Processing CSV and generating blocked sequence files with given parameters

Group 7

Generated by Doxygen 1.9.8

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 BFile Class Reference

`#include <BFile.h>`

Collaboration diagram for BFile:

Public Member Functions

- BFile ()

  Constructs a new BlockFile object with default settings.
- BFile (string fileName)

  Constructs a BlockFile object and opens a specific file.
- void lengthIndexToBlock (string indexString, string lengthData)

Converts a length index to a block structure.

- void open (string fileName)

  Opens a file for reading and writing operations.
- void close ()

  Closes the currently opened file.
- void readHeader ()

  Reads the header information from the current file.
- string writeHeader ()

  Writes header information to the current file.
- string physicalDump ()

  Provides a physical representation of the file's data.
- string logicalDump ()

  Provides a logical representation of the file's data.
- bool split (Block &b)

  Splits a block into two separate blocks.
- bool addRecord (ZipCode &zipCode)

  Adds a new ZipCode record to the file.
- bool deleteRecord (string zipCode)

  Deletes a ZipCode record from the file.
- int getFirstRBN () const

  Retrieves the first relative block number (RBN) in the file.
- int getAvailableSpace () const

  Retrieves the available space indicator in the file.

Private Attributes

- int firstRBN
- int availableSpace
- int totalBlocks
- int totalRecords
- ofstream outputData
- ifstream inputData
- BlockBuffer blockBuffer
- BlockIndex blockIndex

3.1.1 Detailed Description

Definition at line 16 of file BFile.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 BFile() [1/2]

```
BFile::BFile ( )  [inline]
```

Constructs a new BlockFile object with default settings.

Default constructor for BFile. Initializes the class without opening a file.

Definition at line 21 of file BFile.h.

3.1.2.2 BFile() [2/2]

```
BFile::BFile (
            string fileName ) [inline]
```

Constructs a BlockFile object and opens a specific file.

Parameters

fileName	The name of the file to be opened.

Definition at line 29 of file BFile.h.

Here is the call graph for this function:

3.1.3 Member Function Documentation

3.1.3.1 addRecord()

```
bool BFile::addRecord (
            ZipCode & z )
```

Adds a new ZipCode record to the file.

Adds a new ZipCode record to the block file.

Parameters

zipCode	The ZipCode object to be added.

Returns

True if the record was added successfully, false otherwise.

Parameters

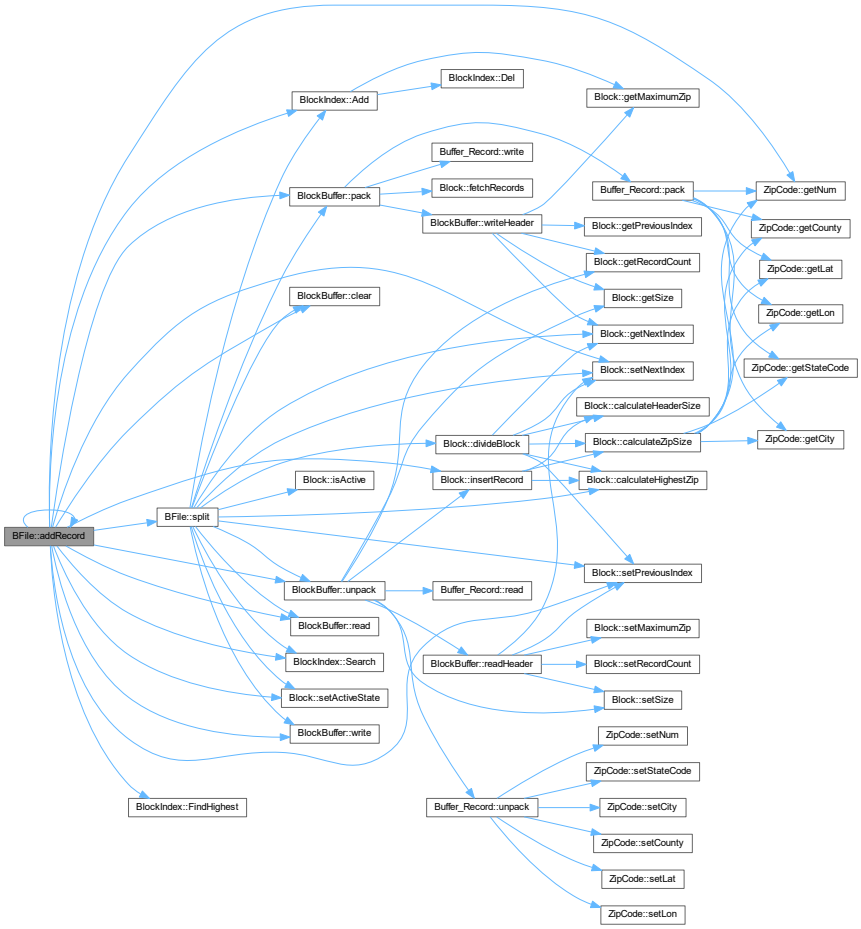
z	The ZipCode object to be added.

