }
00182                 else if (line.find(" - Header Size (bytes): ") != std::string::npos) {
00183                     headerSizeBytes_ = std::stoi(line.substr(line.find(": ") + 2));
00184                 }
00185                 else if (line.find(" - Record Size (bytes): ") != std::string::npos) {
00186                     recordSizeBytes_ = std::stoi(line.substr(line.find(": ") + 2));
00187                 }
00188                 else if (line.find(" - Size Format Type: ") != std::string::npos) {
00189                     sizeFormatType_ = line.substr(line.find(": ") + 2);
00190                 }
00191                 else if (line.find(" - Block Size: ") != std::string::npos) {
00192                     blockSize_ = std::stoi(line.substr(line.find(": ") + 2));
00193                 }
00194                 else if (line.find(" - Minimum Block Capacity: ") != std::string::npos) {
00195                     minimumBlockCapacity_ = std::stoi(line.substr(line.find(": ") + 2));
00196                 }
00197                 else if (line.find(" - Primary Key Index File: ") != std::string::npos) {
00198                     primaryKeyIndexFileName_ = line.substr(line.find(": ") + 2);
00199                 }
00200                 else if (line.find(" - Primary Key Index File Schema: ") != std::string::npos) {
00201                     primaryKeyIndexFileSchema_ = line.substr(line.find(": ") + 2);
00202                 }
00203                 else if (line.find(" - Record Count: ") != std::string::npos) {
00204                     recordCount_ = std::stoi(line.substr(line.find(": ") + 2));
00205                 }
00206                 else if (line.find(" - Block Count: ") != std::string::npos) {
00207                     blockCount_ = std::stoi(line.substr(line.find(": ") + 2));
00208                 }
00209                 else if (line.find(" - Field Count: ") != std::string::npos) {
00210                     fieldCount_ = std::stoi(line.substr(line.find(": ") + 2));
00211                 }
00212                 else if (line.find(" - Primary Key: ") != std::string::npos) {
00213                     primaryKeyFieldIndex_ = std::stoi(line.substr(line.find(": ") + 2));
00214                 }
00215                 else if (line.find(" - RBN link for Avail List: ") != std::string::npos) {
00216                     RBNA_ = std::stoi(line.substr(line.find(": ") + 2));
00217                 }
00218                 else if (line.find(" - RBN link for active sequence set List: ") != std::string::npos) {
00219                     RBNS_ = std::stoi(line.substr(line.find(": ") + 2));
00220                 }
00221                 else if (line.find(" - Stale Flag: ") != std::string::npos) {
00222                     staleFlag_ = std::stoi(line.substr(line.find(": ") + 2));
00223                 }
00224                 else if (line.find("Fields:") != std::string::npos) {
00225                     Field field;
00226                 }
00227             }
00228         }
00229     }
00230
00231     Field field;
00232
00233     Field field;
00234
00235     Field field;
00236
00237     Field field;
00238
00239     Field field;
00240
00241     Field field;
00242
00243     Field field;

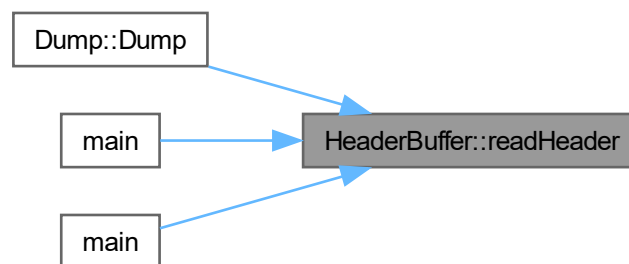
```

```

00244         while (std::getline(file, line)) {
00245             if (line.find("    - Zip Code: ") != std::string::npos) {
00246                 field.zipCode = line.substr(line.find(": ") + 2);
00247             }
00248             else if (line.find("    - Place Name: ") != std::string::npos) {
00249                 field.placeName = line.substr(line.find(": ") + 2);
00250             }
00251             else if (line.find("    - State: ") != std::string::npos) {
00252                 field.state = line.substr(line.find(": ") + 2);
00253             }
00254             else if (line.find("    - County: ") != std::string::npos) {
00255                 field.county = line.substr(line.find(": ") + 2);
00256             }
00257             else if (line.find("    - Latitude: ") != std::string::npos) {
00258                 field.latitude = line.substr(line.find(": ") + 2);
00259             }
00260             else if (line.find("    - Longitude: ") != std::string::npos) {
00261                 field.longitude = line.substr(line.find(": ") + 2);
00262             }
00263         }
00264         fields_.push_back(field);
00265     }
00266 }
00267
00268     file.close();
00269 }

```

Here is the caller graph for this function:



### 3.5.3.4 calculateHeaderSize()

```
int HeaderBuffer::calculateHeaderSize ( ) const
```

calculates the size of the header in bytes

calculates the total bytes the header will take up based on its static structure and variables

#### Precondition

values must be in the instance of headerBuffer's variables to count

the header object must have data to work with

Definition at line 273 of file [HeaderBuffer.cpp](#).

```

00273     {
00274         std::stringstream headerStream;
00275
00276         // Write header data to a stringstream

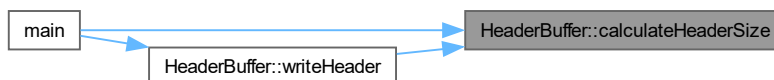
```

```

00277     headerStream << "Header:\n";
00278     headerStream << " - File structure type: " << fileStructureType_ << "\n";
00279     headerStream << " - File structure version: " << fileStructureVersion_ << "\n";
00280     headerStream << " - Header Size (bytes): " << headerSizeBytes_ << "\n";
00281     headerStream << " - Record Size (bytes): " << recordSizeBytes_ << "\n";
00282     headerStream << " - Size Format Type: " << sizeFormatType_ << "\n";
00283     headerStream << " - Block Size: " << blockSize_ << "\n";
00284     headerStream << " - Minimum Block Capacity: " << minimumBlockCapacity_ << "\n";
00285     headerStream << " - Primary Key Index File: " << primaryKeyIndexFileName_ << "\n";
00286     headerStream << " - Primary Key Index File Schema: " << primaryKeyIndexFileSchema_ << "\n";
00287     headerStream << " - Record Count: " << recordCount_ << "\n";
00288     headerStream << " - Block Count: " << blockCount_ << "\n";
00289     headerStream << " - Field Count: " << fieldCount_ << "\n";
00290     headerStream << " - Primary Key: " << primaryKeyFieldIndex_ << "\n";
00291     headerStream << " - RBN link for Avail List: " << RBNA_ << "\n";
00292     headerStream << " - RBN link for active sequence set List: " << RBNS_ << "\n";
00293     headerStream << " - Stale Flag: " << staleFlag_ << "\n";
00294
00295     headerStream << "\nFields:\n";
00296     for (const Field& field : fields_) {
00297         headerStream << " - Zip Code: " << field.zipCode << "\n";
00298         headerStream << " - Place Name: " << field.placeName << "\n";
00299         headerStream << " - State: " << field.state << "\n";
00300         headerStream << " - County: " << field.county << "\n";
00301         headerStream << " - Latitude: " << field.latitude << "\n";
00302         headerStream << " - Longitude: " << field.longitude << "\n";
00303     }
00304
00305     headerStream << "\nData:\n";
00306
00307     // Calculate total size including the newline characters
00308     return static_cast<int>(headerStream.str().size());
00309 }

```

Here is the caller graph for this function:



### 3.5.3.5 setFileStructureType()

```

void HeaderBuffer::setFileStructureType (
    const std::string & fileStructureType )

```

Setters for various header fields.

#### Parameters

<i>fileStructureType</i>	The file structure type as a string.
--------------------------	--------------------------------------

Definition at line 313 of file [HeaderBuffer.cpp](#).

```

00313
00314     fileStructureType_ = fileStructureType;
00315 }

```

### 3.5.3.6 setFileStructureVersion()

```

void HeaderBuffer::setFileStructureVersion (
    const std::string & fileStructureVersion )

```

## Parameters

<i>fileStructureVersion</i>	The file structure version as a string.
-----------------------------	---

Definition at line 318 of file [HeaderBuffer.cpp](#).

```
00318                                     {
00319         fileStructureVersion_ = fileStructureVersion;
00320     }
```

### 3.5.3.7 setHeaderSizeBytes()

```
void HeaderBuffer::setHeaderSizeBytes (
    int headerSizeBytes )
```

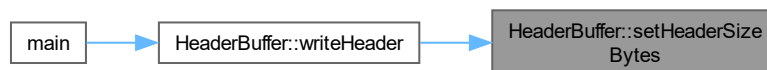
## Parameters

<i>headerSizeBytes</i>	The header size in bytes as an integer.
------------------------	---

Definition at line 323 of file [HeaderBuffer.cpp](#).

```
00323                                     {
00324         headerSizeBytes_ = headerSizeBytes;
00325     }
```

Here is the caller graph for this function:



### 3.5.3.8 setRecordSizeBytes()

```
void HeaderBuffer::setRecordSizeBytes (
    int recordSizeBytes )
```

## Parameters

<i>recordSizeBytes</i>	The record size in bytes as an integer.
------------------------	---

Definition at line 328 of file [HeaderBuffer.cpp](#).

```
00328                                     {
00329         recordSizeBytes_ = recordSizeBytes;
00330     }
```

### 3.5.3.9 setSizeFormatType()

```
void HeaderBuffer::setSizeFormatType (
    const std::string & sizeFormatType )
```

## Parameters

<i>sizeFormatType</i>	The size format type as a string (ASCII or binary).
-----------------------	---

Definition at line 333 of file [HeaderBuffer.cpp](#).

```
00333                                     {
00334         sizeFormatType_ = sizeFormatType;
00335     }
```

**3.5.3.10 setBlockSize()**

```
void HeaderBuffer::setBlockSize (
    int blockSize )
```

## Parameters

<i>blockSize</i>	The size of the blocks.
------------------	-------------------------

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

```
00338                                     {
00339         blockSize_ = blockSize;
00340     }
```

**3.5.3.11 setminimumBlockCapacity()**

```
void HeaderBuffer::setminimumBlockCapacity (
    int minimumBlockCapacity )
```

## Parameters

<i>minimumBlockCapacity</i>	The smallest amount of a block that can be filled.
-----------------------------	--

Definition at line 343 of file [HeaderBuffer.cpp](#).

```
00343                                     {
00344         minimumBlockCapacity_ = minimumBlockCapacity;
00345     }
```

**3.5.3.12 setPrimaryKeyIndexFileName()**

```
void HeaderBuffer::setPrimaryKeyIndexFileName (
    const std::string & primaryKeyIndexFileName )
```

## Parameters

<i>primaryKeyIndexFileName</i>	The primary key index file name as a string.
--------------------------------	--

Definition at line 348 of file [HeaderBuffer.cpp](#).

```
00348                                     {
00349         primaryKeyIndexFileName_ = primaryKeyIndexFileName;
00350     }
```

**3.5.3.13 setprimaryKeyIndexFileSchema()**

```
void HeaderBuffer::setprimaryKeyIndexFileSchema (
    const std::string & primaryKeyIndexFileSchema )
```

**Parameters**

<i>primaryKeyIndexFileSchema</i>	The info on how to read the index file.
----------------------------------	---

Definition at line 353 of file [HeaderBuffer.cpp](#).

```
00353                                     {
00354     primaryKeyIndexFileSchema_ = primaryKeyIndexFileSchema;
00355 }
```

**3.5.3.14 setRecordCount()**

```
void HeaderBuffer::setRecordCount (
    int recordCount )
```

**Parameters**

<i>recordCount</i>	The record count as an integer.
--------------------	---------------------------------

Definition at line 358 of file [HeaderBuffer.cpp](#).

```
00358                                     {
00359     recordCount_ = recordCount;
00360 }
```

**3.5.3.15 setBlockCount()**

```
void HeaderBuffer::setBlockCount (
    int blockCount )
```

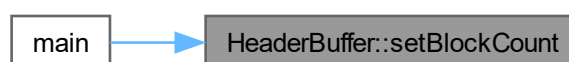
**Parameters**

<i>blockCount</i>	The block count as an integer.
-------------------	--------------------------------

Definition at line 363 of file [HeaderBuffer.cpp](#).

```
00363                                     {
00364     blockCount_ = blockCount;
00365 }
```

Here is the caller graph for this function:



### 3.5.3.16 setFieldCount()

```
void HeaderBuffer::setFieldCount (
    int fieldCount )
```

#### Parameters

<i>fieldCount</i>	The field count as an integer.
-------------------	--------------------------------

Definition at line 368 of file [HeaderBuffer.cpp](#).

```
00368                                     {
00369     fieldCount_ = fieldCount;
00370 }
```

### 3.5.3.17 setPrimaryKeyFieldIndex()

```
void HeaderBuffer::setPrimaryKeyFieldIndex (
    int primaryKeyFieldIndex )
```

#### Parameters

<i>primaryKeyFieldIndex</i>	The primary key field index as an integer.
-----------------------------	--

Definition at line 373 of file [HeaderBuffer.cpp](#).

```
00373                                     {
00374     primaryKeyFieldIndex_ = primaryKeyFieldIndex;
00375 }
```

### 3.5.3.18 setRBNA()

```
void HeaderBuffer::setRBNA (
    int RBNA )
```

#### Parameters

<i>RBNA</i>	The RBNA as an integer.
-------------	-------------------------

Definition at line 378 of file [HeaderBuffer.cpp](#).

```
00378                                     {
00379     RBNA_ = RBNA;
00380 }
```

### 3.5.3.19 setRBNS()

```
void HeaderBuffer::setRBNS (
    int RBNS )
```

#### Parameters

<i>RBNS</i>	The RBNA as an integer.
-------------	-------------------------



Definition at line 383 of file [HeaderBuffer.cpp](#).

```
00383                                     {
00384         RBNS_ = RBNS;
00385     }
```

### 3.5.3.20 setstaleFlag()

```
void HeaderBuffer::setstaleFlag (
    int staleFlag )
```

#### Parameters

<i>staleFlag</i>	The tells if the header record is stale.
------------------	--

Definition at line 388 of file [HeaderBuffer.cpp](#).

```
00388                                     {
00389         staleFlag_ = staleFlag;
00390     }
```

### 3.5.3.21 addField()

```
void HeaderBuffer::addField (
    const Field & field )
```

Add a field to the header.

#### Parameters

<i>field</i>	The <a href="#">Field</a> structure to be added to the header.
--------------	--

Definition at line 394 of file [HeaderBuffer.cpp](#).

```
00394                                     {
00395         fields_.push_back(field);
00396     }
```

### 3.5.3.22 getFileStructureType()

```
std::string HeaderBuffer::getFileStructureType ( ) const
```

Getters for header fields.

Definition at line 399 of file [HeaderBuffer.cpp](#).

```
00399                                     {
00400         return fileStructureType_;
00401     }
```

### 3.5.3.23 getFileStructureVersion()

```
std::string HeaderBuffer::getFileStructureVersion ( ) const
```

Definition at line 403 of file [HeaderBuffer.cpp](#).

```
00403                                     {
00404         return fileStructureVersion_;
00405     }
```

### 3.5.3.24 getHeaderSizeBytes()

```
int HeaderBuffer::getHeaderSizeBytes ( ) const
```

Definition at line 407 of file [HeaderBuffer.cpp](#).

```
00407 {
00408     return headerSizeBytes_;
00409 }
```

### 3.5.3.25 getRecordSizeBytes()

```
int HeaderBuffer::getRecordSizeBytes ( ) const
```

Definition at line 411 of file [HeaderBuffer.cpp](#).

```
00411 {
00412     return recordSizeBytes_;
00413 }
```

### 3.5.3.26 getSizeFormatType()

```
std::string HeaderBuffer::getSizeFormatType ( ) const
```

Definition at line 415 of file [HeaderBuffer.cpp](#).

```
00415 {
00416     return sizeFormatType_;
00417 }
```

### 3.5.3.27 getBlockSize()

```
int HeaderBuffer::getBlockSize ( ) const
```

Definition at line 419 of file [HeaderBuffer.cpp](#).

```
00419 {
00420     return blockSize_;
00421 }
```

### 3.5.3.28 getMinimumBlockCapacity()

```
int HeaderBuffer::getMinimumBlockCapacity ( ) const
```

Definition at line 423 of file [HeaderBuffer.cpp](#).

```
00423 {
00424     return minimumBlockCapacity_;
00425 }
```

### 3.5.3.29 getPrimaryKeyIndexFileName()

```
std::string HeaderBuffer::getPrimaryKeyIndexFileName ( ) const
```

Definition at line 430 of file [HeaderBuffer.cpp](#).

```
00430 {
00431     return primaryKeyIndexFileName_;
00432 }
```

### 3.5.3.30 getRecordCount()

```
int HeaderBuffer::getRecordCount ( ) const
```

Definition at line 434 of file [HeaderBuffer.cpp](#).

```
00434 {  
00435     return recordCount_;  
00436 }
```

### 3.5.3.31 getBlockCount()

```
int HeaderBuffer::getBlockCount ( ) const
```

Definition at line 427 of file [HeaderBuffer.cpp](#).

```
00427 {  
00428     return blockCount_;  
00429 }
```

Here is the caller graph for this function:



### 3.5.3.32 getFieldCount()

```
int HeaderBuffer::getFieldCount ( ) const
```

Definition at line 438 of file [HeaderBuffer.cpp](#).

```
00438 {  
00439     return fieldCount_;  
00440 }
```

### 3.5.3.33 getPrimaryKeyFieldIndex()

```
int HeaderBuffer::getPrimaryKeyFieldIndex ( ) const
```

Definition at line 442 of file [HeaderBuffer.cpp](#).

```
00442 {  
00443     return primaryKeyFieldIndex_;  
00444 }
```

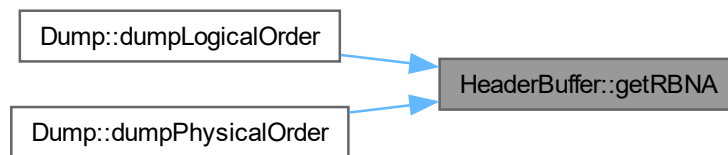
### 3.5.3.34 getRBNA()

```
int HeaderBuffer::getRBNA ( ) const
```

Definition at line 446 of file [HeaderBuffer.cpp](#).

```
00446 {  
00447     return RBNA_;  
00448 }
```

Here is the caller graph for this function:



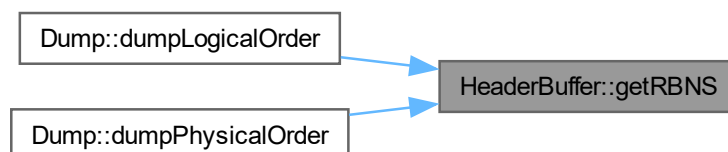
### 3.5.3.35 getRBNS()

```
int HeaderBuffer::getRBNS ( ) const
```

Definition at line 450 of file [HeaderBuffer.cpp](#).

```
00450 {  
00451     return RBNS_;  
00452 }
```

Here is the caller graph for this function:



### 3.5.3.36 getStaleFlag()

```
int HeaderBuffer::getStaleFlag ( ) const
```

Definition at line 454 of file [HeaderBuffer.cpp](#).

```
00454 {  
00455     return staleFlag_;  
00456 }
```

## 3.5.3.37 getFields()

```
const std::vector< Field > & HeaderBuffer::getFields ( ) const
```

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

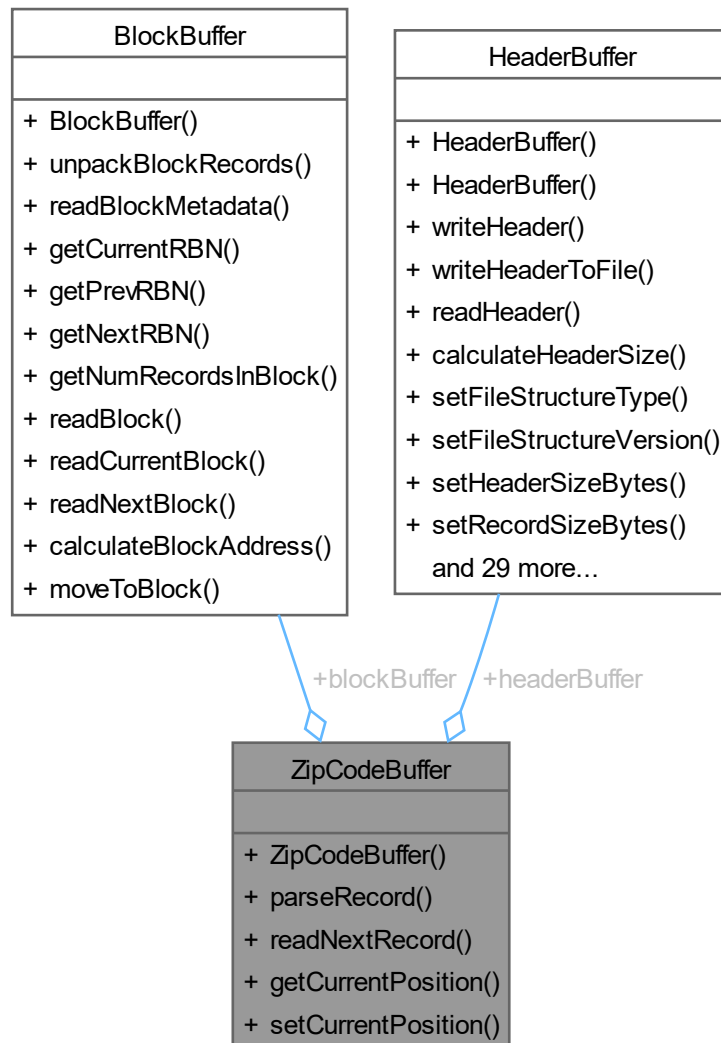
- [HeaderBuffer.h](#)
- [HeaderBuffer.cpp](#)

## 3.6 ZipCodeBuffer Class Reference

The [ZipCodeBuffer](#) class parses the file one record at a time and returns the fields in a [ZipCodeRecord](#) struct.

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

Collaboration diagram for ZipCodeBuffer:



## Public Member Functions

- [ZipCodeBuffer](#) (std::ifstream &file, char fileType, [HeaderBuffer](#) headerBuffer)  
*Constructor that accepts the filename.*
- [ZipCodeRecord](#) [parseRecord](#) (std::string)  
*Parses a string into a [ZipCodeRecord](#) struct.*
- [ZipCodeRecord](#) [readNextRecord](#) ()  
*Reads the next ZIP Code record from the file.*
- std::streampos [getCurrentPosition](#) ()  
*Method to get the current position in the file.*
- std::ifstream & [setCurrentPosition](#) (std::streampos)  
*Method to set the current position in the file to a given streampos.*

## Public Attributes

- [BlockBuffer](#) [blockBuffer](#)
- [HeaderBuffer](#) [headerBuffer](#) = [HeaderBuffer](#)("us\_postal\_codes.txt")

## Friends

- class [Dump](#)

## 3.6.1 Detailed Description

The [ZipCodeBuffer](#) class parses the file one record at a time and returns the fields in a [ZipCodeRecord](#) struct.

Class to parse ZIP code records in a file.

See [ZipCodeBuffer.h](#) for full documentation.

### Author

Kent Biernath  
Emma Hoffmann, Emily Yang

### Date

2023-11-19

### Version

3.0

Definition at line 60 of file [ZipCodeBuffer.h](#).

## 3.6.2 Constructor & Destructor Documentation

### 3.6.2.1 ZipCodeBuffer()

```
ZipCodeBuffer::ZipCodeBuffer (
    std::ifstream & file,
    char fileType = 'L',
    HeaderBuffer headerBuffer = HeaderBuffer("us_postal_codes.txt") )
```

Constructor that accepts the filename.

#### Precondition

The file has one column header row to skip.

#### Postcondition

The file is opened and the header row is skipped.

#### Parameters

<i>fileName</i>	The name of the file to open.
<i>fileType</i>	The type of the file. Case insensitive, stored in uppercase.
<i>headerBuffer</i>	A <a href="#">HeaderBuffer</a> object for the file. – 'C' = CSV, comma-separated values. – 'L' = Length-indicated file structure format with the first field describing the length of the record. – 'B' = Blocked length-indicated records.

Definition at line 15 of file [ZipCodeBuffer.cpp](#).