Returns

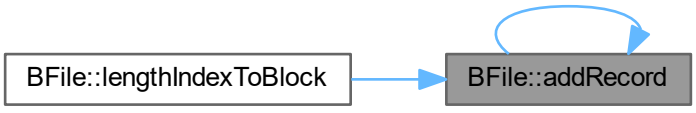
True if the record was added successfully, false otherwise.

Definition at line 120 of file BFile.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.1.3.2 close()

```
void BFile::close ( )  [inline]
```

Closes the currently opened file.

Definition at line 52 of file BFile.h.

3.1.3.3 deleteRecord()

```
bool BFile::deleteRecord (
            string zipCode )
```

Deletes a ZipCode record from the file.

Deletes a record based on the given zip code.

Parameters

zipCode	The zip code of the record to delete.

Returns

True if the record was deleted successfully, false otherwise.

Parameters

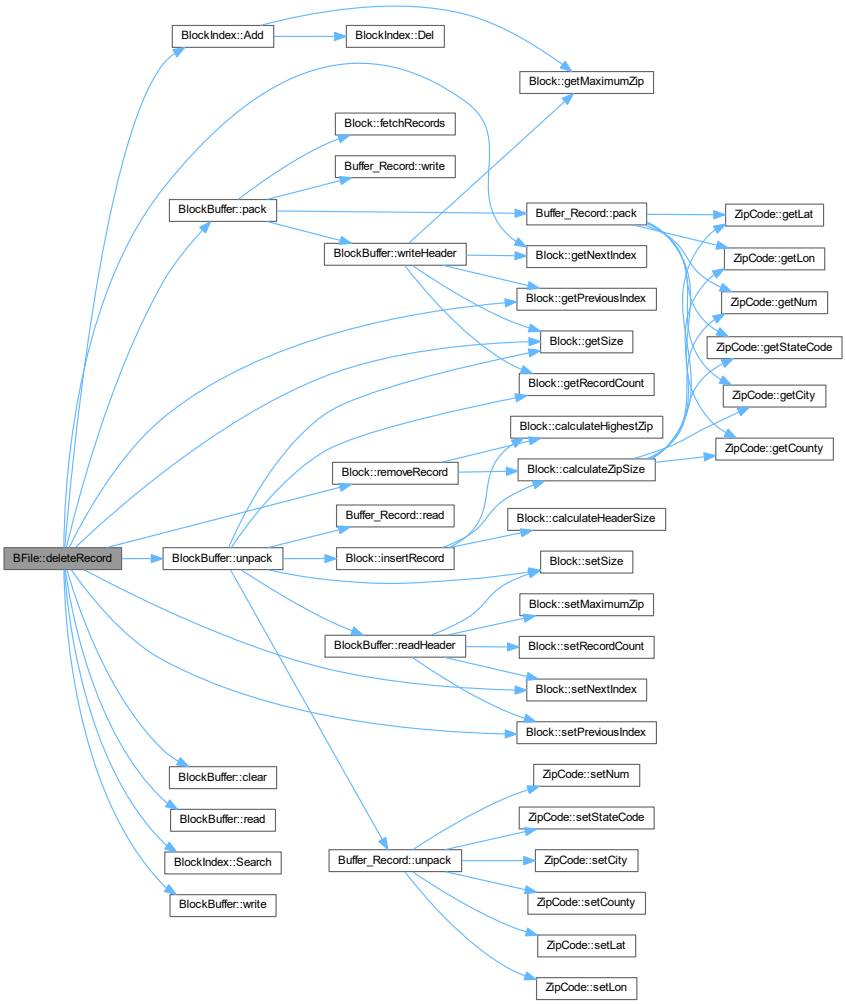
zipCode	The zip code of the record to be deleted.

Returns

True if deletion is successful, false otherwise.

Definition at line 60 of file BFile.cpp.

Here is the call graph for this function:



3.1.3.4 getAvailableSpace()

```
int BFile::getAvailableSpace ( ) const  [inline]
```

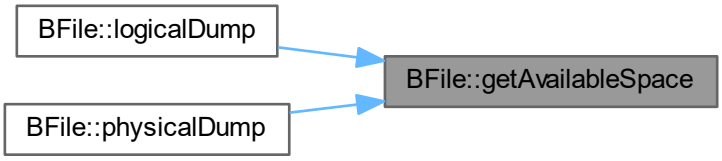
Retrieves the available space indicator in the file.

Returns

The available space as an integer.

Definition at line 111 of file BFile.h.

Here is the caller graph for this function:



3.1.3.5 getFirstRBN()

```
int BFile::getFirstRBN ( ) const  [inline]
```

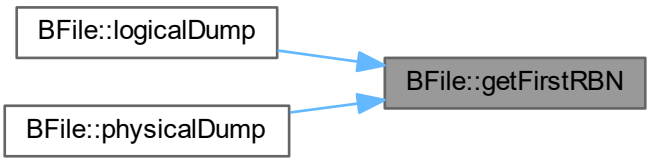
Retrieves the first relative block number (RBN) in the file.

Returns

The first RBN as an integer.

Definition at line 105 of file BFile.h.

Here is the caller graph for this function:



3.1.3.6 lengthIndexToBlock()

```
void BFile::lengthIndexToBlock (
            string indexString,
            string lengthData )
```

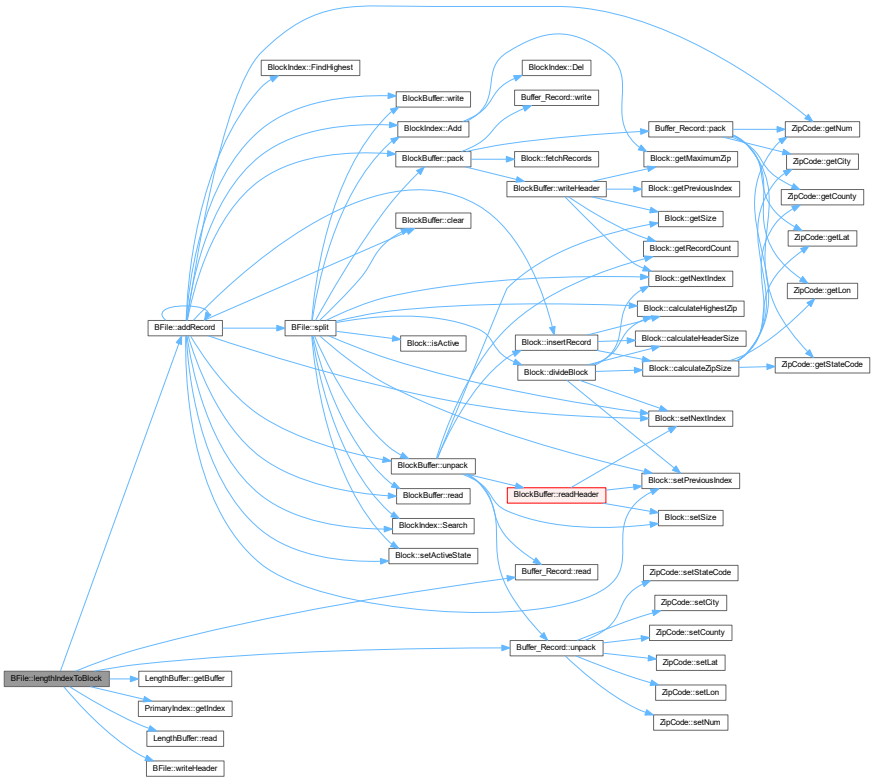
Converts a length index to a block structure.

Parameters

indexString	The string index to be converted.
lengthData	The length indicated data to be processed.

Definition at line 32 of file BFile.cpp.