```
00015 : file(file),
00016     fileType(std::toupper(fileType)), blockBuffer(BlockBuffer(file, headerBuffer)),
    headerBuffer(headerBuffer) { // TODO change HeaderBuffer initialization once it has a generic
    constructor
00017
00018     if (this->fileType == 'C') {
00019         // If CSV, skip the header line.
00020         std::string line;
00021         getline(file, line); // Skip header line
00022     }
00023     else if (this->fileType == 'L' || this->fileType == 'B') {
00024         // If length-indicated or blocked length-indicated file, skip the header line.
00025         // We have to skip past the metadata, up to the "Data: line"
00026         std::string line;
00027         std::getline(file, line);
00028         if (line.find("Header:") != std::string::npos)
00029         {
00030             // File contains a metadata header
00031
00032             // Read lines until "Data:" is found
00033             while (std::getline(file, line)) {
00034                 if (line.find("Data:") != std::string::npos) {
00035                     break;
00036                 }
00037             }
00038         }
00039     }
00040 }
00041 ;;
```

### 3.6.3 Member Function Documentation

#### 3.6.3.1 parseRecord()

```
ZipCodeRecord ZipCodeBuffer::parseRecord (
    std::string recordString )
```

Parses a string into a [ZipCodeRecord](#) struct.

##### Precondition

Receives a string to parse.

##### Postcondition

The [ZipCodeRecord](#) struct is filled with data from the string and returned.

##### Parameters

<i>recordString</i>	<p>The string to parse into a <a href="#">ZipCodeRecord</a> struct. It must have six fields separated by commas and be in this order:</p> <ul style="list-style-type: none"> <li>– ZIP Code (string)</li> <li>– Place Name (string)</li> <li>– State Code (string)</li> <li>– County (string)</li> <li>– Latitude (double)</li> <li>– Longitude (double)</li> </ul>
---------------------	---

##### Returns

Returns the [ZipCodeRecord](#) struct filled with data parsed from the string.  
If the record string is malformed, it returns an empty string for the zipCode field as the terminal string.

Definition at line 45 of file [ZipCodeBuffer.cpp](#).

```
00045                                     {
00046     ZipCodeRecord record;
00047
00048     std::istringstream recordStream(recordString);
00049     std::string field;
00050
00051     // Parse the record fields using istringstream
00052     std::vector<std::string> fields;
00053     while (getline(recordStream, field, ',')) {
00054         fields.push_back(field);
00055     }
00056
00057     if (fields.size() == 6)
00058     {
00059         // Fill the ZipCodeRecord struct with the data from the record
00060         record.zipCode = fields[0];
00061         record.placeName = fields[1];
00062         record.state = fields[2];
00063         record.county = fields[3];
00064         record.latitude = std::stod(fields[4]); // Convert string to double
00065         record.longitude = std::stod(fields[5]);
00066     }
00067     else
00068     {
00069         // The record is malformed and does not have 6 fields. Return terminal string
00070         record.zipCode = "";
00071         std::cerr << "A record contains an invalid number of fields: "
00072                 << recordString << std::endl;
00073     }
```

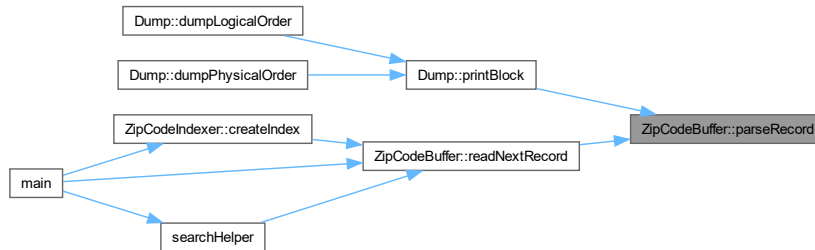


```

00074
00075     return record;
00076 };

```

Here is the caller graph for this function:



### 3.6.3.2 readNextRecord()

```
ZipCodeRecord ZipCodeBuffer::readNextRecord ( )
```

Reads the next ZIP Code record from the file.

This function reads the next line from the file, parses it into a [ZipCodeRecord](#) struct, and returns it.

#### Precondition

The next record must have exactly six fields.

#### Postcondition

The next record in the file was returned.

#### Returns

The next ZIP Code record from the file. When it reaches the end of the file or an invalid record, it returns a ZIP code of "" as a terminal string.

Definition at line 80 of file [ZipCodeBuffer.cpp](#).

```

00080                                     {
00081     ZipCodeRecord record;
00082     std::string recordString;
00083
00084     if (file.eof())
00085     {
00086         // End of file reached. Return terminal character
00087         record.zipCode = "";
00088         return record;
00089     }
00090
00091     if (fileType == 'B')
00092     {
00093         if (blockRecordsIndex >= blockRecords.size() || blockRecordsIndex == -1)
00094         {
00095             // Reached the end of the block, so retrieve the next one
00096             blockRecords = blockBuffer.readNextBlock();
00097             if (blockRecords.size() > 0)
00098             {
00099                 recordString = blockRecords[0]; // Retrieve the first record in the block

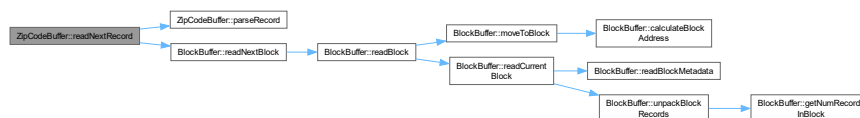
```

```

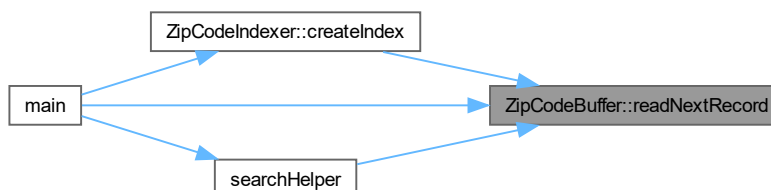
00100         blockRecordsIndex = 1;           // Skip 0 because it reads it immediately
00101     }
00102     else
00103     {
00104         // Did not read a valid block (likely due to the end of file), so return terminal
character
00105         record.zipCode = "";
00106         return record;
00107     }
00108 }
00109 else
00110 {
00111     recordString = blockRecords[blockRecordsIndex++];
00112 }
00113 }
00114 else if (fileType == 'C')
00115 {
00116     // If CSV, retrieve the next line in the file as the record to parse
00117     getline(file, recordString);
00118 }
00119 else if (fileType == 'L')
00120 {
00121     // If length-indicated, reads the length and retrieves that many characters for the record
string
00122     int numCharactersToRead = 0;
00123     file >> numCharactersToRead; // Read the length indicator, the first field in each record
00124     file.ignore(1);             // Skip the comma after the length field
00125     recordString.resize(numCharactersToRead);
00126     file.read(&recordString[0], numCharactersToRead);
00127 }
00128
00129 // If not the end of the file, read the fields in the line into the record object
00130 if (recordString.empty())
00131 {
00132     // Did not read a valid record (likely due to the end of file newline), so return terminal
character
00134     record.zipCode = "";
00135     return record;
00136 }
00137
00138 record = parseRecord(recordString);
00139 return record;
00140 };

```

Here is the call graph for this function:



Here is the caller graph for this function:



### 3.6.3.3 getCurrentPosition()

```
std::streampos ZipCodeBuffer::getCurrentPosition ( )
```

Method to get the current position in the file.

Definition at line 143 of file [ZipCodeBuffer.cpp](#).

```
00143 {
00144     return file.tellg();
00145 }
```

Here is the caller graph for this function:



### 3.6.3.4 setCurrentPosition()

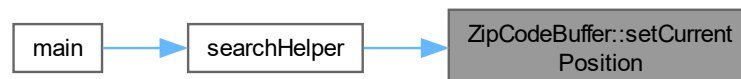
```
std::ifstream & ZipCodeBuffer::setCurrentPosition (
    std::streampos pos )
```

Method to set the current position in the file to a given streampos.

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

```
00148 {
00149     file.seekg(pos);
00150     return file;
00151 }
```

Here is the caller graph for this function:



## 3.6.4 Friends And Related Symbol Documentation

### 3.6.4.1 Dump

```
friend class Dump [friend]
```

Definition at line 132 of file [ZipCodeBuffer.h](#).

### 3.6.5 Member Data Documentation

#### 3.6.5.1 blockBuffer

`BlockBuffer` ZipCodeBuffer::blockBuffer

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

#### 3.6.5.2 headerBuffer

`HeaderBuffer` ZipCodeBuffer::headerBuffer = `HeaderBuffer`("us\_postal\_codes.txt")

Definition at line 69 of file [ZipCodeBuffer.h](#).

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

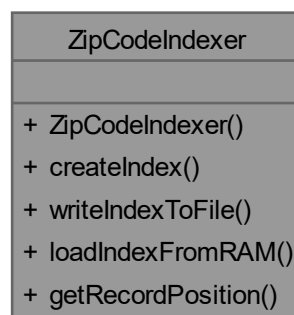
- [ZipCodeBuffer.h](#)
- [ZipCodeBuffer.cpp](#)

## 3.7 ZipCodeIndexer Class Reference

Implementation of the [ZipCodeIndexer](#) class for indexing ZIP code records in a file.

```
#include <ZipCodeIndexer.h>
```

Collaboration diagram for ZipCodeIndexer:



## Public Member Functions

- [ZipCodeIndexer](#) (std::ifstream &file, char fileType, const std::string &idxFileName, [HeaderBuffer](#) headerBuffer)  
*Constructor: initializes the [ZipCodeIndexer](#) with a file name and index file name.*
- void [createIndex](#) ()  
*Method to create an index by reading the file and storing ZIP codes and their positions.*
- void [writeIndexToFile](#) ()  
*Method to write the created index to a file.*
- void [loadIndexFromRAM](#) ()  
*Method to load the index from a file into RAM.*
- std::streampos [getRecordPosition](#) (const std::string &zipCode)  
*Method to get the position of a specific ZIP code record in the file.*

### 3.7.1 Detailed Description

Implementation of the [ZipCodeIndexer](#) class for indexing ZIP code records in a file.

< For std::map container

Class for indexing ZIP code records from a file.

#### Author

Emily Yang  
Kent Biernath  
Emma Hoffmann

#### Date

2023-10-16

#### Version

1.0

The [ZipCodeIndexer](#) class is responsible for creating and managing an index of ZIP code records in a file. It provides methods for creating the index, saving it to a file, loading it into RAM, and retrieving the position of a specific ZIP code record in the file.

#### Assumptions:

– The file is in the same directory as the program. < For file operations < For accessing the [ZipCodeBuffer](#) class

Class for indexing ZIP code records from a file.

The [ZipCodeIndexer](#) class is responsible for creating and managing an index of ZIP code records in a file. It provides methods for creating the index, saving it to a file, loading it into RAM, and retrieving the position of a specific ZIP code record in the file.

Definition at line 49 of file [ZipCodeIndexer.h](#).

### 3.7.2 Constructor & Destructor Documentation

#### 3.7.2.1 ZipCodeIndexer()

```
ZipCodeIndexer::ZipCodeIndexer (
    std::ifstream & file,
    char fileType,
    const std::string & idxFileName,
    HeaderBuffer headerBuffer )
```

Constructor: initializes the [ZipCodeIndexer](#) with a file name and index file name.

Constructor for the [ZipCodeIndexer](#) class.

## Parameters

<i>fileName</i>	The name of the file to index as a string.
<i>fileType</i>	The type of the file, [C]SV or [L]ength-indicated
<i>idxFileName</i>	The name of the index file to save/load as a string.
<i>headerBuffer</i>	A <a href="#">HeaderBuffer</a> object for the file.
<i>fileName</i>	The name of the file to index as a string.
<i>idxFileName</i>	The name of the index file to save/load as a string.

Definition at line 13 of file [ZipCodeIndexer.cpp](#).

```
00014      : buffer(file, fileType, headerBuffer), indexFileName(idxFileName) {}
```

### 3.7.3 Member Function Documentation

#### 3.7.3.1 createIndex()

```
void ZipCodeIndexer::createIndex ( )
```

Method to create an index by reading the file and storing ZIP codes and their positions.

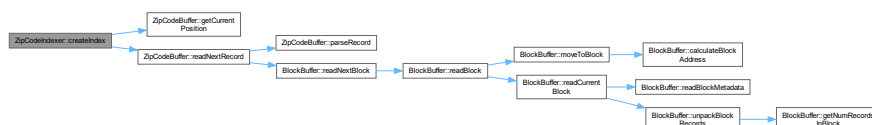
Create an index of ZIP codes to their positions in the file.

This method reads the file and creates an index of ZIP code records by storing each ZIP code and its position in the file.

Definition at line 19 of file [ZipCodeIndexer.cpp](#).

```
00019      {
00020      ZipCodeRecord record;
00021      std::streampos position = buffer.getCurrentPosition();
00022      while (!record = buffer.readNextRecord()).zipCode.empty() {
00023          index[record.zipCode] = position; // Save the position of this ZIP code in the index
00024          position = buffer.getCurrentPosition(); // Get the position of the next record
00025      }
00026 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



### 3.7.3.2 writeIndexToFile()

```
void ZipCodeIndexer::writeIndexToFile ( )
```

Method to write the created index to a file.

Write the created index to a file.

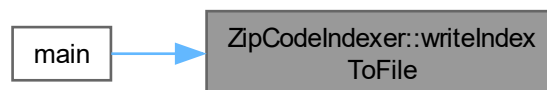
This method saves the created index to a file specified by the index file name.

This function writes the created index to a file. Each line in the file contains a ZIP code and its position in the file.

Definition at line 30 of file [ZipCodeIndexer.cpp](#).

```
00030 {
00031     std::ofstream outFile(indexFileName);
00032     for (const auto& pair : index) {
00033         outFile << pair.first << " " << pair.second << "\n"; // ZIP code and position
00034     }
00035     outFile.close();
00036 }
```

Here is the caller graph for this function:



### 3.7.3.3 loadIndexFromRAM()

```
void ZipCodeIndexer::loadIndexFromRAM ( )
```

Method to load the index from a file into RAM.

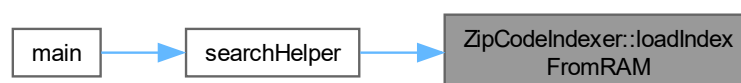
Load the index from a file into RAM.

This method loads the index data from a file into RAM for quick retrieval.

Definition at line 41 of file [ZipCodeIndexer.cpp](#).

```
00041 {
00042     index.clear(); // Clear any existing index
00043     std::ifstream inFile(indexFileName);
00044     std::string zip;
00045     std::streampos pos;
00046     int posInt;
00047     while (inFile >> zip >> posInt) {
00048         pos = posInt;
00049         index[zip] = pos; // Load the ZIP code and its position into the index
00050     }
00051     inFile.close();
00052 }
```

Here is the caller graph for this function:



### 3.7.3.4 getRecordPosition()

```
std::streampos ZipCodeIndexer::getRecordPosition (
    const std::string & zipCode )
```

Method to get the position of a specific ZIP code record in the file.

Get the position in the file of the given ZIP code.

#### Parameters

<i>zipCode</i>	The ZIP code to find the position of.
----------------	---------------------------------------

#### Returns

The position of the ZIP code record in the file.

#### Parameters

<i>zipCode</i>	The ZIP code to find the position of.
----------------	---------------------------------------

#### Returns

The position of the ZIP code record in the file. If not found, returns an invalid position (-1).

Definition at line 58 of file [ZipCodeIndexer.cpp](#).

```
00058
00059     if (index.find(zipCode) != index.end()) { // If the ZIP code is in the index
00060         return index[zipCode]; // Return its position
00061     }
00062     else {
00063         return std::streampos(-1); // Invalid position to indicate not found
00064     }
00065 }
```

Here is the caller graph for this function:



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

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

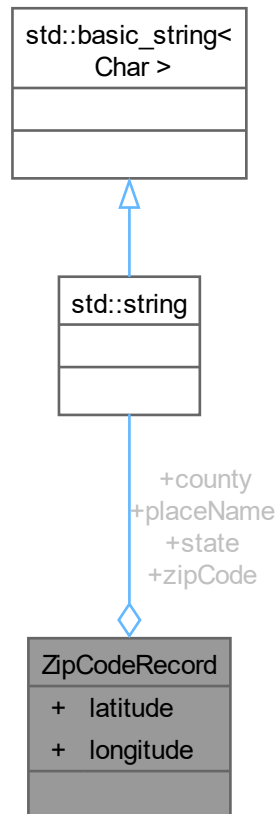


## 3.8 ZipCodeRecord Struct Reference

Structure to hold a ZIP Code record.

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

Collaboration diagram for ZipCodeRecord:



### Public Attributes

- `std::string` `zipCode`
- `std::string` `placeName`
- `std::string` `state`
- `std::string` `county`
- `double` `latitude` = 0.0
- `double` `longitude` = 0.0

### 3.8.1 Detailed Description

Structure to hold a ZIP Code record.

The [ZipCodeBuffer](#) class reads a record from a file with these six fields:

- ZIP Code (string)
- Place Name (string)
- State Code (string)
- County (string)
- Latitude (double)
- Longitude (double)

Whenever `readNextRecord` is called, it reads the next record from the file and returns it in a [ZipCodeRecord](#) struct after parsing it with `parseRecord`.

The name of the file to be opened is passed to the class constructor as a string.

Assumptions:

- The file is in the same directory as the program.
- The records always contain exactly six fields.
- The file has column headers on the first line.

Definition at line 49 of file [ZipCodeBuffer.h](#).

### 3.8.2 Member Data Documentation

#### 3.8.2.1 zipCode

```
std::string ZipCodeRecord::zipCode
```

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

#### 3.8.2.2 placeName

```
std::string ZipCodeRecord::placeName
```

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

#### 3.8.2.3 state

```
std::string ZipCodeRecord::state
```

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

#### 3.8.2.4 county

```
std::string ZipCodeRecord::county
```

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

#### 3.8.2.5 latitude

```
double ZipCodeRecord::latitude = 0.0
```

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

#### 3.8.2.6 longitude

```
double ZipCodeRecord::longitude = 0.0
```

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

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

- [ZipCodeBuffer.h](#)



## Chapter 4

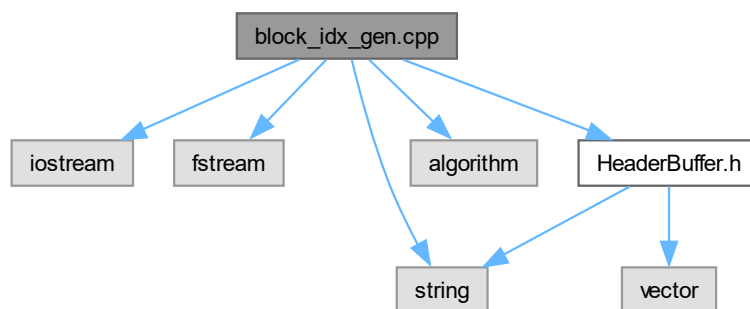
# File Documentation

### 4.1 block\_idx\_gen.cpp File Reference

This class creates an index file for a blocked data file.

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include "HeaderBuffer.h"
```

Include dependency graph for block\_idx\_gen.cpp:



#### Functions

- int `findZipcode` (const string &record)
- int `main` ()

### 4.1.1 Detailed Description

This class creates an index file for a blocked data file.

**Author**

Andrew Clayton

**Date**

11/13/2023

**Version**

1.4

Definition in file [block\\_idx\\_gen.cpp](#).

### 4.1.2 Function Documentation

#### 4.1.2.1 findZipcode()

```
int findZipcode (
    const string & record )
```

Definition at line 27 of file [block\\_idx\\_gen.cpp](#).

```
00027     {
00028         size_t firstComma = record.find(',');
00029         size_t secondComma = record.find(',', firstComma + 1);
00030         if (firstComma == string::npos || secondComma == string::npos) {
00031             cerr << "Error parsing record for zipcode: " << record << "\n";
00032             return -1;
00033         }
00034         return stoi(record.substr(firstComma + 1, secondComma - firstComma - 1));
00035     }
```

Here is the caller graph for this function:



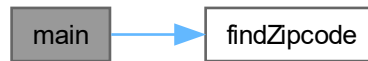
## 4.1.2.2 main()

```
int main ( )
```

Definition at line 38 of file [block\\_idx\\_gen.cpp](#).