Here is the call graph for this function:



3.1.3.7 logicalDump()

```
string BFile::logicalDump ( )
```

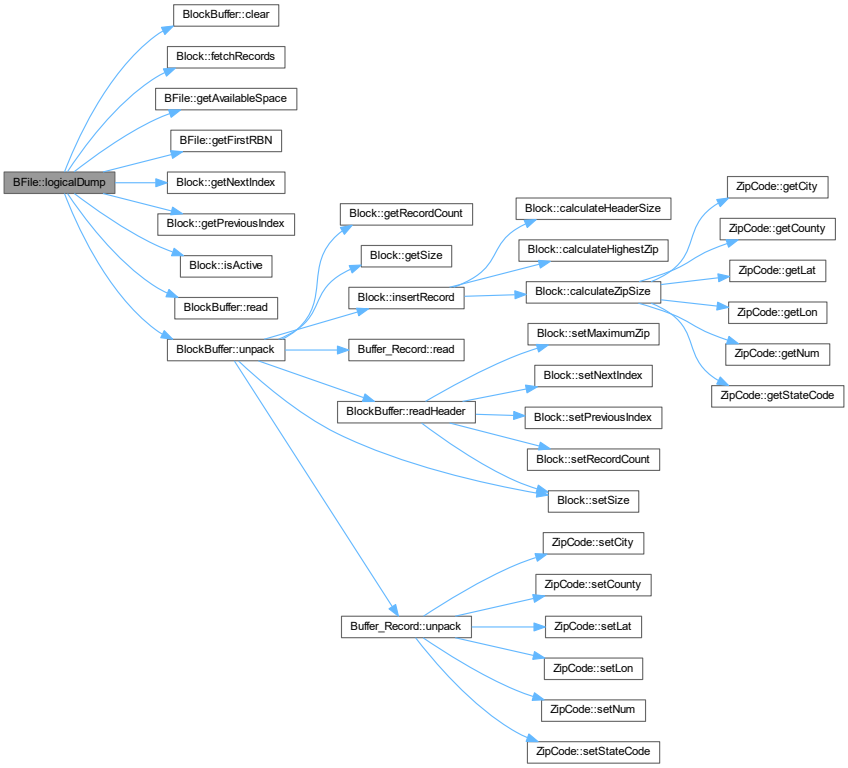
Provides a logical representation of the file's data.

Returns

String containing a logical dump of the file.

Definition at line 306 of file BFile.cpp.

Here is the call graph for this function:



3.1.3.8 open()

```
void BFile::open (
            string fileName )  [inline]
```

Opens a file for reading and writing operations.

Parameters

fileName	The name of the file to open.

Definition at line 44 of file BFile.h.

Here is the caller graph for this function:

3.1.3.9 physicalDump()

```
string BFile::physicalDump ( )
```

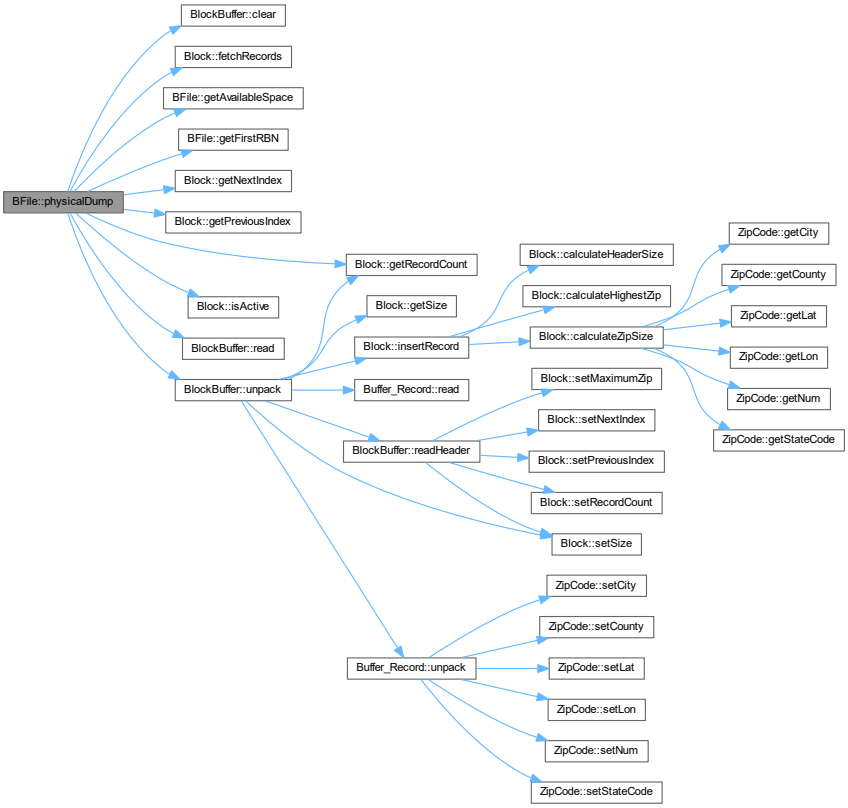
Provides a physical representation of the file's data.