```
00038     {
00039
00040         // We need to first read and write the header of the file
00041         /* HeaderBuffer header = HeaderBuffer("blocked_postal_codes.txt");
00042         header.readHeader();
00043         header.writeHeaderToFile("blocked_Index.txt"); */ // The blocked index file actually does
not include any metadata
00044
00045
00046         ofstream writeFile;
00047         writeFile.open("blocked_Index.txt");
00048         if (!writeFile.is_open()) {
00049             cerr << "Error: Could not open file 'blocked_Index.txt' for writing.\n";
00050             return 1;
00051         }
00052
00053         ifstream readFile("us_postal_codes_blocked.txt");
00054         if (!readFile.is_open()) {
00055             cerr << "Error: Could not open file 'us_postal_codes_blocked.txt' for reading.\n";
00056             return 1;
00057         }
00058
00059         // We need to skip past the metadata, up to the "Data: line"
00060         string line;
00061         while (getline(readFile, line)) {
00062             if (line == "Data:") {
00063                 break;
00064             }
00065         }
00066
00067         int blockNumber = 0;
00068         int maxZipcode = 0;
00069         string currentBlock;
00070
00071         while (getline(readFile, currentBlock)) {
00072             // for each line, we have to skip the metadata!
00073             int start = stoi(currentBlock.substr(0, 2));
00074             size_t endOfBlock = currentBlock.find('~'); // Assuming '~' is the padding character
00075
00076             maxZipcode = 0;
00077
00078             // Put all the records into a vector
00079             vector<string> records;
00080
00081             while (start < endOfBlock) {
00082                 size_t recordLength;
00083                 try {
00084                     recordLength = stoi(currentBlock.substr(start, 2)) + 3; // plus three since `LI,` does
not include itself
00085                 } catch (const std::invalid_argument& ia) {
00086                     cerr << "Invalid argument: " << ia.what() << " for record length at block " << blockNumber
<< "\n";
00087                     return 1;
00088                 }
00089
00090                 // Traversing record by record
00091                 string currentRecord = currentBlock.substr(start, recordLength);
00092                 records.push_back(currentRecord);
00093                 // Checking the zipcode of each individual record
00094
00095
00096                 start += recordLength;
00097             }
00098
00099             maxZipcode = findZipcode(records.back());
00100
00101             writeFile << blockNumber << "," << maxZipcode << "\n";
00102
00103             blockNumber++;
00104             // maxZipcode = 0;
00105         }
00106
00107         readFile.close();
00108         writeFile.close();
00109         return 0;
00110 }
```

Here is the call graph for this function:



## 4.2 block\_idx\_gen.cpp

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

```

00001 // -----
00009 // -----
00017 // -----
00018
00019 #include <iostream>
00020 #include <fstream>
00021 #include <string>
00022 #include <algorithm>
00023 #include "HeaderBuffer.h"
00024
00025 using namespace std;
00026
00027 int findZipcode(const string& record) {
00028     size_t firstComma = record.find(',');
00029     size_t secondComma = record.find(',', firstComma + 1);
00030     if (firstComma == string::npos || secondComma == string::npos) {
00031         cerr << "Error parsing record for zipcode: " << record << "\n";
00032         return -1;
00033     }
00034     return stoi(record.substr(firstComma + 1, secondComma - firstComma - 1));
00035 }
00036
00037
00038 int main() {
00039     // We need to first read and write the header of the file
00040     /* HeaderBuffer header = HeaderBuffer("blocked_postal_codes.txt");
00041     header.readHeader();
00042     header.writeHeaderToFile("blocked_Index.txt"); */ // The blocked index file actually does
00043     not include any metadata
00044
00045     ofstream writeFile;
00046     writeFile.open("blocked_Index.txt");
00047     if (!writeFile.is_open()) {
00048         cerr << "Error: Could not open file 'blocked_Index.txt' for writing.\n";
00049         return 1;
00050     }
00051
00052     ifstream readFile("us_postal_codes_blocked.txt");
00053     if (!readFile.is_open()) {
00054         cerr << "Error: Could not open file 'us_postal_codes_blocked.txt' for reading.\n";
00055         return 1;
00056     }
00057
00058     // We need to skip past the metadata, up to the "Data: line"
00059     string line;
00060     while (getline(readFile, line)) {
00061         if (line == "Data:") {
00062             break;
00063         }
00064     }
00065
00066     int blockNumber = 0;
00067     int maxZipcode = 0;
00068     string currentBlock;
00069
00070     while (getline(readFile, currentBlock)) {
00071         // for each line, we have to skip the metadata!
00072         int start = stoi(currentBlock.substr(0, 2));
00073         size_t endOfBlock = currentBlock.find('~'); // Assuming '~' is the padding character
00074     }
  
```



```

00075
00076     maxZipcode = 0;
00077
00078     // Put all the records into a vector
00079     vector<string> records;
00080
00081     while (start < endOfBlock) {
00082         size_t recordLength;
00083         try {
00084             recordLength = stoi(currentBlock.substr(start, 2)) + 3; // plus three since `LI,` does
not include itself
00085         } catch (const std::invalid_argument& ia) {
00086             cerr << "Invalid argument: " << ia.what() << " for record length at block " << blockNumber
<< "\n";
00087             return 1;
00088         }
00089
00090         // Traversing record by record
00091         string currentRecord = currentBlock.substr(start, recordLength);
00092         records.push_back(currentRecord);
00093         // Checking the zipcode of each individual record
00094
00095         start += recordLength;
00096     }
00097
00098     maxZipcode = findZipcode(records.back());
00099
00100     writeFile << blockNumber << "," << maxZipcode << "\n";
00101
00102     blockNumber++;
00103     // maxZipcode = 0;
00104 }
00105
00106 readFile.close();
00107 writeFile.close();
00108 return 0;
00109 }
00110 }

```

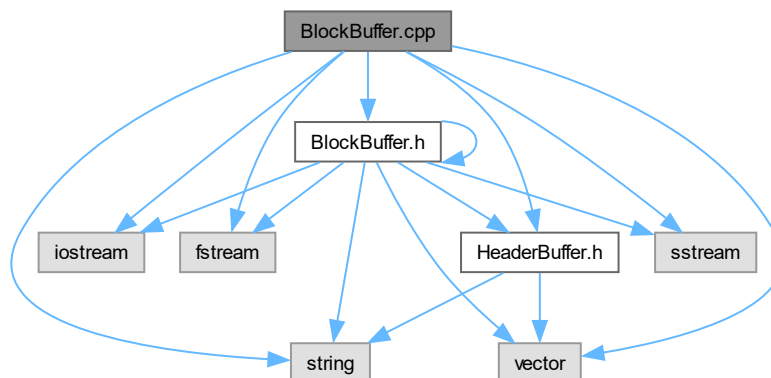
## 4.3 BlockBuffer.cpp File Reference

```

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "BlockBuffer.h"
#include "HeaderBuffer.h"
#include <sstream>

```

Include dependency graph for BlockBuffer.cpp:



## 4.4 BlockBuffer.cpp

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

```

00001
00004
00005 #include <iostream>
00006 #include <fstream> // for file operations
00007 #include <string>
00008 #include <vector>
00009 #include "BlockBuffer.h"
00010 #include "HeaderBuffer.h"
00011 #include <sstream>
00012
00013
00014
00015 BlockBuffer::BlockBuffer(std::ifstream &file, HeaderBuffer headerBuffer =
HeaderBuffer("blocked_postal_codes.txt")) : file(file) { // TODO remove HeaderBuffer filename once it
allows generic constructor
00016     headerBuffer.readHeader();
00017     headerSize = headerBuffer.getHeaderSizeBytes();
00018     blockSize = headerBuffer.getBlockSize();
00019     nextRBN = headerBuffer.getRBNS();
00020 }
00021
00022
00023 vector<string> BlockBuffer::unpackBlockRecords() {
00024     // This will convert a block to a vector of records
00025     size_t idx = 0;
00026     vector<string> records;
00027
00028     for (size_t i = 0; i < getNumRecordsInBlock(); i++)
00029     {
00030         // Reads the length and retrieves that many characters for the record
00031         std::string recordString;
00032         int numCharactersToRead = 0;
00033         file » numCharactersToRead; // Read the length indicator, the first field in each record
00034         file.ignore(1); // Skip the comma after the length field
00035         recordString.resize(numCharactersToRead);
00036         file.read(&recordString[0], numCharactersToRead);
00037         records.push_back(recordString);
00038     }
00039
00040     return records;
00041 }
00042
00043
00044
00046 void BlockBuffer::readBlockMetadata() {
00047     int metadataRecordLength = -1;
00048     int newRelativeBlockNumber = -1;
00049     int newNumRecordsInBlock = -1;
00050     int newPrevRBN = -1;
00051     int newNextRBN = -1;
00052
00053     file » metadataRecordLength;
00054     file.ignore(1); // Ignore the commas separating the fields
00055     file » newRelativeBlockNumber;
00056     file.ignore(1);
00057     file » newNumRecordsInBlock;
00058     file.ignore(1);
00059     file » newPrevRBN;
00060     file.ignore(1);
00061     file » newNextRBN;
00062     file.ignore(1); // Skip the comma after the last metadata field
00063
00064     // TODO throw exception if any of these reads failed or the values are invalid
00065
00066     currentRBN = newRelativeBlockNumber;
00067     numRecordsInBlock = newNumRecordsInBlock;
00068     prevRBN = newPrevRBN;
00069     nextRBN = newNextRBN;
00070 }
00071
00072
00073
00075 int BlockBuffer::calculateBlockAddress(int relativeBlockNumber) {
00076     return headerSize + relativeBlockNumber*blockSize;
00077 }
00078
00079
00080
00082 void BlockBuffer::moveToBlock(int relativeBlockNumber) {
00083     int address = calculateBlockAddress(relativeBlockNumber);
00084     file.seekg(address);
00085 }

```

```

00086
00087
00088
00090 vector<string> BlockBuffer::readBlock(int relativeBlockNumber) {
00091     vector<string> recordStrings;
00092     std::string line;
00093
00094     // If the RBN is -1, the end of the chain has been reached.
00095     if (relativeBlockNumber == -1)
00096     {
00097         currentRBN = -1;
00098         return recordStrings;
00099     }
00100
00101     moveToBlock(relativeBlockNumber);           // Move to the next block
00102     return readCurrentBlock();                  // Read the metadata and the records
00103 }
00104
00105
00106
00108 vector<string> BlockBuffer::readCurrentBlock() {
00109     readBlockMetadata();                        // Read the metadata for the block
00110     return unpackBlockRecords();               // Read the length-indicated records into strings and return
00111     them
00112 }
00113
00114 vector<string> BlockBuffer::readNextBlock() {
00115     return readBlock(nextRBN);
00116 }

```

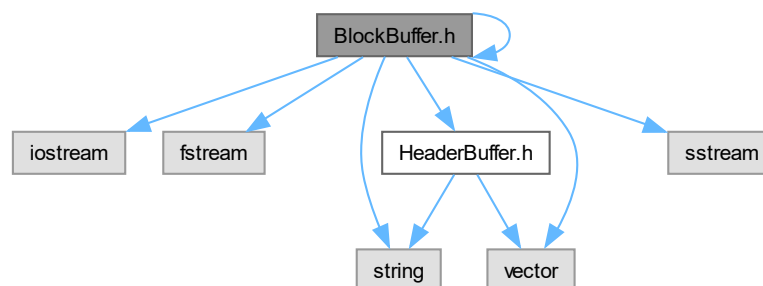
## 4.5 BlockBuffer.h File Reference

```

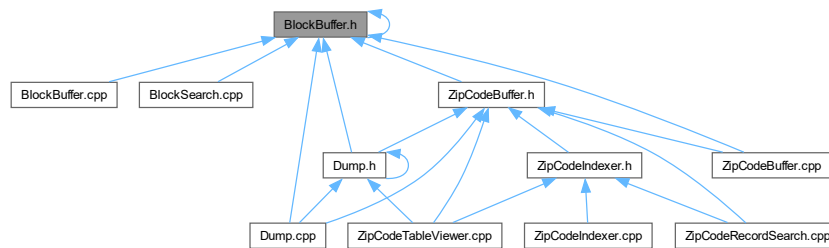
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "BlockBuffer.h"
#include "HeaderBuffer.h"
#include <sstream>

```

Include dependency graph for BlockBuffer.h:



This graph shows which files directly or indirectly include this file:



## Classes

- class [BlockBuffer](#)

See [BlockBuffer.h](#) for full documentation.

## 4.6 BlockBuffer.h

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

```

00001 // -----
00011 // -----
00033 // -----
00034
00035 #ifndef BLOCKBUFFER_H
00036 #define BLOCKBUFFER_H
00037
00038 #include <iostream>
00039 #include <fstream> // for file operations
00040 #include <string>
00041 #include <vector>
00042 #include "BlockBuffer.h"
00043 #include "HeaderBuffer.h"
00044 #include <sstream>
00045
00046 using namespace std;
00047
00048 class BlockBuffer {
00049 private:
00050     std::ifstream &file; // The ifstream to read blocks from.
00051     int numRecordsInBlock = 0; // Number of records in the current block (read from metadata)
00052     int currentRBN = 0; // Relative Block Number (RBN) of the current block
00053     int prevRBN = -1; // RBN of the previous block in the linked list
00054     int nextRBN = 0; // RBN of the next block in the linked list
00055     int blockSize = 512; // Number of bytes in every block, which will be read from the
00056     metadata
00057     int headerSize = 53; // Number of bytes in the metadata header record, which will be read
00058     from the metadata
00059
00060 public:
00061     BlockBuffer(std::ifstream &file, HeaderBuffer headerBuffer);
00062     //BlockBuffer(std::ifstream &file) : BlockBuffer(file, HeaderBuffer("blocked_postal_codes.txt"))
00063     {} // TODO replace hardcoded file name once HeaderBuffer allows generic constructor
00064
00065     vector<string> unpackBlockRecords();
00066
00067     void readBlockMetadata();
00068
00069     // Metadata getters
00070     int getCurrentRBN() const { return currentRBN; }
00071     int getPrevRBN() const { return prevRBN; }
00072     int getNextRBN() const { return nextRBN; }
00073     int getNumRecordsInBlock() const { return numRecordsInBlock; }
00074
00075     vector<string> readBlock(int relativeBlockNumber);

```

```

00101
00109     vector<string> readCurrentBlock();
00110
00118     vector<string> readNextBlock();
00119     //vector<string> readPreviousBlock();
00120
00121
00128     int calculateBlockAddress(int relativeBlockNumber);
00129
00130
00136     void moveToBlock(int relativeBlockNumber);
00137
00138 };
00139
00140 // #include "BlockBuffer.cpp"
00141
00142 #endif

```

## 4.7 BlockGenerator.cpp File Reference

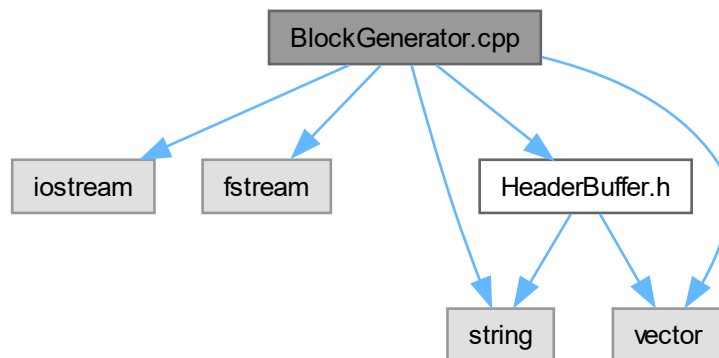
(Blocked Sequence Set Generator)

```

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "HeaderBuffer.h"

```

Include dependency graph for BlockGenerator.cpp:



### Functions

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

#### 4.7.1 Detailed Description

(Blocked Sequence Set Generator)

File for generating a blocked sequence set

**Author**

Andrew Clayton

**Date**

11/13/2023

**Version**

1.5

Definition in file [BlockGenerator.cpp](#).**4.7.2 Function Documentation****4.7.2.1 main()**

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

Definition at line 30 of file [BlockGenerator.cpp](#).

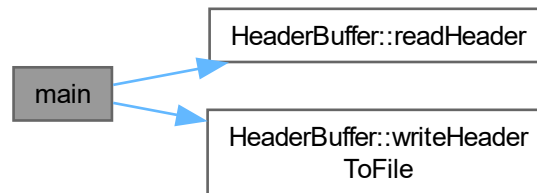
```
00030     {
00031         const int BLOCK_SIZE = 510;
00032         const int BLOCK_CAPACITY = 0.75 * BLOCK_SIZE; // 75% of the block size
00033
00034
00035         // We need to first read and write the header of the file
00036         HeaderBuffer header = HeaderBuffer("us_postal_codes_blocked.txt");
00037         header.readHeader();
00038         header.writeHeaderToFile("us_postal_codes_blocked.txt");
00039
00040         // Now we can proceed with the blocked data generation, but we have to make sure the file opens up
00041         // where we left off
00042
00043
00044         // Check if the correct number of command line arguments were given
00045         if (argc < 2) {
00046             cerr << "Error: No file name given.\n";
00047             return 1;
00048         }
00049
00050         // File to write data out to
00051         string blockedDataFile = argv[1]; // Assumes the first command line argument is the file name
00052         ofstream writeFile;
00053         // we need to append to the file, not overwrite it
00054         writeFile.open(blockedDataFile + ".txt", ios::app);
00055         if (!writeFile.is_open()) {
00056             cerr << "Error: Could not open file " << blockedDataFile << " for writing.\n";
00057             return 1;
00058         }
00059
00060         // File to read information from
00061         ifstream readFile("uspostal_codes.txt");
00062         if (!readFile.is_open()) {
00063             cerr << "Error: Could not open file uspostal_codes.txt for reading.\n";
00064             return 1;
00065         }
00066
00067         // We need to have the file open to the actual content, past the metadata.
00068
00069         // Now we go through the file and convert the length-indicated data to blocked data, ensuring that
00070         // records stay complete within the BLOCK_CAPACITY
00071
00072         // Initialize variables for looping
00073         string currentLine;
00074         getline(readFile, currentLine); // Skipping metadata
00075         int currentBlockSize = 0;
00076         int currentBlock = 0;
```

```

00077     int numRecords = 0;
00078     vector<string> recordsInBlock;
00079     bool isLastBlock = false;
00080
00081     /*
00082     As we process through records, we store all of them in a vector. When the vector cannot add
00083     another record without reaching block capacity, then we
00084     write the (length-indicated) metadata to the file, followed by all of the vectors. Then we
00085     proceeding with checking more records.
00086     - First block will have its previous block number as -1
00087     - Last block (when we reach the end of the file for records) will have its next block number
00088     as -1
00089     */
00090
00091     while (getline(readFile, currentLine)) {
00092         int currentLineLength = stoi(currentLine.substr(0, 2)) + 3; // Including LI and comma
00093         recordsInBlock.push_back(currentLine);
00094         numRecords++;
00095         currentBlockSize += currentLineLength;
00096
00097         if (currentBlockSize > BLOCK_CAPACITY || readFile.eof()) {
00098             isLastBlock = readFile.eof();
00099
00100             // Metadata calculation
00101             string metadata = to_string(currentBlock) + "," + to_string(numRecords) + "," +
00102             (currentBlock == 0 ? "-1" : to_string(currentBlock - 1)) + "," + (isLastBlock ? "-1" :
00103             to_string(currentBlock + 1)) + ",";
00104             int metadataLength = metadata.length() + 3; // Including LI and comma and ending comma
00105
00106             // Write metadata and records
00107             // Metadata format: LI,RBN,#ofRecords,prevBlock,nextBlock,
00108             writeFile << metadataLength << "," << metadata;
00109             for (const string& record : recordsInBlock) {
00110                 writeFile << record;
00111             }
00112
00113             // Pad the block with '~'
00114             for (int i = 0; i < BLOCK_SIZE - currentBlockSize - metadataLength; i++) {
00115                 writeFile << "~";
00116             }
00117             writeFile << "\n"; // New block
00118
00119             // Reset for the next block
00120             recordsInBlock.clear();
00121             currentBlockSize = 0;
00122             numRecords = 0;
00123             currentBlock++;
00124         }
00125     }
00126
00127     // Anything left in the vector now will be the last block
00128     isLastBlock = true;
00129
00130     // We need to print out everything left in the vector
00131     string metadata = to_string(currentBlock) + "," + to_string(numRecords) + "," + (currentBlock == 0
00132     ? "-1" : to_string(currentBlock - 1)) + "," + (isLastBlock ? "-1" : to_string(currentBlock + 1)) +
00133     ",";
00134     int metadataLength = metadata.length() + 3; // Including LI and comma and ending comma
00135
00136     // Write metadata and records
00137     // Metadata format: LI,RBN,#ofRecords,prevBlock,nextBlock,
00138     writeFile << metadataLength << "," << metadata;
00139     for (const string& record : recordsInBlock) {
00140         writeFile << record;
00141     }
00142
00143     // Pad the block with '~'
00144     for (int i = 0; i < BLOCK_SIZE - currentBlockSize - metadataLength; i++) {
00145         writeFile << "~";
00146     }
00147
00148     // Once all of this is done, we need to create an empty avail list. The next and previous RBNs
00149     will be -1
00150     // The avail list will be at the end of the file, and will be the last block
00151     // The metadata will be: LI,RBN,#ofRecords,prevBlock,nextBlock,
00152     string availListMetadata = to_string(currentBlock+1) + ",0,-1,-1,";
00153     int availListMetadataLength = availListMetadata.length() + 3; // Including LI and comma and ending
00154     comma
00155     writeFile << "\n" << availListMetadataLength << "," << availListMetadata;
00156     for (int i = 0; i < BLOCK_SIZE - availListMetadataLength; i++) {
00157         writeFile << "~";
00158     }
00159
00160     readFile.close();
00161     writeFile.close();
00162     return 0;
00163 }

```

Here is the call graph for this function:



## 4.8 BlockGenerator.cpp

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

```

00001 // -----
00009 // -----
//----- 00020
00021
00022 #include <iostream>
00023 #include <fstream>
00024 #include <string>
00025 #include <vector>
00026 #include "HeaderBuffer.h"
00027
00028 using namespace std;
00029
00030 int main(int argc, char* argv[]) {
00031     const int BLOCK_SIZE = 510;
00032     const int BLOCK_CAPACITY = 0.75 * BLOCK_SIZE; // 75% of the block size
00033
00034
00035     // We need to first read and write the header of the file
00036     HeaderBuffer header = HeaderBuffer("us_postal_codes_blocked.txt");
00037     header.readHeader();
00038     header.writeHeaderToFile("us_postal_codes_blocked.txt");
00039
00040     // Now we can proceed with the blocked data generation, but we have to make sure the file opens up
    where we left off
00041
00042
00043
00044     // Check if the correct number of command line arguments were given
00045     if (argc < 2) {
00046         cerr << "Error: No file name given.\n";
00047         return 1;
00048     }
00049
00050     // File to write data out to
00051     string blockedDataFile = argv[1]; // Assumes the first command line argument is the file name
00052     ofstream writeFile;
00053     // we need to append to the file, not overwrite it
00054     writeFile.open(blockedDataFile + ".txt", ios::app);
00055     if (!writeFile.is_open()) {
00056         cerr << "Error: Could not open file " << blockedDataFile << " for writing.\n";
00057         return 1;
00058     }
00059
00060     // File to read information from
00061     ifstream readFile("uspostal_codes.txt");
00062     if (!readFile.is_open()) {
00063         cerr << "Error: Could not open file uspostal_codes.txt for reading.\n";
00064         return 1;
00065     }
00066
00067     // We need to have the file open to the actual content, past the metadata.
00068
00069     // Now we go through the file and convert the length-indicated data to blocked data, ensuring that
00070     // records stay complete within the BLOCK_CAPACITY
  
```



```

00071
00072     // Initialize variables for looping
00073     string currentLine;
00074     getline(readFile, currentLine); // Skipping metadata
00075     int currentBlockSize = 0;
00076     int currentBlock = 0;
00077     int numRecords = 0;
00078     vector<string> recordsInBlock;
00079     bool isLastBlock = false;
00080
00081     /*
00082     As we process through records, we store all of them in a vector. When the vector cannot add
00083     another record without reaching block capacity, then we
00084     write the (length-indicated) metadata to the file, followed by all of the vectors. Then we
00085     proceeding with checking more records.
00086     - First block will have its previous block number as -1
00087     - Last block (when we reach the end of the file for records) will have its next block number
00088     as -1
00089     */
00090
00091     while (getline(readFile, currentLine)) {
00092         int currentLineLength = stoi(currentLine.substr(0, 2)) + 3; // Including LI and comma
00093         recordsInBlock.push_back(currentLine);
00094         numRecords++;
00095         currentBlockSize += currentLineLength;
00096
00097         if (currentBlockSize > BLOCK_CAPACITY || readFile.eof()) {
00098             isLastBlock = readFile.eof();
00099
00100             // Metadata calculation
00101             string metadata = to_string(currentBlock) + "," + to_string(numRecords) + "," +
00102             (currentBlock == 0 ? "-1" : to_string(currentBlock - 1)) + "," + (isLastBlock ? "-1" :
00103             to_string(currentBlock + 1)) + ",";
00104             int metadataLength = metadata.length() + 3; // Including LI and comma and ending comma
00105
00106             // Write metadata and records
00107             // Metadata format: LI,RBN,#ofRecords,prevBlock,nextBlock,
00108             writeFile << metadataLength << "," << metadata;
00109             for (const string& record : recordsInBlock) {
00110                 writeFile << record;
00111             }
00112
00113             // Pad the block with '~'
00114             for (int i = 0; i < BLOCK_SIZE - currentBlockSize - metadataLength; i++) {
00115                 writeFile << "~";
00116             }
00117             writeFile << "\n"; // New block
00118
00119             // Reset for the next block
00120             recordsInBlock.clear();
00121             currentBlockSize = 0;
00122             numRecords = 0;
00123             currentBlock++;
00124         }
00125     }
00126
00127     // Anything left in the vector now will be the last block
00128     isLastBlock = true;
00129
00130     // We need to print out everything left in the vector
00131     string metadata = to_string(currentBlock) + "," + to_string(numRecords) + "," + (currentBlock == 0
00132     ? "-1" : to_string(currentBlock - 1)) + "," + (isLastBlock ? "-1" : to_string(currentBlock + 1)) +
00133     ",";
00134     int metadataLength = metadata.length() + 3; // Including LI and comma and ending comma
00135
00136     // Write metadata and records
00137     // Metadata format: LI,RBN,#ofRecords,prevBlock,nextBlock,
00138     writeFile << metadataLength << "," << metadata;
00139     for (const string& record : recordsInBlock) {
00140         writeFile << record;
00141     }
00142
00143     // Pad the block with '~'
00144     for (int i = 0; i < BLOCK_SIZE - currentBlockSize - metadataLength; i++) {
00145         writeFile << "~";
00146     }
00147
00148     // Once all of this is done, we need to create an empty avail list. The next and previous RBNS
00149     will be -1
00150     // The avail list will be at the end of the file, and will be the last block
00151     // The metadata will be: LI,RBN,#ofRecords,prevBlock,nextBlock,
00152     string availListMetadata = to_string(currentBlock+1) + ",0,-1,-1,";
00153     int availListMetadataLength = availListMetadata.length() + 3; // Including LI and comma and ending
00154     comma
00155     writeFile << "\n" << availListMetadataLength << "," << availListMetadata;
00156     for (int i = 0; i < BLOCK_SIZE - availListMetadataLength; i++) {
00157         writeFile << "~";
00158     }

```

```

00149     }
00150
00151     readFile.close();
00152     writeFile.close();
00153     return 0;
00154 }

```

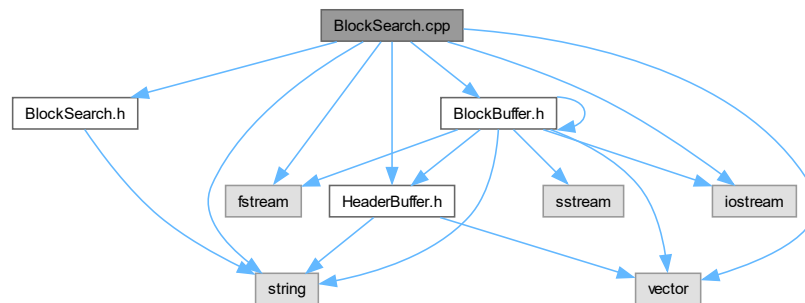
## 4.9 BlockSearch.cpp File Reference

```

#include <string>
#include <fstream>
#include <iostream>
#include "BlockSearch.h"
#include <vector>
#include "BlockBuffer.h"
#include "HeaderBuffer.h"

```

Include dependency graph for BlockSearch.cpp:



### Functions

- int [findZipcode](#) (const string &record)

### 4.9.1 Function Documentation

#### 4.9.1.1 findZipcode()

```

int findZipcode (
    const string & record )

```

Definition at line 23 of file [BlockSearch.cpp](#).

```

00023     {
00024         // The zipcode is the substring between the first and second comma
00025         size_t firstComma = record.find(',');
00026         size_t secondComma = record.find(',', firstComma + 1);
00027         if (firstComma == string::npos || secondComma == string::npos) {
00028             cerr << "Error parsing record for zipcode: " << record << "\n";
00029             return -1;
00030         }
00031         return stoi(record.substr(firstComma + 1, secondComma - firstComma - 1));
00032     }

```

## 4.10 BlockSearch.cpp

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

```

00001
00005
00006 #include <string>
00007 #include <fstream>
00008 #include <iostream>
00009 #include "BlockSearch.h"
00010 #include <vector>
00011 #include "BlockBuffer.h"
00012 #include "HeaderBuffer.h"
00013
00014 using namespace std;
00015
00016
00017 // Default constructor
00018 BlockSearch::BlockSearch(string idxFile) {
00019     indexFile = idxFile;
00020 }
00021
00022
00023 int findZipcode(const string& record) {
00024     // The zipcode is the substring between the first and second comma
00025     size_t firstComma = record.find(',');
00026     size_t secondComma = record.find(',', firstComma + 1);
00027     if (firstComma == string::npos || secondComma == string::npos) {
00028         cerr << "Error parsing record for zipcode: " << record << "\n";
00029         return -1;
00030     }
00031     return stoi(record.substr(firstComma + 1, secondComma - firstComma - 1));
00032 }
00033
00034 // Searches for a record in the blocked index file by key (zipcode)
00035 string BlockSearch::searchForRecord(int target) {
00036     // Open the index file
00037     ifstream readFile(indexFile);
00038     string line;
00039
00040     // Read through the file until we find target <= greatestKeyInBlock
00041
00042     // Iterate through each line of the file
00043     while (getline(readFile, line)) {
00044         int commaIdx = line.find(',');
00045         int rbn = 0;
00046         try {
00047             rbn = stoi(line.substr(0, commaIdx));
00048         } catch (invalid_argument& e) {
00049             cerr << "Error parsing RBN: " << e.what() << endl;
00050             // return "-1";
00051         }
00052
00053         int greatestKeyInBlock;
00054         try {
00055             greatestKeyInBlock = stoi(line.substr(commaIdx+1));
00056         } catch (invalid_argument& e) {
00057             cerr << "Error parsing greatest key in block: " << e.what() << endl;
00058             // return "-1";
00059         }
00060
00061         if (target <= greatestKeyInBlock) {
00062             // We have found the block that contains the record we are looking for
00063             // now we need to actually access the block itself, which we should be able to do with
00064             BlockBuffer
00065
00066             ifstream dataFile("us_postal_codes_blocked.txt", std::ios::app);
00067             HeaderBuffer headerBuffer2("us_postal_codes_blocked.txt");
00068             BlockBuffer blockbuffer(dataFile, headerBuffer2);
00069
00070             // We break down all the block into a vector of records
00071
00072             vector<string> records = blockbuffer.readBlock(rbn);
00073
00074             for (string record : records) { // Check if each record is the target record
00075                 int commaIdx = record.find(',');
00076                 int zipcode = stoi(record.substr(0, commaIdx));
00077
00078                 if (zipcode == target) {
00079                     return record;
00080                 }
00081             }
00082             break; // not found in this block, and next blocks have greater keys
00083         }
00084     }
    // We could not find the block that contains the record we are looking for

```

```

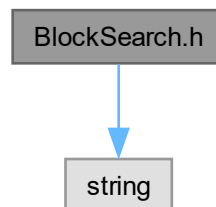
00085     return "-1";
00086 }
00087
00088 void BlockSearch::displayRecord(string record) {
00089     // The format of a record is: zipcode,town,state,county,latitude,longitude
00090     vector<string> fields;
00091     stringstream ss(record);
00092     string field;
00093
00094     while (getline(ss, field, ',')) {
00095         fields.push_back(field);
00096     }
00097
00098     if (fields.size() == 6) { // Ensure there are exactly 6 fields: zipcode, town, state, county,
latitude, longitude
00099         cout << "Zipcode: " << fields[0] << endl;
00100         cout << "Town: " << fields[1] << endl;
00101         cout << "State: " << fields[2] << endl;
00102         cout << "County: " << fields[3] << endl;
00103         cout << "Latitude: " << fields[4] << endl;
00104         cout << "Longitude: " << fields[5] << "\n\n";
00105     } else {
00106         cerr << "Zipcode not found." << endl;
00107     }
00108 }
00109 }

```

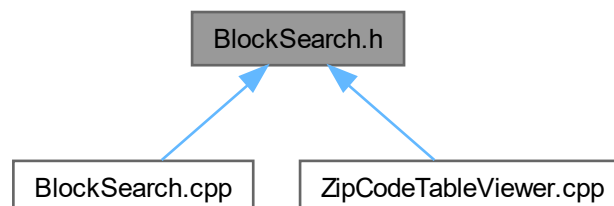
## 4.11 BlockSearch.h File Reference

```
#include <string>
```

Include dependency graph for BlockSearch.h:



This graph shows which files directly or indirectly include this file:



## Classes

- class [BlockSearch](#)

Implementation of the [BlockSearch](#) class for searching for records in the blocked index file.

## 4.12 BlockSearch.h

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

```

00001 // -----
00010 // -----
00018 // -----
00019
00020 #ifndef BLOCKSEARCH_H
00021 #define BLOCKSEARCH_H
00022
00023 #include <string>
00024 using namespace std;
00025
00026 class BlockSearch {
00027 private:
00028
00029     // The index file to open
00030     string indexFile;
00031
00032
00033 public:
00039     BlockSearch() { indexFile = "blocked_Index.txt"; }
00040
00047     BlockSearch(string idxFile);
00048
00049
00057     string searchForRecord(int target);
00058
00065     void displayRecord(string record);
00066
00067 };
00068 #endif

```

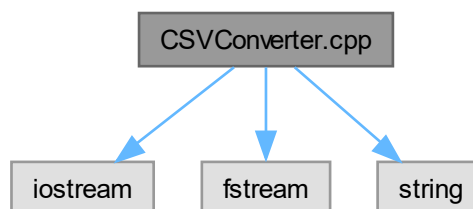
## 4.13 CSVConverter.cpp File Reference

```

#include <iostream>
#include <fstream>
#include <string>

```

Include dependency graph for CSVConverter.cpp:



## Functions

- int [convertCSV](#) ()

## 4.13.1 Function Documentation

### 4.13.1.1 convertCSV()

int convertCSV ( )

Definition at line 7 of file [CSVConverter.cpp](#).

```

00007         {
00008             std::ifstream inputFile("us_postal_codes.csv"); // Open the input file
00009             std::ofstream outputFile("us_postal_codes.txt"); // Open the output file
00010
00011             if (!inputFile.is_open() || !outputFile.is_open()) {
00012                 std::cerr << "Error opening files." << std::endl;
00013                 return 1;
00014             }
00015
00016             std::string record;
00017             int recordNumber = 1;
00018
00019             std::getline(inputFile, record); // Skip the header line
00020
00021             while (std::getline(inputFile, record)) {
00022                 // Calculate the length of the record
00023                 int recordLength = record.length();
00024
00025                 outputFile << recordLength << ',' << record << std::endl;
00026
00027                 ++recordNumber;
00028             }
00029
00030             inputFile.close(); // Close the input file
00031             outputFile.close(); // Close the output file
00032
00033             std::cout << "Prepended length field to " << recordNumber - 1 << " records." << std::endl;
00034 }

```

## 4.14 CSVConverter.cpp

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

```

00001 // Program to convert a CSV file to the length-indicated file structure.
00002
00003 #include <iostream>
00004 #include <fstream>
00005 #include <string>
00006
00007 int convertCSV() {
00008     std::ifstream inputFile("us_postal_codes.csv"); // Open the input file
00009     std::ofstream outputFile("us_postal_codes.txt"); // Open the output file
00010
00011     if (!inputFile.is_open() || !outputFile.is_open()) {
00012         std::cerr << "Error opening files." << std::endl;
00013         return 1;
00014     }
00015
00016     std::string record;
00017     int recordNumber = 1;
00018
00019     std::getline(inputFile, record); // Skip the header line
00020
00021     while (std::getline(inputFile, record)) {
00022         // Calculate the length of the record
00023         int recordLength = record.length();
00024
00025         outputFile << recordLength << ',' << record << std::endl;
00026
00027         ++recordNumber;
00028     }
00029
00030     inputFile.close(); // Close the input file
00031     outputFile.close(); // Close the output file
00032
00033     std::cout << "Prepended length field to " << recordNumber - 1 << " records." << std::endl;
00034 }

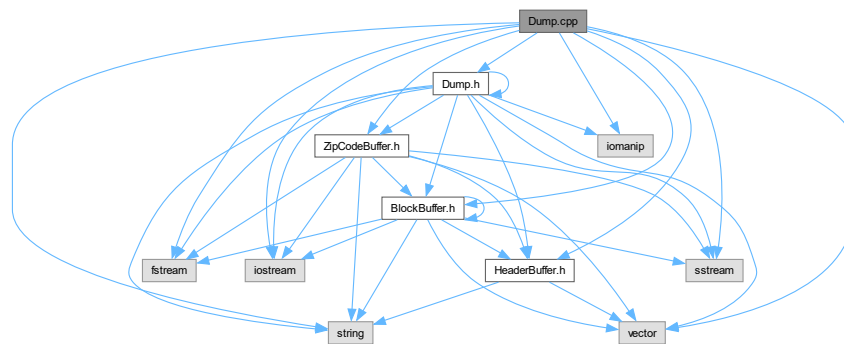
```

## 4.15 Dump.cpp File Reference

Implementation of the [Dump](#) class methods for printing and dumping records.

```
#include "Dump.h"
#include <iostream>
#include <iomanip>
#include "ZipCodeBuffer.h"
#include "BlockBuffer.h"
#include "HeaderBuffer.h"
#include <string>
#include <vector>
#include <sstream>
#include <fstream>
```

Include dependency graph for Dump.cpp:



### 4.15.1 Detailed Description

Implementation of the [Dump](#) class methods for printing and dumping records.

See [Dump.h](#) for the class declaration and documentation.

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

## 4.16 Dump.cpp

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

```
00001
00004
00005 #include "Dump.h"
00006 #include <iostream>
00007 #include <iomanip>
00008 #include "ZipCodeBuffer.h"
00009 #include "BlockBuffer.h"
00010 #include "HeaderBuffer.h"
00011
00012 #include <string>
00013 #include <vector>
00014 #include <sstream>
00015 #include <fstream>
00016 using namespace std;
00017
```

```

00018 Dump::Dump(ZipCodeBuffer &recordBuffer) : recordBuffer(recordBuffer),
    blockBuffer(recordBuffer.blockBuffer), headerBuffer(recordBuffer.headerBuffer) {
00019     // The constructor takes an ifstream and HeaderBuffer, initializing the BlockBuffer
00020     // with the provided file and headerBuffer.
00021     headerBuffer.readHeader();
00022 }
00023
00024
00025
00027 void Dump::dumpLogicalOrder() {
00028     // Display the list head Relative Block Numbers
00029     std::cout << "List Head RBN: " << headerBuffer.getRBNS() << std::endl;
00030     std::cout << "Avail Head RBN: " << headerBuffer.getRBNA() << std::endl;
00031
00032     while (true)
00033     {
00034         std::vector<std::string> records = blockBuffer.readNextBlock();
00035         if (records.size() == 0) // TODO could change to when it reads -1 as next RBN to skip a step
00036         {
00037             // When it receives an empty vector, end the loop
00038             break;
00039         }
00040
00041         printBlock(records);
00042     }
00043 }
00044
00045
00046
00048 void Dump::dumpPhysicalOrder() {
00049     // Display the list head Relative Block Numbers
00050     std::cout << "List Head RBN: " << headerBuffer.getRBNS() << std::endl;
00051     std::cout << "Avail Head RBN: " << headerBuffer.getRBNA() << std::endl;
00052
00053     int i = 0;
00054     int endpoint = headerBuffer.getBlockCount();
00055     while (i < endpoint)
00056     {
00057         // Read the number of blocks listed in the file metadata
00058         std::vector<std::string> records = blockBuffer.readBlock(i++);
00059         printBlock(records);
00060     }
00061 }
00062
00063
00064
00066 void Dump::printBlock(std::vector<std::string> records) {
00067
00068     std::cout << std::left << std::setw(6) << blockBuffer.getPrevRBN(); // Display preceding RBN
00069     for (const std::string& recordString : records) {
00070         ZipCodeRecord record = recordBuffer.parseRecord(recordString);
00071         std::cout << std::setw(6) << record.zipCode; // Display the key (zipCode) for
    each record in the block
00072     }
00073     std::cout << std::left << std::setw(6) << blockBuffer.getNextRBN(); // Display succeeding RBN
00074     std::cout << std::endl;
00075 }
00076
00077
00078
00080 void Dump::dumpBlockIndex(const std::string& filename) {
00081     std::ifstream mainFile(filename);
00082     if (!mainFile.is_open()) {
00083         std::cerr << "Error opening index file." << std::endl;
00084         return;
00085     }
00086
00087     std::string line;
00088     while (std::getline(mainFile, line)) {
00089         std::istringstream ss(line);
00090         std::vector<std::string> tokens;
00091
00092         // Split the line by comma and store tokens in the vector
00093         while (std::getline(ss, line, ',')) {
00094             tokens.push_back(line);
00095         }
00096         std::cout << "RBN: ";
00097         int key = 0;
00098         // Print each value in the tokens vector
00099         for (const auto& token : tokens) {
00100             std::cout << token << " ";
00101             if (key == 0) {
00102                 std::cout << "Primary Key: ";
00103                 key = 1;
00104             } else {
00105                 key = 0;
00106             }

```



```

00107
00108     }
00109
00110     std::cout << "\n";
00111 }
00112 }

```

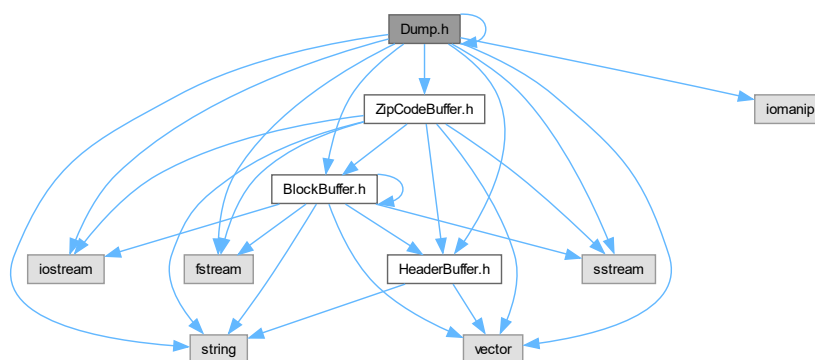
## 4.17 Dump.h File Reference

```

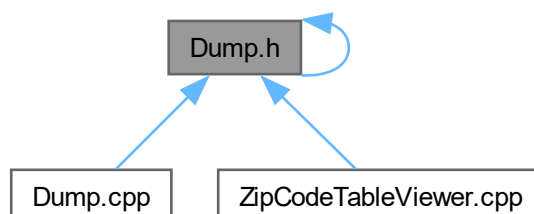
#include "Dump.h"
#include <iostream>
#include <iomanip>
#include "ZipCodeBuffer.h"
#include "BlockBuffer.h"
#include "HeaderBuffer.h"
#include <string>
#include <vector>
#include <sstream>
#include <fstream>

```

Include dependency graph for Dump.h:



This graph shows which files directly or indirectly include this file:



## Classes

- class [Dump](#)

*Class for dumping and printing block records in logical and physical order and for dumping the simple index for the blocks.*

## 4.18 Dump.h

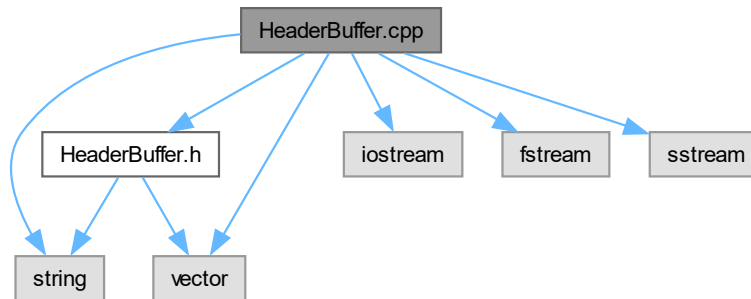
[Go to the documentation of this file.](#)

```
00001 // -----
00012 // -----
00039 // -----
00040
00041 #ifndef PRINTBLOCK_H
00042 #define PRINTBLOCK_H
00043
00044 #include "Dump.h"
00045 #include <iostream>
00046 #include <iomanip>
00047 #include "ZipCodeBuffer.h"
00048 #include "BlockBuffer.h"
00049 #include "HeaderBuffer.h"
00050
00051 #include <string>
00052 #include <vector>
00053 #include <sstream>
00054 #include <fstream>
00055
00056 class Dump {
00057 private:
00058     ZipCodeBuffer &recordBuffer;
00059     BlockBuffer &blockBuffer;
00060     HeaderBuffer &headerBuffer;
00061
00062 public:
00063     Dump(ZipCodeBuffer &recordBuffer);
00064
00070     void dumpLogicalOrder();
00071
00077     void dumpPhysicalOrder();
00078
00085     void printBlock(std::vector<std::string> records);
00086
00092     void dumpBlockIndex(const std::string& filename);
00093 };
00094 #endif // PRINTBLOCK_H
```

## 4.19 HeaderBuffer.cpp File Reference

```
#include "HeaderBuffer.h"
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
```

Include dependency graph for HeaderBuffer.cpp:



## Functions

- int [headerBuffer](#) ( )

### 4.19.1 Function Documentation

#### 4.19.1.1 headerBuffer()

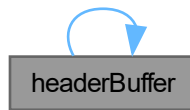
```
int headerBuffer ( )
```

Definition at line 461 of file [HeaderBuffer.cpp](#).

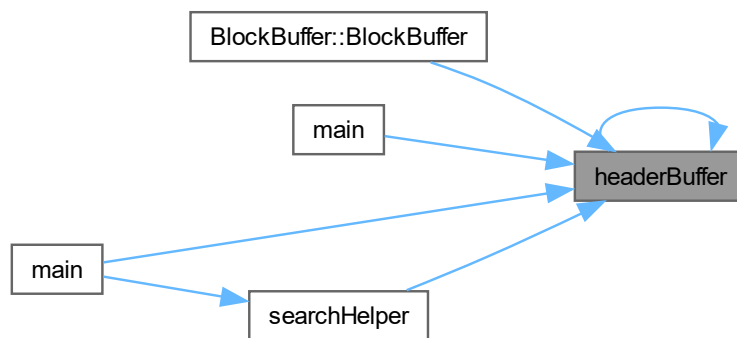
```

00461     {
00462         HeaderBuffer headerBuffer("header.txt");
00463
00464         // Set header fields
00465         headerBuffer.setFileStructureType("1.0");
00466         headerBuffer.setFileStructureVersion("1.0");
00467         headerBuffer.setHeaderSizeBytes(256);
00468         headerBuffer.setRecordSizeBytes(128);
00469         headerBuffer.setSizeFormatType("ASCII");
00470         headerBuffer.setPrimaryKeyIndexFileName("index.txt");
00471         headerBuffer.setRecordCount(1000);
00472         headerBuffer.setFieldCount(2); // Set field count
00473         headerBuffer.setPrimaryKeyFieldIndex(1); // Set primary key index
00474
00475         // Add fields
00476         HeaderBuffer::Field field1;
00477         field1.zipCode = "string";
00478         field1.placeName = "string";
00479         field1.state = "string";
00480         field1.county = "string";
00481         field1.latitude = "double";
00482         field1.longitude = "double";
00483         headerBuffer.addField(field1);
00484
00485         // Write the header to a file
00486         headerBuffer.writeHeader();
00487
00488         // Read the header from a file
00489         headerBuffer.readHeader();
00490
00491         return 0;
00492     }
  
```

Here is the call graph for this function:



Here is the caller graph for this function:



## 4.20 HeaderBuffer.cpp

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

```

00001
00005
00006 #include "HeaderBuffer.h"
00007 #include <iostream>
00008 #include <fstream>
00009 #include <string>
00010 #include <vector>
00011 #include <sstream>
00012
00015 HeaderBuffer::HeaderBuffer(){
00016     // Set default values for member variables
00017     fileStructureType_ = "DefaultType";
00018     fileStructureVersion_ = "0.0";
00019     headerSizeBytes_ = 0;
00020     recordSizeBytes_ = 0;
00021     sizeFormatType_ = "ASCII";
00022     blockSize_ = 0;
00023     minimumBlockCapacity_ = 0;
00024     primaryKeyIndexFileName_ = "default_index.txt";
00025     primaryKeyIndexFileSchema_ = "default_schema";
00026     recordCount_ = 0;
00027     blockCount_ = 0;
00028     fieldCount_ = 0;
00029     primaryKeyFieldIndex_ = 0;
00030     RBNA_ = 0;
00031     RBNS_ = 0;
  
```

```

00032     staleFlag_ = 0;
00033
00034     // Add some default fields
00035     Field defaultField;
00036     defaultField.zipCode = "default_zip";
00037     defaultField.placeName = "default_place";
00038     defaultField.state = "default_state";
00039     defaultField.county = "default_county";
00040     defaultField.latitude = "default_latitude";
00041     defaultField.longitude = "default_longitude";
00042
00043     fields_.push_back(defaultField);
00044 }
00045
00046 HeaderBuffer::HeaderBuffer(const std::string& filename) : filename_(filename) {
00047 }
00048
00049 void HeaderBuffer::writeHeader() {
00050     const std::string tempFilename = "tempfile.txt";
00051
00052     // Step 1: Write the data portion to the temporary file
00053     std::ofstream tempFile(tempFilename);
00054
00055     if (!tempFile.is_open()) {
00056         std::cerr << "Error creating temporary file." << std::endl;
00057         return;
00058     }
00059
00060     // Open the main file
00061     std::ifstream mainFile(filename_);
00062
00063     if (!mainFile.is_open()) {
00064         std::cerr << "Error opening main file." << std::endl;
00065         tempFile.close();
00066         return;
00067     }
00068
00069     // Write your data to the temporary file here
00070     std::string line;
00071     bool copyStarted = false;
00072
00073     while (std::getline(mainFile, line)) {
00074         if (copyStarted) {
00075             tempFile << line << std::endl;
00076         } else if (line.find("Data:") != std::string::npos) {
00077             copyStarted = true;
00078         }
00079     }
00080
00081     // Close the main file and the temporary file
00082     mainFile.close();
00083     tempFile.close();
00084
00085     // Step 2: Overwrite the main file with the header
00086     this->setHeaderSizeBytes(calculateHeaderSize());
00087     writeHeaderToFile(filename_);
00088
00089     // Step 3: Append the data from the temporary file to the main file
00090     std::ifstream tempFileReader(tempFilename);
00091     std::ofstream mainFileWriter(filename_, std::ios::app); // Open the file in append mode
00092
00093     if (!tempFileReader.is_open() || !mainFileWriter.is_open()) {
00094         std::cerr << "Error opening files." << std::endl;
00095         tempFileReader.close();
00096         mainFileWriter.close();
00097         return;
00098     }
00099
00100     mainFileWriter << tempFileReader.rdbuf();
00101
00102     // Close files and remove the temporary file
00103     tempFileReader.close();
00104     mainFileWriter.close();
00105     std::remove(tempFilename.c_str());
00106 }
00107
00108 //version of writeHeader that prints to a file of choice rather than the file held by the object
00109 void HeaderBuffer::writeHeaderToFile(const std::string& filename) {
00110     std::ofstream file(filename);
00111
00112     if (!file.is_open()) {
00113         // Print an error message if the file cannot be opened
00114         std::cerr << "Error opening the file(writeHeaderToFile)." << std::endl;
00115         return;
00116     }
00117 }

```

```

00125
00126 //version for seeing all the stuff
00127 file << "Header:" << std::endl;
00128 file << " - File structure type: " << fileType_ << std::endl;
00129 file << " - File structure version: " << fileStructureVersion_ << std::endl;
00130 file << " - Header Size (bytes): " << headerSizeBytes_ << std::endl;
00131 file << " - Record Size (bytes): " << recordSizeBytes_ << std::endl;
00132 file << " - Size Format Type: " << sizeFormatType_ << std::endl;
00133 file << " - Block Size: " << blockSize_ << std::endl;
00134 file << " - Minimum Block Capacity: " << minimumBlockCapacity_ << std::endl;
00135 file << " - Primary Key Index File: " << primaryKeyIndexFileName_ << std::endl;
00136 file << " - Primary Key Index File Schema: " << primaryKeyIndexFileSchema_ << std::endl;
00137 file << " - Record Count: " << recordCount_ << std::endl;
00138 file << " - Block Count: " << blockCount_ << std::endl;
00139 file << " - Field Count: " << fieldCount_ << std::endl;
00140 file << " - Primary Key: " << primaryKeyFieldIndex_ << std::endl;
00141 file << " - RBN link for Avail List: " << RBNA_ << std::endl;
00142 file << " - RBN link for active sequence set List: " << RBNS_ << std::endl;
00143 file << " - Stale Flag: " << staleFlag_ << std::endl;
00144
00145 for (const Field& field : fields_) {
00146     file << std::endl;
00147     file << "Fields:" << std::endl;
00148     file << " - Zip Code: " << field.zipCode << std::endl;
00149     file << " - Place Name: " << field.placeName << std::endl;
00150     file << " - State: " << field.state << std::endl;
00151     file << " - County: " << field.county << std::endl;
00152     file << " - Latitude: " << field.latitude << std::endl;
00153     file << " - Longitude: " << field.longitude << std::endl;
00154 }
00155
00156 file << std::endl;
00157 file << "Data:" << std::endl;
00158
00159 file.close();
00160 }
00161
00162 void HeaderBuffer::readHeader() {
00163     std::ifstream file(filename_);
00164
00165     if (!file.is_open()) {
00166         // Print an error mesage if the file cannot be opened
00167         std::cerr << "Error opening the file(readHeader)." << std::endl;
00168         return;
00169     }
00170
00171     std::string line;
00172
00173     while (std::getline(file, line)) {
00174         if (line.find(" - File structure type: ") != std::string::npos) {
00175             fileType_ = line.substr(line.find(": ") + 2);
00176         }
00177         else if (line.find(" - File structure version: ") != std::string::npos) {
00178             fileStructureVersion_ = line.substr(line.find(": ") + 2);
00179         }
00180         else if (line.find(" - Header Size (bytes): ") != std::string::npos) {
00181             headerSizeBytes_ = std::stoi(line.substr(line.find(": ") + 2));
00182         }
00183         else if (line.find(" - Record Size (bytes): ") != std::string::npos) {
00184             recordSizeBytes_ = std::stoi(line.substr(line.find(": ") + 2));
00185         }
00186         else if (line.find(" - Size Format Type: ") != std::string::npos) {
00187             sizeFormatType_ = line.substr(line.find(": ") + 2);
00188         }
00189         else if (line.find(" - Block Size: ") != std::string::npos) {
00190             blockSize_ = std::stoi(line.substr(line.find(": ") + 2));
00191         }
00192         else if (line.find(" - Minimum Block Capacity: ") != std::string::npos) {
00193             minimumBlockCapacity_ = std::stoi(line.substr(line.find(": ") + 2));
00194         }
00195         else if (line.find(" - Primary Key Index File: ") != std::string::npos) {
00196             primaryKeyIndexFileName_ = line.substr(line.find(": ") + 2);
00197         }
00198         else if (line.find(" - Primary Key Index File Schema: ") != std::string::npos) {
00199             primaryKeyIndexFileSchema_ = line.substr(line.find(": ") + 2);
00200         }
00201         else if (line.find(" - Record Count: ") != std::string::npos) {
00202             recordCount_ = std::stoi(line.substr(line.find(": ") + 2));
00203         }
00204     }
00205 }
00206
00207
00208
00209
00210
00211
00212
00213

```

```

00214     }
00215     }
00216     else if (line.find(" - Block Count: ") != std::string::npos) {
00217         blockCount_ = std::stoi(line.substr(line.find(": ") + 2));
00218     }
00219     }
00220     else if (line.find(" - Field Count: ") != std::string::npos) {
00221         fieldCount_ = std::stoi(line.substr(line.find(": ") + 2));
00222     }
00223     }
00224     else if (line.find(" - Primary Key: ") != std::string::npos) {
00225         primaryKeyFieldIndex_ = std::stoi(line.substr(line.find(": ") + 2));
00226     }
00227     }
00228     else if (line.find(" - RBN link for Avail List: ") != std::string::npos) {
00229         RBNA_ = std::stoi(line.substr(line.find(": ") + 2));
00230     }
00231     }
00232     else if (line.find(" - RBN link for active sequence set List: ") != std::string::npos) {
00233         RBNS_ = std::stoi(line.substr(line.find(": ") + 2));
00234     }
00235     }
00236     else if (line.find(" - Stale Flag: ") != std::string::npos) {
00237         staleFlag_ = std::stoi(line.substr(line.find(": ") + 2));
00238     }
00239     }
00240     else if (line.find("Fields:") != std::string::npos) {
00241     }
00242     }
00243     Field field;
00244     while (std::getline(file, line)) {
00245         if (line.find(" - Zip Code: ") != std::string::npos) {
00246             field.zipCode = line.substr(line.find(": ") + 2);
00247         }
00248         else if (line.find(" - Place Name: ") != std::string::npos) {
00249             field.placeName = line.substr(line.find(": ") + 2);
00250         }
00251         else if (line.find(" - State: ") != std::string::npos) {
00252             field.state = line.substr(line.find(": ") + 2);
00253         }
00254         else if (line.find(" - County: ") != std::string::npos) {
00255             field.county = line.substr(line.find(": ") + 2);
00256         }
00257         else if (line.find(" - Latitude: ") != std::string::npos) {
00258             field.latitude = line.substr(line.find(": ") + 2);
00259         }
00260         else if (line.find(" - Longitude: ") != std::string::npos) {
00261             field.longitude = line.substr(line.find(": ") + 2);
00262         }
00263     }
00264     fields_.push_back(field);
00265 }
00266 }
00267
00268 file.close();
00269 }
00270
00273 int HeaderBuffer::calculateHeaderSize() const {
00274     std::stringstream headerStream;
00275
00276     // Write header data to a stringstream
00277     headerStream << "Header:\n";
00278     headerStream << " - File structure type: " << fileStructureType_ << "\n";
00279     headerStream << " - File structure version: " << fileStructureVersion_ << "\n";
00280     headerStream << " - Header Size (bytes): " << headerSizeBytes_ << "\n";
00281     headerStream << " - Record Size (bytes): " << recordSizeBytes_ << "\n";
00282     headerStream << " - Size Format Type: " << sizeFormatType_ << "\n";
00283     headerStream << " - Block Size: " << blockSize_ << "\n";
00284     headerStream << " - Minimum Block Capacity: " << minimumBlockCapacity_ << "\n";
00285     headerStream << " - Primary Key Index File: " << primaryKeyIndexFileName_ << "\n";
00286     headerStream << " - Primary Key Index File Schema: " << primaryKeyIndexFileSchema_ << "\n";
00287     headerStream << " - Record Count: " << recordCount_ << "\n";
00288     headerStream << " - Block Count: " << blockCount_ << "\n";
00289     headerStream << " - Field Count: " << fieldCount_ << "\n";
00290     headerStream << " - Primary Key: " << primaryKeyFieldIndex_ << "\n";
00291     headerStream << " - RBN link for Avail List: " << RBNA_ << "\n";
00292     headerStream << " - RBN link for active sequence set List: " << RBNS_ << "\n";
00293     headerStream << " - Stale Flag: " << staleFlag_ << "\n";
00294
00295     headerStream << "\nFields:\n";
00296     for (const Field& field : fields_) {
00297         headerStream << " - Zip Code: " << field.zipCode << "\n";
00298         headerStream << " - Place Name: " << field.placeName << "\n";
00299         headerStream << " - State: " << field.state << "\n";
00300         headerStream << " - County: " << field.county << "\n";
00301         headerStream << " - Latitude: " << field.latitude << "\n";
00302         headerStream << " - Longitude: " << field.longitude << "\n";

```

```

00303     }
00304
00305     headerStream << "\nData:\n";
00306
00307     // Calculate total size including the newline characters
00308     return static_cast<int>(headerStream.str().size());
00309 }
00310
00311 void HeaderBuffer::setFileStructureType(const std::string& fileStructureType) {
00312     fileStructureType_ = fileStructureType;
00313 }
00314
00315 void HeaderBuffer::setFileStructureVersion(const std::string& fileStructureVersion) {
00316     fileStructureVersion_ = fileStructureVersion;
00317 }
00318
00319 void HeaderBuffer::setHeaderSizeBytes(int headerSizeBytes) {
00320     headerSizeBytes_ = headerSizeBytes;
00321 }
00322
00323 void HeaderBuffer::setRecordSizeBytes(int recordSizeBytes) {
00324     recordSizeBytes_ = recordSizeBytes;
00325 }
00326
00327 void HeaderBuffer::setSizeFormatType(const std::string& sizeFormatType) {
00328     sizeFormatType_ = sizeFormatType;
00329 }
00330
00331 void HeaderBuffer::setBlockSize(int blockSize) {
00332     blockSize_ = blockSize;
00333 }
00334
00335 void HeaderBuffer::setminimumBlockCapacity(int minimumBlockCapacity) {
00336     minimumBlockCapacity_ = minimumBlockCapacity;
00337 }
00338
00339 void HeaderBuffer::setPrimaryKeyIndexFileName(const std::string& primaryKeyIndexFileName) {
00340     primaryKeyIndexFileName_ = primaryKeyIndexFileName;
00341 }
00342
00343 void HeaderBuffer::setprimaryKeyIndexFileSchema(const std::string& primaryKeyIndexFileSchema) {
00344     primaryKeyIndexFileSchema_ = primaryKeyIndexFileSchema;
00345 }
00346
00347 void HeaderBuffer::setRecordCount(int recordCount) {
00348     recordCount_ = recordCount;
00349 }
00350
00351 void HeaderBuffer::setBlockCount(int blockCount) {
00352     blockCount_ = blockCount;
00353 }
00354
00355 void HeaderBuffer::setFieldCount(int fieldCount) {
00356     fieldCount_ = fieldCount;
00357 }
00358
00359 void HeaderBuffer::setPrimaryKeyFieldIndex(int primaryKeyFieldIndex) {
00360     primaryKeyFieldIndex_ = primaryKeyFieldIndex;
00361 }
00362
00363 void HeaderBuffer::setRBNA(int RBNA) {
00364     RBNA_ = RBNA;
00365 }
00366
00367 void HeaderBuffer::setRBNS(int RBNS) {
00368     RBNS_ = RBNS;
00369 }
00370
00371 void HeaderBuffer::setstaleFlag(int staleFlag) {
00372     staleFlag_ = staleFlag;
00373 }
00374
00375 void HeaderBuffer::addField(const Field& field) {
00376     fields_.push_back(field);
00377 }
00378
00379 std::string HeaderBuffer::getFileStructureType() const {
00380     return fileStructureType_;
00381 }
00382
00383 std::string HeaderBuffer::getFileStructureVersion() const {
00384     return fileStructureVersion_;
00385 }
00386
00387 int HeaderBuffer::getHeaderSizeBytes() const {
00388     return headerSizeBytes_;
00389 }

```



```

00410
00411     int HeaderBuffer::getRecordSizeBytes() const {
00412         return recordSizeBytes_;
00413     }
00414
00415     std::string HeaderBuffer::getSizeFormatType() const {
00416         return sizeFormatType_;
00417     }
00418
00419     int HeaderBuffer::getBlockSize() const {
00420         return blockSize_;
00421     }
00422
00423     int HeaderBuffer::getMinimumBlockCapacity() const {
00424         return minimumBlockCapacity_;
00425     }
00426
00427     int HeaderBuffer::getBlockCount() const {
00428         return blockCount_;
00429     }
00430     std::string HeaderBuffer::getPrimaryKeyIndexFileName() const {
00431         return primaryKeyIndexFileName_;
00432     }
00433
00434     int HeaderBuffer::getRecordCount() const {
00435         return recordCount_;
00436     }
00437
00438     int HeaderBuffer::getFieldCount() const {
00439         return fieldCount_;
00440     }
00441
00442     int HeaderBuffer::getPrimaryKeyFieldIndex() const {
00443         return primaryKeyFieldIndex_;
00444     }
00445
00446     int HeaderBuffer::getRBNA() const {
00447         return RBNA_;
00448     }
00449
00450     int HeaderBuffer::getRBNS() const {
00451         return RBNS_;
00452     }
00453
00454     int HeaderBuffer::getStaleFlag() const {
00455         return staleFlag_;
00456     }
00457     //const std::vector<Field>& HeaderBuffer::getFields() const {
00458     //    return fields_;
00459     // }
00460
00461     int headerBuffer() {
00462         HeaderBuffer headerBuffer("header.txt");
00463
00464         // Set header fields
00465         headerBuffer.setFileStructureType("1.0");
00466         headerBuffer.setFileStructureVersion("1.0");
00467         headerBuffer.setHeaderSizeBytes(256);
00468         headerBuffer.setRecordSizeBytes(128);
00469         headerBuffer.setSizeFormatType("ASCII");
00470         headerBuffer.setPrimaryKeyIndexFileName("index.txt");
00471         headerBuffer.setRecordCount(1000);
00472         headerBuffer.setFieldCount(2); // Set field count
00473         headerBuffer.setPrimaryKeyFieldIndex(1); // Set primary key index
00474
00475         // Add fields
00476         HeaderBuffer::Field field1;
00477         field1.zipCode = "string";
00478         field1.placeName = "string";
00479         field1.state = "string";
00480         field1.county = "string";
00481         field1.latitude = "double";
00482         field1.longitude = "double";
00483         headerBuffer.addField(field1);
00484
00485         // Write the header to a file
00486         headerBuffer.writeHeader();
00487
00488         // Read the header from a file
00489         headerBuffer.readHeader();
00490
00491         return 0;
00492     }

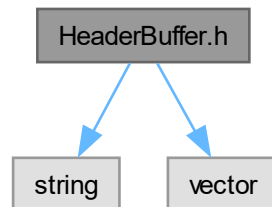
```

## 4.21 HeaderBuffer.h File Reference

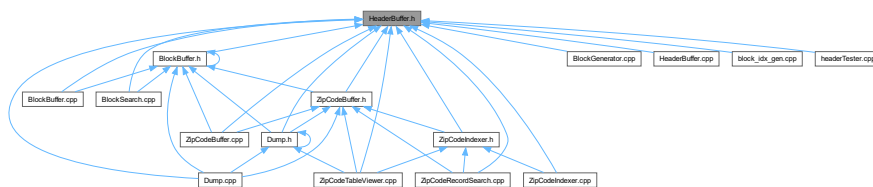
```
#include <string>
```

```
#include <vector>
```

Include dependency graph for HeaderBuffer.h:



This graph shows which files directly or indirectly include this file:



### Classes

- class [HeaderBuffer](#)  
Implementation of the [HeaderBuffer](#) class for for handling header data.
- struct [HeaderBuffer::Field](#)

## 4.22 HeaderBuffer.h

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

```

00001 // -----
00012 // -----
00043 // -----
00044
00045
00046 #ifndef HEADERBUFFER_H
00047 #define HEADERBUFFER_H
00048
00049 #include <string>
00050 #include <vector>
00051
00052 class HeaderBuffer {
00053 public:
00054
00055     struct Field {
00056         std::string zipCode;
00057         std::string placeName;
  
```

```

00058         std::string state;
00059         std::string county;
00060         std::string latitude;
00061         std::string longitude;
00062     };
00063
00064     HeaderBuffer();
00065
00066     HeaderBuffer(const std::string& filename);
00067
00068     void writeHeader();
00069
00070     void writeHeaderToFile(const std::string& filename);
00071
00072     void readHeader();
00073
00074     int calculateHeaderSize() const;
00075
00076     void setFileStructureType(const std::string& fileStructureType);
00077     void setFileStructureVersion(const std::string& fileStructureVersion);
00078     void setHeaderSizeBytes(int headerSizeBytes);
00079     void setRecordSizeBytes(int recordSizeBytes);
00080     void setSizeFormatType(const std::string& sizeFormatType);
00081     void setBlockSize(int blockSize);
00082     void setminimumBlockCapacity(int minimumBlockCapacity);
00083     void setPrimaryKeyIndexFileName(const std::string& primaryKeyIndexFileName);
00084     void setprimaryKeyIndexFileSchema(const std::string& primaryKeyIndexFileSchema);
00085     void setRecordCount(int recordCount);
00086     void setBlockCount(int blockCount);
00087     void setFieldCount(int fieldCount);
00088     void setPrimaryKeyFieldIndex(int primaryKeyFieldIndex);
00089     void setRBNA(int RBNA);
00090     void setRBNS(int RBNS);
00091     void setstaleFlag(int staleFlag);
00092     void addField(const Field& field);
00093
00094     std::string getFileStructureType() const;
00095     std::string getFileStructureVersion() const;
00096     int getHeaderSizeBytes() const;
00097     int getRecordSizeBytes() const;
00098     std::string getSizeFormatType() const;
00099     int getBlockSize() const;
00100     int getMinimumBlockCapacity() const;
00101     std::string getPrimaryKeyIndexFileName() const;
00102     int getRecordCount() const;
00103     int getBlockCount() const;
00104     int getFieldCount() const;
00105     int getPrimaryKeyFieldIndex() const;
00106     int getRBNA() const;
00107     int getRBNS() const;
00108     int getStaleFlag() const;
00109     const std::vector<Field>& getFields() const;
00110
00111 private:
00112     std::string filename_;
00113     std::string fileStructureType_;
00114     std::string fileStructureVersion_;
00115     int headerSizeBytes_;
00116     int recordSizeBytes_;
00117     std::string sizeFormatType_;
00118     int blockSize_;
00119     int minimumBlockCapacity_;
00120     std::string primaryKeyIndexFileName_;
00121     std::string primaryKeyIndexFileSchema_;
00122     int recordCount_;
00123     int blockCount_;
00124     int fieldCount_;
00125     int primaryKeyFieldIndex_;
00126     int RBNA_;
00127     int RBNS_;
00128     int staleFlag_;
00129     std::vector<Field> fields_;
00130 };
00131 // #include "HeaderBuffer.cpp"
00132 #endif // HEADERBUFFER_H

```

## 4.23 headerTester.cpp File Reference

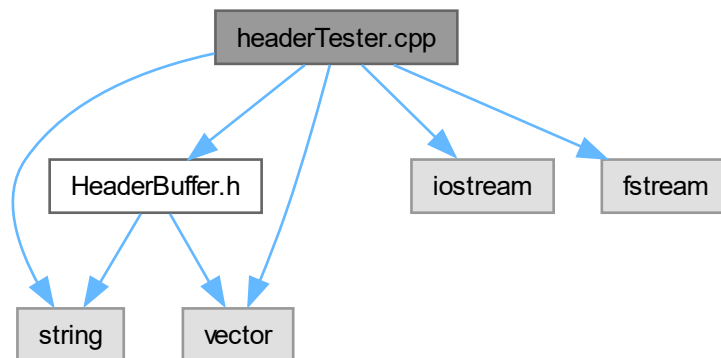
```

#include "HeaderBuffer.h"
#include <string>

```

```
#include <vector>
#include <iostream>
#include <fstream>
```

Include dependency graph for headerTester.cpp:



## Functions

- int [main](#) ()

### 4.23.1 Function Documentation

#### 4.23.1.1 main()

```
int main ( )
```

Definition at line 10 of file [headerTester.cpp](#).

```

00010     {
00011         cout << "testing of the write function\n" << endl;
00012         HeaderBuffer headerBuffer("testwrite.txt");
00013         // Set header fields
00014         headerBuffer.setFileStructureType("1.0");
00015         headerBuffer.setFileStructureVersion("1.0");
00016         headerBuffer.setHeaderSizeBytes(256);
00017         headerBuffer.setRecordSizeBytes(128);
00018         headerBuffer.setSizeFormatType("ASCII");
00019         headerBuffer.setBlockSize(54);
00020         headerBuffer.setMinimumBlockCapacity(6422);
00021         headerBuffer.setPrimaryKeyIndexFileName("index.txt");
00022         headerBuffer.setPrimaryKeyIndexFileSchema("sample");
00023         headerBuffer.setRecordCount(1000);
00024         headerBuffer.setBlockCount(1001);
00025         headerBuffer.setFieldCount(5); // Set field count
00026         headerBuffer.setPrimaryKeyFieldIndex(1); // Set primary key index
00027         headerBuffer.setRBNA(1);
00028         headerBuffer.setRBNS(1);
00029         headerBuffer.setstaleFlag(0);
00030
00031         // Add fields
00032         HeaderBuffer::Field field1;
00033         field1.zipCode = "string";
00034         field1.placeName = "string";
00035         field1.state = "string";
00036         field1.county = "string";
00037         field1.latitude = "double";
00038         field1.longitude = "double";

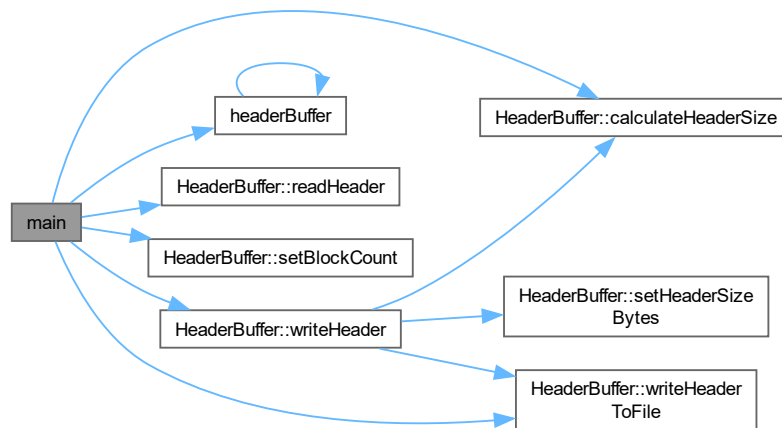
```

```

00039     headerBuffer.addField(field1);
00040
00041     // Write the header to a file
00042     //headerBuffer.writeHeader();
00043
00044     // Read the header from a file
00045     HeaderBuffer headerBuffer2("us_postal_codes.txt");
00046     headerBuffer2.readHeader();
00047     headerBuffer2.setBlockCount(29);
00048     headerBuffer2.writeHeaderToFile("testread.txt");
00049     headerBuffer2.writeHeader();
00050
00051     int calculatedSize = headerBuffer.calculateHeaderSize();
00052     std::cout << "Calculated Header Size: " << calculatedSize << " bytes\n" << std::endl;
00053
00054     int calculatedSize2 = headerBuffer2.calculateHeaderSize();
00055     std::cout << "Calculated Header Size: " << calculatedSize2 << " bytes\n" << std::endl;
00056     return 0;
00057
00058 }

```

Here is the call graph for this function:



## 4.24 headerTester.cpp

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

```

00001 #include "HeaderBuffer.h"
00002 #include <string>
00003 #include <vector>
00004 #include <iostream>
00005 #include <fstream>
00006 using namespace std;
00007
00008
00009
00010 int main(){
00011     cout << "testing of the write function\n" << endl;
00012     HeaderBuffer headerBuffer("testwrite.txt");
00013     // Set header fields
00014     headerBuffer.setFileStructureType("1.0");
00015     headerBuffer.setFileStructureVersion("1.0");
00016     headerBuffer.setHeaderSizeBytes(256);
00017     headerBuffer.setRecordSizeBytes(128);
00018     headerBuffer.setSizeFormatType("ASCII");
00019     headerBuffer.setBlockSize(54);
00020     headerBuffer.setMinimumBlockCapacity(6422);
00021     headerBuffer.setPrimaryKeyIndexFileName("index.txt");
00022     headerBuffer.setprimaryKeyIndexFileSchema("sample");
00023     headerBuffer.setRecordCount(1000);
00024     headerBuffer.setBlockCount(1001);

```

```

00025     headerBuffer.setFieldCount(5); // Set field count
00026     headerBuffer.setPrimaryKeyFieldIndex(1); // Set primary key index
00027     headerBuffer.setRBNA(1);
00028     headerBuffer.setRBNS(1);
00029     headerBuffer.setstaleFlag(0);
00030
00031     // Add fields
00032     HeaderBuffer::Field field1;
00033     field1.zipCode = "string";
00034     field1.placeName = "string";
00035     field1.state = "string";
00036     field1.county = "string";
00037     field1.latitude = "double";
00038     field1.longitude = "double";
00039     headerBuffer.addField(field1);
00040
00041     // Write the header to a file
00042     //headerBuffer.writeHeader();
00043
00044     // Read the header from a file
00045     HeaderBuffer headerBuffer2("us_postal_codes.txt");
00046     headerBuffer2.readHeader();
00047     headerBuffer2.setBlockCount(29);
00048     headerBuffer2.writeHeaderToFile("testread.txt");
00049     headerBuffer2.writeHeader();
00050
00051     int calculatedSize = headerBuffer.calculateHeaderSize();
00052     std::cout << "Calculated Header Size: " << calculatedSize << " bytes\n" << std::endl;
00053
00054     int calculatedSize2 = headerBuffer2.calculateHeaderSize();
00055     std::cout << "Calculated Header Size: " << calculatedSize2 << " bytes\n" << std::endl;
00056     return 0;
00057
00058 }

```

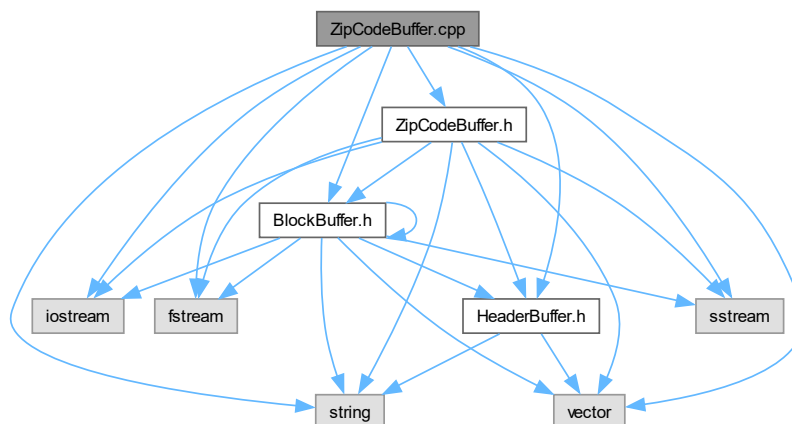
## 4.25 ZipCodeBuffer.cpp File Reference

```

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include "ZipCodeBuffer.h"
#include "BlockBuffer.h"
#include "HeaderBuffer.h"

```

Include dependency graph for ZipCodeBuffer.cpp:



## 4.26 ZipCodeBuffer.cpp

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

```

00001
00004
00005 #include <iostream>
00006 #include <fstream>
00007 #include <string>
00008 #include <vector>
00009 #include <sstream>
00010 #include "ZipCodeBuffer.h"
00011 #include "BlockBuffer.h"
00012 #include "HeaderBuffer.h"
00013
00015 ZipCodeBuffer::ZipCodeBuffer(std::ifstream &file, char fileType = 'L', HeaderBuffer headerBuffer =
HeaderBuffer("us_postal_codes.txt")) : file(file),
00016     fileType(std::toupper(fileType)), blockBuffer(BlockBuffer(file, headerBuffer)),
headerBuffer(headerBuffer) { // TODO change HeaderBuffer initialization once it has a generic
constructor
00017
00018     if (this->fileType == 'C') {
00019         // If CSV, skip the header line.
00020         std::string line;
00021         getline(file, line); // Skip header line
00022     }
00023     else if (this->fileType == 'L' || this->fileType == 'B') {
00024         // If length-indicated or blocked length-indicated file, skip the header line.
00025         // We have to skip past the metadata, up to the "Data: line"
00026         std::string line;
00027         std::getline(file, line);
00028         if (line.find("Header:") != std::string::npos)
00029         {
00030             // File contains a metadata header
00031
00032             // Read lines until "Data:" is found
00033             while (std::getline(file, line)) {
00034                 if (line.find("Data:") != std::string::npos) {
00035                     break;
00036                 }
00037             }
00038         }
00039     }
00040 }
00041 };
00042
00043
00045 ZipCodeRecord ZipCodeBuffer::parseRecord(std::string recordString) {
00046     ZipCodeRecord record;
00047
00048     std::istringstream recordStream(recordString);
00049     std::string field;
00050
00051     // Parse the record fields using istringstream
00052     std::vector<std::string> fields;
00053     while (getline(recordStream, field, ',')) {
00054         fields.push_back(field);
00055     }
00056
00057     if (fields.size() == 6)
00058     {
00059         // Fill the ZipCodeRecord struct with the data from the record
00060         record.zipCode = fields[0];
00061         record.placeName = fields[1];
00062         record.state = fields[2];
00063         record.county = fields[3];
00064         record.latitude = std::stod(fields[4]); // Convert string to double
00065         record.longitude = std::stod(fields[5]);
00066     }
00067     else
00068     {
00069         // The record is malformed and does not have 6 fields. Return terminal string
00070         record.zipCode = "";
00071         std::cerr << "A record contains an invalid number of fields: "
00072             << recordString << std::endl;
00073     }
00074
00075     return record;
00076 };
00077
00078
00080 ZipCodeRecord ZipCodeBuffer::readNextRecord() {
00081     ZipCodeRecord record;
00082     std::string recordString;
00083
00084     if (file.eof())

```

```

00085     {
00086         // End of file reached. Return terminal character
00087         record.zipCode = "";
00088         return record;
00089     }
00090
00091     if (fileType == 'B')
00092     {
00093         if (blockRecordsIndex >= blockRecords.size() || blockRecordsIndex == -1)
00094         {
00095             // Reached the end of the block, so retrieve the next one
00096             blockRecords = blockBuffer.readNextBlock();
00097             if (blockRecords.size() > 0)
00098             {
00099                 recordString = blockRecords[0]; // Retrieve the first record in the block
00100                 blockRecordsIndex = 1;          // Skip 0 because it reads it immediately
00101             }
00102             else
00103             {
00104                 // Did not read a valid block (likely due to the end of file), so return terminal
character
00105                 record.zipCode = "";
00106                 return record;
00107             }
00108         }
00109         else
00110         {
00111             recordString = blockRecords[blockRecordsIndex++];
00112         }
00113     }
00114     else if (fileType == 'C')
00115     {
00116         // If CSV, retrieve the next line in the file as the record to parse
00117         getline(file, recordString);
00118     }
00119     else if (fileType == 'L')
00120     {
00121         // If length-indicated, reads the length and retrieves that many characters for the record
string
00122         int numCharactersToRead = 0;
00123         file >> numCharactersToRead; // Read the length indicator, the first field in each record
00124         file.ignore(1);              // Skip the comma after the length field
00125         recordString.resize(numCharactersToRead);
00126         file.read(&recordString[0], numCharactersToRead);
00127     }
00128
00129     // If not the end of the file, read the fields in the line into the record object
00130     if (recordString.empty())
00131     {
00132         // Did not read a valid record (likely due to the end of file newline), so return terminal
character
00133         record.zipCode = "";
00134         return record;
00135     }
00136
00137     record = parseRecord(recordString);
00138     return record;
00139 };
00140
00141
00143 std::streampos ZipCodeBuffer::getCurrentPosition() {
00144     return file.tellg();
00145 }
00146
00148 std::ifstream& ZipCodeBuffer::setCurrentPosition(std::streampos pos) {
00149     file.seekg(pos);
00150     return file;
00151 }

```

## 4.27 ZipCodeBuffer.h File Reference

```

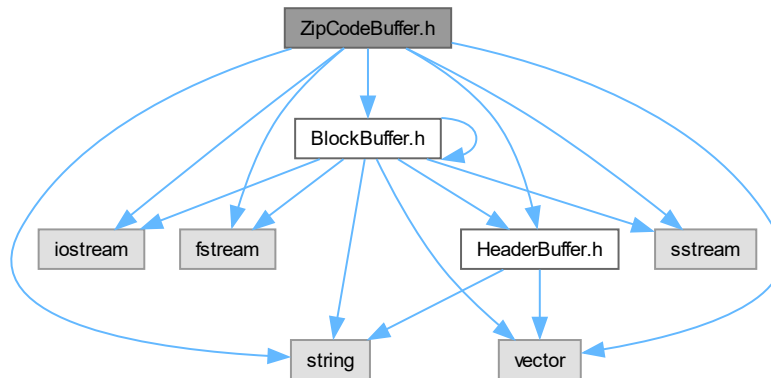
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include "BlockBuffer.h"

```

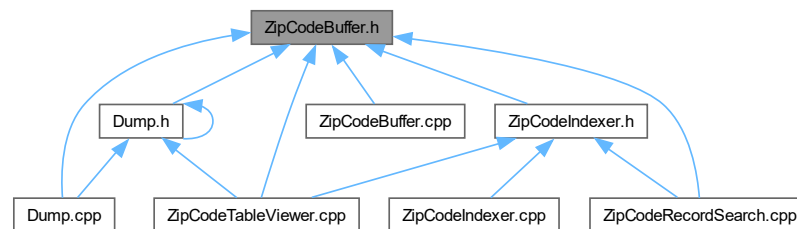


```
#include "HeaderBuffer.h"
```

Include dependency graph for ZipCodeBuffer.h:



This graph shows which files directly or indirectly include this file:



## Classes

- struct [ZipCodeRecord](#)

*Structure to hold a ZIP Code record.*

- class [ZipCodeBuffer](#)

*The [ZipCodeBuffer](#) class parses the file one record at a time and returns the fields in a [ZipCodeRecord](#) struct.*

## 4.28 ZipCodeBuffer.h

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

```

00001 // -----
00011 // -----
00034 // -----
00035
00036
00037 #ifndef ZIPCODEBUFFER_H
00038 #define ZIPCODEBUFFER_H
00039
00040 #include <iostream>

```

```

00041 #include <fstream>
00042 #include <string>
00043 #include <vector>
00044 #include <sstream>
00045 #include "BlockBuffer.h"
00046 #include "HeaderBuffer.h"
00047
00049 struct ZipCodeRecord {
00050     std::string zipCode;
00051     std::string placeName;
00052     std::string state;
00053     std::string county;
00054     double latitude = 0.0;
00055     double longitude = 0.0;
00056 };
00057
00060 class ZipCodeBuffer {
00061 private:
00062     std::ifstream &file;
00063     char fileType;
00064     vector<string> blockRecords; // Stores the current block of records if using a block file format
00065     int blockRecordsIndex = -1; // Default to index 0 so it retrieves the first block on first check
00066
00067 public:
00068     BlockBuffer blockBuffer; // Stores the block metadata if using a block file format
00069     HeaderBuffer headerBuffer = HeaderBuffer("us_postal_codes.txt"); // Stores the header buffer
00070     // TODO once default constructor is added, make it use generic constructor instead
00071
00084     ZipCodeBuffer(std::ifstream &file, char fileType, HeaderBuffer headerBuffer);
00085
00086
00087     /** @brief Destructor to close the file when done. */
00088     ~ZipCodeBuffer();
00089
00109     ZipCodeRecord parseRecord(std::string);
00110
00124     ZipCodeRecord readNextRecord();
00125
00127     std::streampos getCurrentPosition();
00129     std::ifstream& setCurrentPosition(std::streampos);
00130
00131     // Give Dump access to private member functions and variables.
00132     friend class Dump;
00133 };
00134
00135 #endif // ZIPCODEBUFFER_H

```

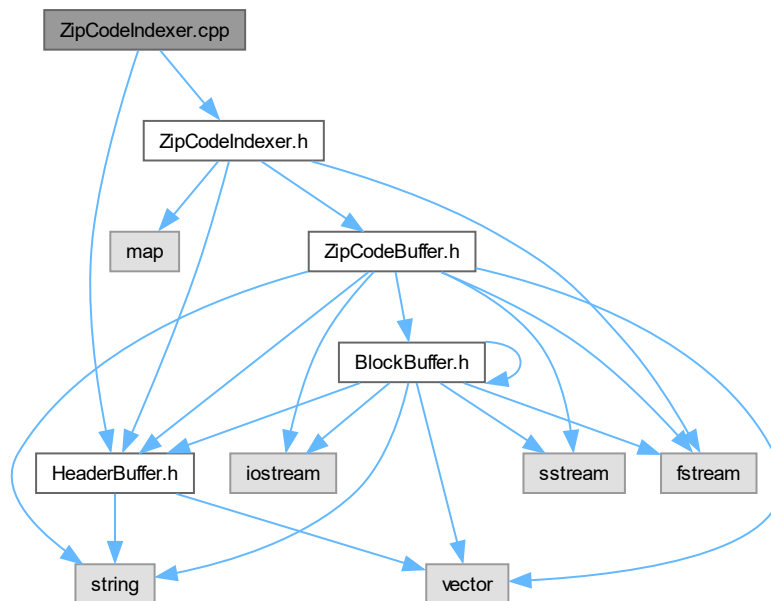
## 4.29 ZipCodeIndexer.cpp File Reference

```

#include "ZipCodeIndexer.h"
#include "HeaderBuffer.h"

```

Include dependency graph for ZipCodeIndexer.cpp:



## 4.30 ZipCodeIndexer.cpp

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

```

00001
00004
00005 #include "ZipCodeIndexer.h"
00006 #include "HeaderBuffer.h"
00007
00009 // Initializes the buffer object with the given file name
00010 // and sets the index file name
00013 ZipCodeIndexer::ZipCodeIndexer(std::ifstream &file, char fileType, const std::string& idxFileName,
    HeaderBuffer headerBuffer)
00014     : buffer(file, fileType, headerBuffer), indexFileName(idxFileName) {}
00015
00017 // This function creates an index of ZIP codes to their positions in the file
00018 // by reading each record in the file using the buffer.
00019 void ZipCodeIndexer::createIndex() {
00020     ZipCodeRecord record;
00021     std::streampos position = buffer.getCurrentPosition();
00022     while (!record = buffer.readNextRecord()).zipCode.empty() {
00023         index[record.zipCode] = position; // Save the position of this ZIP code in the index
00024         position = buffer.getCurrentPosition(); // Get the position of the next record
00025     }
00026 }
00027
00030 void ZipCodeIndexer::writeIndexToFile() {
00031     std::ofstream outFile(indexFileName);
00032     for (const auto& pair : index) {
00033         outFile << pair.first << " " << pair.second << "\n"; // ZIP code and position
00034     }
00035     outFile.close();
00036 }
00037
00039 // This function loads the index from a file into RAM. The index is stored in a std::map
00040 // where the key is the ZIP code and the value is its position in the file.
00041 void ZipCodeIndexer::loadIndexFromRAM() {
00042     index.clear(); // Clear any existing index
00043     std::ifstream inFile(indexFileName);
00044     std::string zip;
00045     std::streampos pos;
00046     int posInt;

```

```

00047     while (inFile » zip » posInt) {
00048         pos = posInt;
00049         index[zip] = pos; // Load the ZIP code and its position into the index
00050     }
00051     inFile.close();
00052 }
00053
00055 // If the ZIP code is not in the index, it returns an invalid position (-1).
00058 std::streampos ZipCodeIndexer::getRecordPosition(const std::string& zipCode) {
00059     if (index.find(zipCode) != index.end()) { // If the ZIP code is in the index
00060         return index[zipCode]; // Return its position
00061     }
00062     else {
00063         return std::streampos(-1); // Invalid position to indicate not found
00064     }
00065 }

```

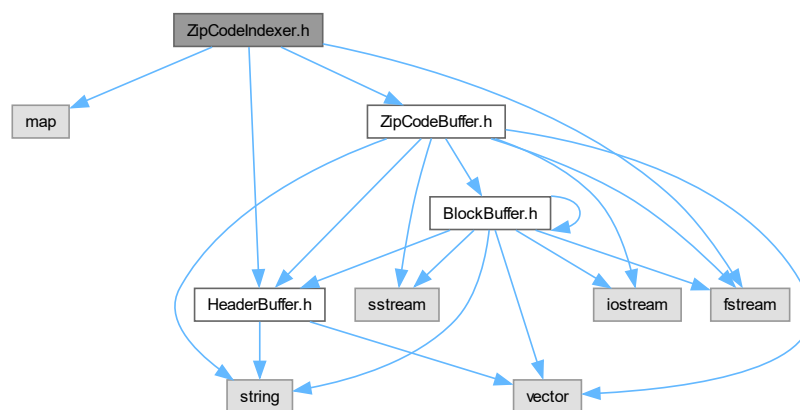
## 4.31 ZipCodeIndexer.h File Reference

```

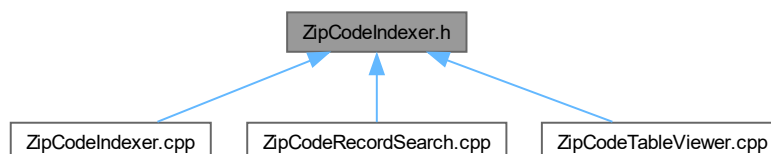
#include <map>
#include <fstream>
#include "ZipCodeBuffer.h"
#include "HeaderBuffer.h"

```

Include dependency graph for ZipCodeIndexer.h:



This graph shows which files directly or indirectly include this file:



## Classes

- class [ZipCodeIndexer](#)

Implementation of the [ZipCodeIndexer](#) class for indexing ZIP code records in a file.

## 4.32 ZipCodeIndexer.h

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

```

00001 // -----
00012 // -----
00027 // -----
00028
00029
00030 // Include guards to prevent double inclusion of this header file.
00031 #ifndef ZIPCODEINDEXER_H
00032 #define ZIPCODEINDEXER_H
00033
00034 // Include necessary header files
00035 #include <map>
00036 #include <fstream>
00037 #include "ZipCodeBuffer.h"
00038 #include "HeaderBuffer.h"
00039
00049 class ZipCodeIndexer {
00050 private:
00051     // Map to store each ZIP code and its position in the file.
00052     std::map<std::string, std::streampos> index;
00053
00054     // File name of the index to be saved/loaded.
00055     std::string idxFileName;
00056
00057     // Instance of ZipCodeBuffer to read ZIP code records from the file.
00058     ZipCodeBuffer buffer;
00059
00060 public:
00069     ZipCodeIndexer(std::ifstream &file, char fileType, const std::string& idxFileName, HeaderBuffer
headerBuffer);
00070
00077     void createIndex();
00078
00084     void writeIndexToFile();
00085
00091     void loadIndexFromRAM();
00092
00099     std::streampos getRecordPosition(const std::string& zipCode);
00100 };
00101
00102 // End of the include guard.
00103 #endif // ZIPCODEINDEXER_H

```

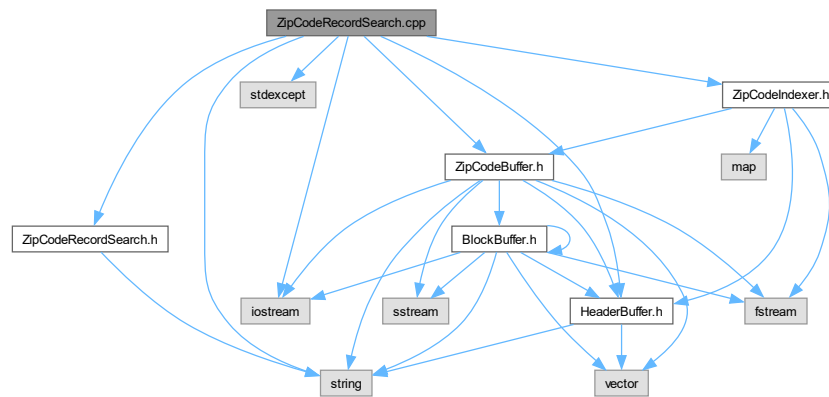
## 4.33 ZipCodeRecordSearch.cpp File Reference

```

#include <iostream>
#include <string>
#include <stdexcept>
#include "ZipCodeBuffer.h"
#include "ZipCodeRecordSearch.h"
#include "ZipCodeIndexer.h"
#include "HeaderBuffer.h"

```

Include dependency graph for ZipCodeRecordSearch.cpp:



## Functions

- bool [isNumber](#) (const char \*s)  
See [ZipCodeRecordSearch.h](#) for details.
- void [displayHelp](#) (const std::string &commandName)  
Prints usage information to the console.
- void [defaultMessage](#) (const std::string &commandName)  
Prints basic message to the console.
- void [searchHelper](#) (std::string fileName, char fileType, char \*zip)  
Helper function to search the buffer for a given ZIP code.

## 4.33.1 Function Documentation

### 4.33.1.1 isNumber()

```
bool isNumber (
    const char * s )
```

See [ZipCodeRecordSearch.h](#) for details.

See [ZipCodeRecordSearch.cpp](#) for details.

Checks to see if a given string is a number.

#### Parameters

<i>s</i>	
----------	--

#### Returns

true  
false

Definition at line 18 of file [ZipCodeRecordSearch.cpp](#).

```
00018         {
00019     try{
00020         int num = std::stoi(std::string(s));
00021         return true;
00022     } catch (std::invalid_argument& e) {
00023         return false;
00024     }
00025 }
```

Here is the caller graph for this function:



#### 4.33.1.2 displayHelp()

```
void displayHelp (
    const std::string & commandName )
```

Prints usage information to the console.

##### Parameters

<i>commandName</i>	
--------------------	--

Definition at line 32 of file [ZipCodeRecordSearch.cpp](#).

```
00032     {
00033     std::cout << std::endl
00034     << "Usage: " << commandName << " [options]" << std::endl
00035     << "-h, --help          Show help options" << std::endl
00036     << "-Z <zipcode>," << std::endl
00037     << "--zipcode <zipcode>  Search record file for <zipcode>" << std::endl << std::endl;
00038 }
```

Here is the caller graph for this function:



#### 4.33.1.3 defaultMessage()

```
void defaultMessage (
    const std::string & commandName )
```

Prints basic message to the console.

#### Parameters

<i>commandName</i>	
--------------------	--

Definition at line 45 of file [ZipCodeRecordSearch.cpp](#).

```
00045                                     {
00046     std::cout << std::endl
00047         << "Try \" " << commandName << " -h\" for more information." << std::endl << std::endl;
00048 }
```

Here is the caller graph for this function:



#### 4.33.1.4 searchHelper()

```
void searchHelper (
    std::string fileName,
    char fileType,
    char * zip )
```

Helper function to search the buffer for a given ZIP code.

#### Parameters

<i>fileName</i>	Name of the file to open.
<i>fileType</i>	Type of the file to open to pass to the indexer for the buffer.
<i>zip</i>	ZIP codes to search for, if any.

Definition at line 57 of file [ZipCodeRecordSearch.cpp](#).

```
00057                                     {
00058     // Create an index and load it from the index file
00059     std::ifstream file(fileName);
00060     HeaderBuffer headerBuffer(fileName);
00061     ZipCodeIndexer index(file, fileType, fileName + "_index.txt", headerBuffer);
00062     index.loadIndexFromRAM();
00063
00064     // Get the position of the ZIP code in the file
00065     std::streampos position = index.getRecordPosition(zip);
00066
00067     if (position != std::streampos(-1)) {
00068         // Open the buffer and set the position
00069         ZipCodeBuffer buffer(file, fileType, headerBuffer);
00070         buffer.setCurrentPosition(position);
00071
00072         // Read the record at the specified position
00073         ZipCodeRecord record = buffer.readNextRecord();
00074
00075         // Print the record info
00076         std::cout << "Zip Code: " << record.zipCode << std::endl
```

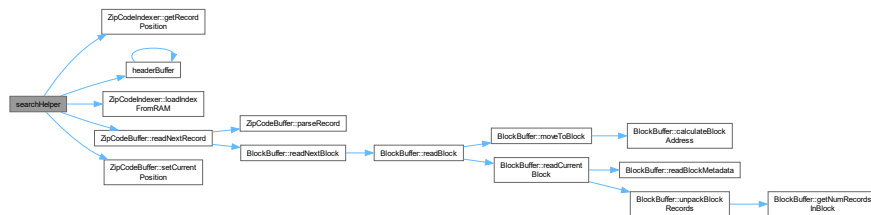


```

00077         « "Place Name: " « record.placeName « std::endl
00078         « "State: " « record.state « std::endl
00079         « "County: " « record.county « std::endl
00080         « "Latitude: " « record.latitude « " Longitude: " « record.longitude
00081         « std::endl « std::endl;
00082     }
00083     else {
00084         // If record not found, print a message
00085         std::cout « "No record of " « zip « std::endl « std::endl;
00086     }
00087 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



## 4.34 ZipCodeRecordSearch.cpp

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

```

00001
00002
00003 #include <iostream>
00004 #include <string>
00005 #include <stdexcept>
00006 #include "ZipCodeBuffer.h"
00007 #include "ZipCodeRecordSearch.h"
00008 #include "ZipCodeIndexer.h"
00009 #include "HeaderBuffer.h"
00010
00018 bool isNumber(const char* s) {
00019     try{
00020         int num = std::stoi(std::string(s));
00021         return true;
00022     } catch (std::invalid_argument& e) {
00023         return false;
00024     }
00025 }
00026
00032 void displayHelp(const std::string& commandName) {
00033     std::cout « std::endl
00034     « "Usage: " « commandName « " [options]" « std::endl
00035     « "-h, --help          Show help options" « std::endl
00036     « "-Z <zipcode>," « std::endl
00037     « "--zipcode <zipcode>  Search record file for <zipcode>" « std::endl « std::endl;
00038 }
00039

```

```

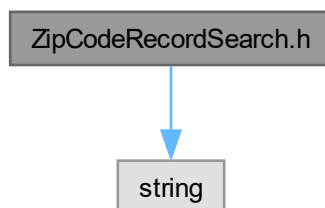
00045 void defaultMessage(const std::string& commandName) {
00046     std::cout << std::endl
00047         << "Try \" " << commandName << " -h\" for more information." << std::endl << std::endl;
00048 }
00049
00057 void searchHelper(std::string fileName, char fileType, char* zip) {
00058     // Create an index and load it from the index file
00059     std::ifstream file(fileName);
00060     HeaderBuffer headerBuffer(fileName);
00061     ZipCodeIndexer index(file, fileType, fileName + "_index.txt", headerBuffer);
00062     index.loadIndexFromRAM();
00063
00064     // Get the position of the ZIP code in the file
00065     std::streampos position = index.getRecordPosition(zip);
00066
00067     if (position != std::streampos(-1)) {
00068         // Open the buffer and set the position
00069         ZipCodeBuffer buffer(file, fileType, headerBuffer);
00070         buffer.setCurrentPosition(position);
00071
00072         // Read the record at the specified position
00073         ZipCodeRecord record = buffer.readNextRecord();
00074
00075         // Print the record info
00076         std::cout << "Zip Code: " << record.zipCode << std::endl
00077             << "Place Name: " << record.placeName << std::endl
00078             << "State: " << record.state << std::endl
00079             << "County: " << record.county << std::endl
00080             << "Latitude: " << record.latitude << " Longitude: " << record.longitude
00081             << std::endl << std::endl;
00082     }
00083     else {
00084         // If record not found, print a message
00085         std::cout << "No record of " << zip << std::endl << std::endl;
00086     }
00087 }

```

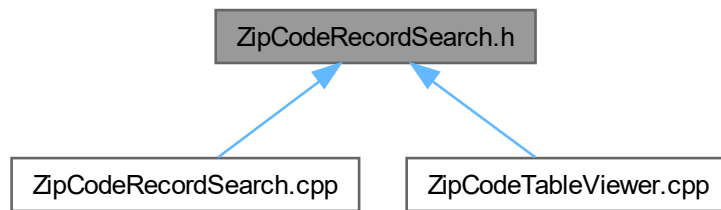
## 4.35 ZipCodeRecordSearch.h File Reference

#include <string>

Include dependency graph for ZipCodeRecordSearch.h:



This graph shows which files directly or indirectly include this file:



## Macros

- `#define` [ZIPCODERECORDSEARCH\\_H](#)

## Functions

- `bool` [isNumber](#) (const char \*s)  
*See [ZipCodeRecordSearch.cpp](#) for details.*
- `void` [displayHelp](#) (const std::string &commandName)  
*Prints usage information to the console.*
- `void` [defaultMessage](#) (const std::string &commandName)  
*Prints basic message to the console.*
- `void` [searchHelper](#) (std::string fileName, char fileType, char \*zip)  
*Helper function to search the buffer for a given ZIP code.*

## 4.35.1 Macro Definition Documentation

### 4.35.1.1 ZIPCODERECORDSEARCH\_H

```
#define ZIPCODERECORDSEARCH_H
```

Definition at line 59 of file [ZipCodeRecordSearch.h](#).

## 4.35.2 Function Documentation

### 4.35.2.1 isNumber()

```
bool isNumber (  
    const char * s )
```

See [ZipCodeRecordSearch.cpp](#) for details.

See [ZipCodeRecordSearch.cpp](#) for details.

Checks to see if a given string is a number.

**Parameters**

<b>s</b>	
----------	--

**Returns**

true

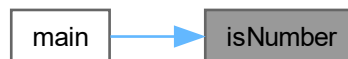
false

Definition at line 18 of file [ZipCodeRecordSearch.cpp](#).

```

00018         {
00019     try{
00020         int num = std::stoi(std::string(s));
00021         return true;
00022     } catch (std::invalid_argument& e) {
00023         return false;
00024     }
00025 }
```

Here is the caller graph for this function:

**4.35.2.2 displayHelp()**

```

void displayHelp (
    const std::string & commandName )
```

Prints usage information to the console.

**Parameters**

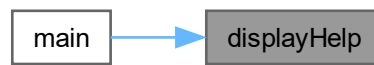
<i>commandName</i>	
--------------------	--

Definition at line 32 of file [ZipCodeRecordSearch.cpp](#).

```

00032     {
00033     std::cout << std::endl
00034     << "Usage: " << commandName << " [options]" << std::endl
00035     << "-h, --help          Show help options" << std::endl
00036     << "-Z <zipcode>," << std::endl
00037     << "--zipcode <zipcode>  Search record file for <zipcode>" << std::endl << std::endl;
00038 }
```

Here is the caller graph for this function:



#### 4.35.2.3 defaultMessage()

```
void defaultMessage (
    const std::string & commandName )
```

Prints basic message to the console.

##### Parameters

<i>commandName</i>	
--------------------	--

Definition at line 45 of file [ZipCodeRecordSearch.cpp](#).

```
00045                                     {
00046     std::cout << std::endl
00047         << "Try \" " << commandName << " -h\" for more information." << std::endl << std::endl;
00048 }
```

Here is the caller graph for this function:



#### 4.35.2.4 searchHelper()

```
void searchHelper (
    std::string fileName,
    char fileType,
    char * zip )
```

Helper function to search the buffer for a given ZIP code.

## Parameters

<i>fileName</i>	Name of the file to open.
<i>fileType</i>	Type of the file to open to pass to the indexer for the buffer.
<i>zip</i>	ZIP codes to search for, if any.

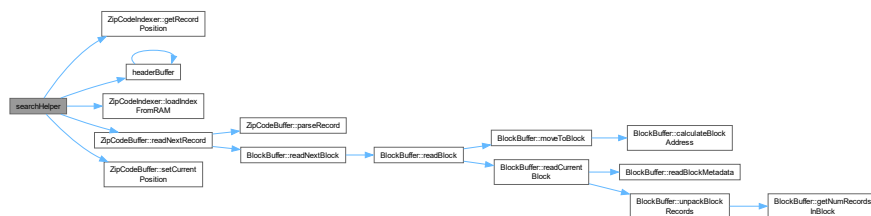
Definition at line 57 of file [ZipCodeRecordSearch.cpp](#).

```

00057
00058     // Create an index and load it from the index file
00059     std::ifstream file(fileName);
00060     HeaderBuffer headerBuffer(fileName);
00061     ZipCodeIndexer index(file, fileType, fileName + "_index.txt", headerBuffer);
00062     index.loadIndexFromRAM();
00063
00064     // Get the position of the ZIP code in the file
00065     std::streampos position = index.getRecordPosition(zip);
00066
00067     if (position != std::streampos(-1)) {
00068         // Open the buffer and set the position
00069         ZipCodeBuffer buffer(file, fileType, headerBuffer);
00070         buffer.setCurrentPosition(position);
00071
00072         // Read the record at the specified position
00073         ZipCodeRecord record = buffer.readNextRecord();
00074
00075         // Print the record info
00076         std::cout << "Zip Code: " << record.zipCode << std::endl
00077             << "Place Name: " << record.placeName << std::endl
00078             << "State: " << record.state << std::endl
00079             << "County: " << record.county << std::endl
00080             << "Latitude: " << record.latitude << " Longitude: " << record.longitude
00081             << std::endl << std::endl;
00082     }
00083     else {
00084         // If record not found, print a message
00085         std::cout << "No record of " << zip << std::endl << std::endl;
00086     }
00087 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



## 4.36 ZipCodeRecordSearch.h

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

```

00001 /*****
00002  * @file ZipCodeRecordSearch.h
00003  * @author Devon Lattery
00004  * @author Kent Biernath
00005  * @author Emma Hoffmann
00006  * @author Emily Yang, Rediet Gelaw, Bhukima Basnet
00007  * @brief Console program for searching a record file for one or more ZIP
00008  *        codes as given by the user input.
00009  * @version 1.0
00010  * @date 2023-10-16
00011  *
00012  *****/
00013  * @details
00014  * \n Searches the records for given search terms in the console using flags.
00015  * To perform a search, the -Z flag should be used followed by the ZIP code.
00016  * e.g. -Z 12345 searches for the ZIP code 12345.
00017  * \n
00018  * \n If the record is in the file, it will be displayed in the console.
00019  * e.g. -Z 12345 would return the following:
00020  *
00021  * \n Zip Code: 12345
00022  * \n Place Name: Example City
00023  * \n State: EG
00024  * \n County: Example
00025  * \n Latitude: 12.3456 Longitude: -12.3456
00026  * \n
00027  * \n If the record is not in the file, the console will display the message:
00028  * No record of 12345
00029  * \n
00030  * \n The user may enter as many searches as they like, using the -Z flag for
00031  * every search entry. This will result in multiple lines in the results
00032  * table and any records not in the file will be stated after the table.
00033  * e.g. -Z 12345 -Z 23456 -Z 34567 -Z 45678 would return the following:
00034  * \n
00035  * \n Zip Code: 12345
00036  * \n Place Name: Example City
00037  * \n State: EG
00038  * \n County: Example
00039  * \n Latitude: 12.3456 Longitude: -12.3456
00040  * \n
00041  * \n Zip Code: 34567
00042  * \n Place Name: Sample City
00043  * \n State: ST
00044  * \n County: Sample
00045  * \n Latitude: 34.5678 Longitude: -34.5678
00046  * \n
00047  * \n No record of 23456
00048  * \n
00049  * \n No record of 45678
00050  * \n
00051  * \n Assumptions:
00052  * \n -- The record file is in the same directory as the program.
00053  * \n -- The first line of the file contains the label for each column.
00054  *****/
00055
00056 #pragma once
00057
00058 #ifndef ZIPCODERECORDSEARCH_H
00059 #define ZIPCODERECORDSEARCH_H
00060
00061 #include <string>
00062
00064 bool isNumber(const char* s);
00065 void displayHelp(const std::string& commandName);
00066 void defaultMessage(const std::string& commandName);
00067 void searchHelper(std::string fileName, char fileType, char* zip);
00068
00069 #endif

```

## 4.37 ZipCodeTableViewer.cpp File Reference

Console program for displaying a table of the most eastern, western, northern, and southern ZIP codes for each state code in a file.

```

#include <iostream>
#include <fstream>

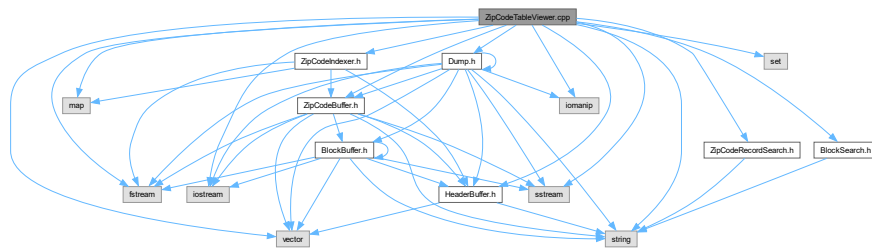
```

```

#include <sstream>
#include <string>
#include <map>
#include <vector>
#include <set>
#include <iomanip>
#include "ZipCodeBuffer.h"
#include "ZipCodeRecordSearch.h"
#include "ZipCodeIndexer.h"
#include "HeaderBuffer.h"
#include "BlockSearch.h"
#include "Dump.h"

```

Include dependency graph for ZipCodeTableViewer.cpp:



## Functions

- int [main](#) (int argc, char \*argv[])

### 4.37.1 Detailed Description

Console program for displaying a table of the most eastern, western, northern, and southern ZIP codes for each state code in a file.

#### Author

Kent Biernath  
 Andrew Clayton  
 Emma Hoffmann, Emily Yang, Devon Lattery

#### Date

2023-11-19

#### Version

3.0

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



## 4.37.2 Function Documentation

### 4.37.2.1 main()

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

Displays a table of the most eastern, western, northern, and southern ZIP codes for each state code contained in a given file.

It uses the [ZipCodeBuffer](#) class to retrieve each record as a [ZipCodeRecord](#) struct.

As it processes each record, it stores the state codes in a sorted set and the most eastern/western/northern/southern ZIP codes and coordinates in maps.

Once it has processed every record in the file, it displays a table with five columns on the console sorted alphabetically by state code.

The table columns are (in order):

- State Code
- Easternmost ZIP Code
- Westernmost ZIP Code
- Northernmost ZIP Code
- Southernmost ZIP Code

Length-indicated files also have a "Record Length" field at the start of the record.

If the program is launched with command line arguments -Z or -Zip, it will do a search. See [ZipCodeRecordSearch.cpp](#) and [BlockSearch.cpp](#) for details.

Assumptions:

- The file is in the same directory as the program.
- The file records always contain exactly six fields.
- The file has column headers on the first line.

Definition at line 66 of file [ZipCodeTableViewer.cpp](#).

```
00066         {
00067
00068         std::ifstream file;
00069         std::string fileName;
00070         char fileType = 'L'; // Default to length-indicated file type
00071
00072         // Loop until a valid file name is provided by the user
00073         while (true) {
00074             // Prompt the user for a file name until a valid one is given
00075
00076             std::cout << "Enter the file name to open: ";
00077             std::cin >> fileName;
00078
00079             // Attempt to open the file
00080             file.open(fileName);
00081
00082             if (file.is_open()) {
00083                 std::cout << "File successfully opened." << std::endl;
00084                 break; // Exit the loop when a valid file is provided
00085             }
00086             else {
00087                 std::cerr << "Failed to open the file. Please enter a valid file name." << std::endl;
00088             }
00089         }
00090
00091
00092         HeaderBuffer headerBuffer(fileName);
00093
00094         if (fileName.find(".csv") != std::string::npos)
```

```

00095     {
00096         fileType = 'C';
00097     }
00098     else
00099     {
00100         // If not a CSV, then it is either a length-indicated or blocked file with a metadata record,
so read metadata
00101         headerBuffer.readHeader();
00102         if (headerBuffer.getBlockSize() == 0)
00103         {
00104             // The file does not use blocks, so it is length-indicated
00105
00106             fileType = 'L';
00107         }
00108         else
00109         {
00110             // The file uses blocks
00111             fileType = 'B';
00112         }
00113     }
00114 }
00115
00116
00117 // Create a ZipCodeBuffer for accessing the records in the file
00118 ZipCodeBuffer recordBuffer(file, fileType, headerBuffer);
00119 ZipCodeRecord record;
00120
00121
00122 // Test code for the dumps
00123 /*
00124 ifstream dumpInputFile(fileName);
00125 ZipCodeBuffer dumpRecordBuffer(dumpInputFile, fileType, HeaderBuffer(fileName));
00126 Dump dump(dumpRecordBuffer);
00127 dump.dumpPhysicalOrder();
00128 dump.dumpBlockIndex("blocked_Index.txt");
00129 */
00130
00131
00132 // If the program is given no arguments, display the table
00133 if (argc == 1) {
00134
00135     // Make a set and maps to store the state code and ZIP code coordinate extrema
00136     std::set<std::string> stateCodes;
00137     std::map<std::string, std::vector<double>> stateCodeToCoordinatesMap;
00138     std::map<std::string, std::vector<std::string>> stateCodeToZipCodesMap;
00139
00140     // Iterate through records until the terminal string "" is returned from the buffer
00141     while (true)
00142     {
00143         ZipCodeRecord record = recordBuffer.readNextRecord();
00144         if (record.zipCode == "") {
00145             // Exit the loop if the terminal string "" was returned from the buffer
00146             break;
00147         }
00148
00149         // Try to add the state initial to the set and save the boolean result of whether it
succeeded
00150         std::pair<std::set<std::string>::iterator, bool> result = stateCodes.insert(record.state);
00151
00152         if (result.second) {
00153             // The initial was not already present in the set, so add it to the maps too
00154
00155             // Create a vector with the current longitude and latitude values in all four extrema
00156             std::vector<double> coordinates;
00157             coordinates.push_back(record.longitude); // [0] Easternmost
00158             coordinates.push_back(record.longitude); // [1] Westernmost
00159             coordinates.push_back(record.latitude); // [2] Northernmost
00160             coordinates.push_back(record.latitude); // [3] Southernmost
00161             stateCodeToCoordinatesMap[record.state] = coordinates;
00162
00163             // Create a vector with the current ZIP code in all four extrema
00164             std::vector<std::string> zipCodes;
00165             zipCodes.push_back(record.zipCode); // [0] Easternmost
00166             zipCodes.push_back(record.zipCode); // [1] Westernmost
00167             zipCodes.push_back(record.zipCode); // [2] Northernmost
00168             zipCodes.push_back(record.zipCode); // [3] Southernmost
00169             stateCodeToZipCodesMap[record.state] = zipCodes;
00170         }
00171         else {
00172             if (record.longitude < stateCodeToCoordinatesMap[record.state][0]) {
00173                 // New Easternmost (least longitude)
00174                 stateCodeToCoordinatesMap[record.state][0] = record.longitude;
00175                 stateCodeToZipCodesMap[record.state][0] = record.zipCode;
00176             }
00177             else if (record.longitude > stateCodeToCoordinatesMap[record.state][1]) {
00178                 // New Westernmost
00179                 stateCodeToCoordinatesMap[record.state][1] = record.longitude;

```

```

00180         stateCodeToZipCodesMap[record.state][1] = record.zipCode;
00181     }
00182
00183     if (record.latitude > stateCodeToCoordinatesMap[record.state][2]) {
00184         // New Northernmost (greatest latitude)
00185         stateCodeToCoordinatesMap[record.state][2] = record.latitude;
00186         stateCodeToZipCodesMap[record.state][2] = record.zipCode;
00187     }
00188     else if (record.latitude < stateCodeToCoordinatesMap[record.state][3]) {
00189         // New Southernmost
00190         stateCodeToCoordinatesMap[record.state][3] = record.latitude;
00191         stateCodeToZipCodesMap[record.state][3] = record.zipCode;
00192     }
00193 }
00194 }
00195
00196 // Display the table column headers
00197 std::cout << std::left << std::setw(8) << "State"
00198           << std::left << std::setw(8) << "East"
00199           << std::left << std::setw(8) << "West"
00200           << std::left << std::setw(8) << "North"
00201           << std::left << std::setw(8) << "South" << std::endl;
00202 // Display the spacers below the table column headers
00203 for (size_t i = 0; i < 5; i++)
00204 {
00205     std::cout << std::left << std::setw(8) << "-----";
00206 }
00207 std::cout << std::endl;
00208
00209 // Display the records in the table sorted alphabetically by state name.
00210 // Displays State, Easternmost ZIP Code, Westernmost ZIP Code, Northernmost ZIP Code, and
Southernmost ZIP Code per row
00211 for (const std::string& stateCode : stateCodes) {
00212
00213     std::cout << std::left << std::setw(8) << stateCode;
00214     for (const std::string& zipCode : stateCodeToZipCodesMap[stateCode]) {
00215         std::cout << std::setw(8) << zipCode;
00216     }
00217     std::cout << std::endl;
00218 }
00219 }
00220 else
00221 {
00222     // If command line parameters were given, do a search.
00223
00224     if (fileType != 'B') {
00225         // Generate an index
00226         std::ifstream searchFile(fileName);
00227         ZipCodeIndexer index(searchFile, fileType, fileName + "_index.txt", headerBuffer);
00228         index.createIndex();
00229         index.writeIndexToFile();
00230
00231         const std::string COMMAND_NAME = std::string(argv[0]);
00232         // If no flags are used, display the default message
00233         if (argc == 1) {
00234             defaultMessage(COMMAND_NAME);
00235             return 0;
00236         } // If a help flag is used, display usage information
00237         else if (std::string(argv[1]) == "-h" || std::string(argv[1]) == "--help") {
00238             displayHelp(COMMAND_NAME);
00239             return 0;
00240         }
00241         else {
00242             std::string flag = "";
00243             for (int i = 1; i < argc; i++) {
00244                 if (flag == "") {
00245                     flag = std::string(argv[i]);
00246                 }
00247                 else if ((flag == "-Z" || flag == "-z" || flag == "--zipcode") &&
isNumber(argv[i])) {
00248                     searchHelper(fileName, fileType, argv[i]);
00249                     flag = "";
00250                 }
00251                 else {
00252                     std::cerr << "INVALID ARGUMENT" << std::endl;
00253                     defaultMessage(COMMAND_NAME);
00254                     return 1;
00255                 }
00256             }
00257         }
00258     }
00259     else // else fileType == B
00260     {
00261         // Run blocked file search
00262
00263         for (int i = 1; i < argc; ++i) {
00264             string arg = argv[i];

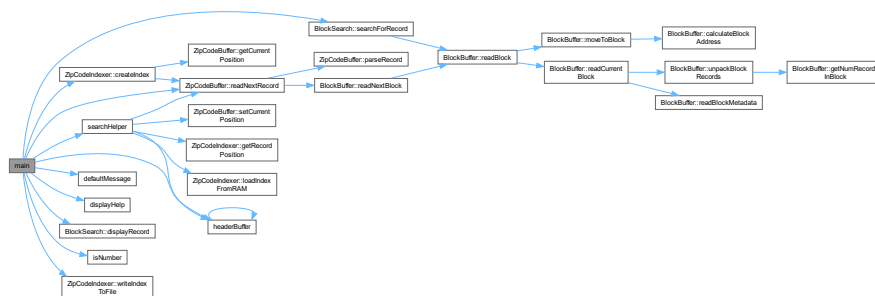
```

```

00265
00266 // Check if argument starts with -z or -Z
00267 if (arg.size() > 2 && (arg[0] == '-' && (arg[1] == 'z' || arg[1] == 'Z'))) {
00268     string zipcodeStr = arg.substr(2); // Extract the zipcode part
00269     int zipcode;
00270
00271     try {
00272         BlockSearch searcher;
00273         zipcode = stoi(zipcodeStr);
00274         string result = searcher.searchForRecord(zipcode);
00275
00276         if (result != "-1") {
00277             cout << "Information for zipcode " << zipcode << ":\n";
00278             searcher.displayRecord(result);
00279         } else {
00280             cout << "Zipcode " << zipcode << " not found." << "\n\n";
00281         }
00282     } catch (const invalid_argument& ia) {
00283         cerr << "Invalid zipcode format: " << zipcodeStr << endl;
00284     }
00285 } else {
00286     // Invalid argument format
00287     cout << "Invalid argument: " << arg << endl;
00288     cout << "Please use the format: -z<zipcode> or -Z<zipcode>" << endl;
00289 }
00290 }
00291 }
00292 }
00293
00294 file.close();
00295 return 0;
00296 }

```

Here is the call graph for this function:



## 4.38 ZipCodeTableViewer.cpp

Go to the documentation of this file.

```

00001 // -----
00012 // -----
00047 // -----
00048
00049 #include <iostream>
00050 #include <fstream>
00051 #include <sstream>
00052 #include <string>
00053 #include <map>
00054 #include <vector>
00055 #include <set>
00056 #include <iomanip>
00057 #include "ZipCodeBuffer.h"
00058 #include "ZipCodeRecordSearch.h"
00059 #include "ZipCodeIndexer.h"
00060 #include "HeaderBuffer.h"
00061 #include "BlockSearch.h"
00062 #include "Dump.h"
00063
00064
00065
00066 int main(int argc, char* argv[]) {
00067

```

```

00068     std::ifstream file;
00069     std::string fileName;
00070     char fileType = 'L'; // Default to length-indicated file type
00071
00072     // Loop until a valid file name is provided by the user
00073     while (true) {
00074         // Prompt the user for a file name until a valid one is given
00075
00076         std::cout << "Enter the file name to open: ";
00077         std::cin >> fileName;
00078
00079         // Attempt to open the file
00080         file.open(fileName);
00081
00082         if (file.is_open()) {
00083             std::cout << "File successfully opened." << std::endl;
00084             break; // Exit the loop when a valid file is provided
00085         }
00086         else {
00087             std::cerr << "Failed to open the file. Please enter a valid file name." << std::endl;
00088         }
00089     }
00090
00091     HeaderBuffer headerBuffer(fileName);
00092
00093     if (fileName.find(".csv") != std::string::npos)
00094     {
00095         fileType = 'C';
00096     }
00097     else
00098     {
00099         // If not a CSV, then it is either a length-indicated or blocked file with a metadata record,
00100         so read metadata
00101         headerBuffer.readHeader();
00102         if (headerBuffer.getBlockSize() == 0)
00103         {
00104             // The file does not use blocks, so it is length-indicated
00105
00106             fileType = 'L';
00107         }
00108         else
00109         {
00110             // The file uses blocks
00111             fileType = 'B';
00112         }
00113     }
00114 }
00115
00116 // Create a ZipCodeBuffer for accessing the records in the file
00117 ZipCodeBuffer recordBuffer(file, fileType, headerBuffer);
00118 ZipCodeRecord record;
00119
00120 // Test code for the dumps
00121 /*
00122 ifstream dumpInputFile(fileName);
00123 ZipCodeBuffer dumpRecordBuffer(dumpInputFile, fileType, HeaderBuffer(fileName));
00124 Dump dump(dumpRecordBuffer);
00125 dump.dumpPhysicalOrder();
00126 dump.dumpBlockIndex("blocked_Index.txt");
00127 */
00128
00129 // If the program is given no arguments, display the table
00130 if (argc == 1) {
00131
00132     // Make a set and maps to store the state code and ZIP code coordinate extrema
00133     std::set<std::string> stateCodes;
00134     std::map<std::string, std::vector<double>> stateCodeToCoordinatesMap;
00135     std::map<std::string, std::vector<std::string>> stateCodeToZipCodesMap;
00136
00137     // Iterate through records until the terminal string "" is returned from the buffer
00138     while (true)
00139     {
00140         ZipCodeRecord record = recordBuffer.readNextRecord();
00141         if (record.zipCode == "") {
00142             // Exit the loop if the terminal string "" was returned from the buffer
00143             break;
00144         }
00145
00146         // Try to add the state initial to the set and save the boolean result of whether it
00147         succeeded
00148         std::pair<std::set<std::string>::iterator, bool> result = stateCodes.insert(record.state);
00149         if (result.second) {

```

```

00153         // The initial was not already present in the set, so add it to the maps too
00154
00155         // Create a vector with the current longitude and latitude values in all four extrema
00156         std::vector<double> coordinates;
00157         coordinates.push_back(record.longitude); // [0] Easternmost
00158         coordinates.push_back(record.longitude); // [1] Westernmost
00159         coordinates.push_back(record.latitude); // [2] Northernmost
00160         coordinates.push_back(record.latitude); // [3] Southernmost
00161         stateCodeToCoordinatesMap[record.state] = coordinates;
00162
00163         // Create a vector with the current ZIP code in all four extrema
00164         std::vector<std::string> zipCodes;
00165         zipCodes.push_back(record.zipCode); // [0] Easternmost
00166         zipCodes.push_back(record.zipCode); // [1] Westernmost
00167         zipCodes.push_back(record.zipCode); // [2] Northernmost
00168         zipCodes.push_back(record.zipCode); // [3] Southernmost
00169         stateCodeToZipCodesMap[record.state] = zipCodes;
00170     }
00171     else {
00172         if (record.longitude < stateCodeToCoordinatesMap[record.state][0]) {
00173             // New Easternmost (least longitude)
00174             stateCodeToCoordinatesMap[record.state][0] = record.longitude;
00175             stateCodeToZipCodesMap[record.state][0] = record.zipCode;
00176         }
00177         else if (record.longitude > stateCodeToCoordinatesMap[record.state][1]) {
00178             // New Westernmost
00179             stateCodeToCoordinatesMap[record.state][1] = record.longitude;
00180             stateCodeToZipCodesMap[record.state][1] = record.zipCode;
00181         }
00182
00183         if (record.latitude > stateCodeToCoordinatesMap[record.state][2]) {
00184             // New Northernmost (greatest latitude)
00185             stateCodeToCoordinatesMap[record.state][2] = record.latitude;
00186             stateCodeToZipCodesMap[record.state][2] = record.zipCode;
00187         }
00188         else if (record.latitude < stateCodeToCoordinatesMap[record.state][3]) {
00189             // New Southernmost
00190             stateCodeToCoordinatesMap[record.state][3] = record.latitude;
00191             stateCodeToZipCodesMap[record.state][3] = record.zipCode;
00192         }
00193     }
00194 }
00195
00196 // Display the table column headers
00197 std::cout << std::left << std::setw(8) << "State"
00198           << std::left << std::setw(8) << "East"
00199           << std::left << std::setw(8) << "West"
00200           << std::left << std::setw(8) << "North"
00201           << std::left << std::setw(8) << "South" << std::endl;
00202 // Display the spacers below the table column headers
00203 for (size_t i = 0; i < 5; i++)
00204 {
00205     std::cout << std::left << std::setw(8) << "-----";
00206 }
00207 std::cout << std::endl;
00208
00209 // Display the records in the table sorted alphabetically by state name.
00210 // Displays State, Easternmost ZIP Code, Westernmost ZIP Code, Northernmost ZIP Code, and
Southernmost ZIP Code per row
00211 for (const std::string& stateCode : stateCodes) {
00212
00213     std::cout << std::left << std::setw(8) << stateCode;
00214     for (const std::string& zipCode : stateCodeToZipCodesMap[stateCode]) {
00215         std::cout << std::setw(8) << zipCode;
00216     }
00217     std::cout << std::endl;
00218 }
00219 }
00220 else
00221 {
00222     // If command line parameters were given, do a search.
00223
00224     if (fileType != 'B') {
00225         // Generate an index
00226         std::ifstream searchFile(fileName);
00227         ZipCodeIndexer index(searchFile, fileType, fileName + "_index.txt", headerBuffer);
00228         index.createIndex();
00229         index.writeIndexToFile();
00230
00231         const std::string COMMAND_NAME = std::string(argv[0]);
00232         // If no flags are used, display the default message
00233         if (argc == 1) {
00234             defaultMessage(COMMAND_NAME);
00235             return 0;
00236         } // If a help flag is used, display usage information
00237         else if (std::string(argv[1]) == "-h" || std::string(argv[1]) == "--help") {
00238             displayHelp(COMMAND_NAME);

```

```

00239         return 0;
00240     }
00241     else {
00242         std::string flag = "";
00243         for (int i = 1; i < argc; i++) {
00244             if (flag == "") {
00245                 flag = std::string(argv[i]);
00246             }
00247             else if ((flag == "-Z" || flag == "-z" || flag == "--zipcode") &&
isNumber(argv[i])) {
00248                 searchHelper(fileName, fileType, argv[i]);
00249                 flag = "";
00250             }
00251             else {
00252                 std::cerr << "INVALID ARGUMENT" << std::endl;
00253                 defaultMessage(COMMAND_NAME);
00254                 return 1;
00255             }
00256         }
00257     }
00258 }
00259 else // else fileType == B
00260 {
00261     // Run blocked file search
00262
00263     for (int i = 1; i < argc; ++i) {
00264         string arg = argv[i];
00265
00266         // Check if argument starts with -z or -Z
00267         if (arg.size() > 2 && (arg[0] == '-' && (arg[1] == 'z' || arg[1] == 'Z'))) {
00268             string zipcodeStr = arg.substr(2); // Extract the zipcode part
00269             int zipcode;
00270
00271             try {
00272                 BlockSearch searcher;
00273                 zipcode = stoi(zipcodeStr);
00274                 string result = searcher.searchForRecord(zipcode);
00275
00276                 if (result != "-1") {
00277                     cout << "Information for zipcode " << zipcode << ":\n";
00278                     searcher.displayRecord(result);
00279                 } else {
00280                     cout << "Zipcode " << zipcode << " not found." << "\n\n";
00281                 }
00282             } catch (const invalid_argument& ia) {
00283                 cerr << "Invalid zipcode format: " << zipcodeStr << endl;
00284             }
00285         } else {
00286             // Invalid argument format
00287             cout << "Invalid argument: " << arg << endl;
00288             cout << "Please use the format: -z<zipcode> or -Z<zipcode>" << endl;
00289         }
00290     }
00291 }
00292 }
00293
00294 file.close();
00295 return 0;
00296 }

```