Returns

String containing a physical dump of the file.

Definition at line 263 of file BFile.cpp.

Here is the call graph for this function:



3.1.3.10 readHeader()

```
void BFile::readHeader ( )
```

Reads the header information from the current file.

Definition at line 180 of file BFile.cpp.

3.1.3.11 split()

```
bool BFile::split (
            Block & b )
```

Splits a block into two separate blocks.

Parameters

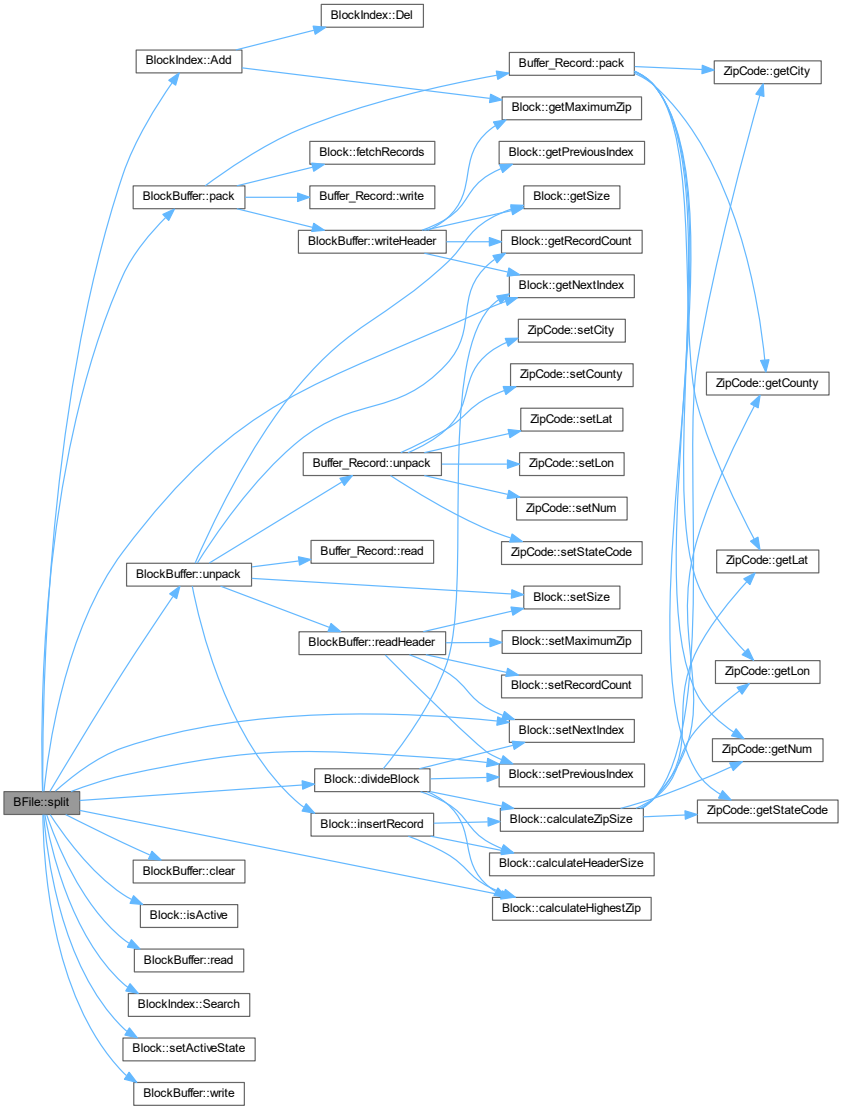
b	The block to be split.

Returns

True if the split was successful, false otherwise.

Definition at line 352 of file BFile.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

3.1.3.12 writeHeader()

```
string BFile::writeHeader ( )
```

Writes header information to the current file.

Returns

A string representation of the header.

Definition at line 193 of file BFile.cpp.

Here is the caller graph for this function:

3.1.4 Member Data Documentation

3.1.4.1 availableSpace

```
int BFile::availableSpace  [private]
```

Definition at line 114 of file BFile.h.

3.1.4.2 blockBuffer

```
BlockBuffer BFile::blockBuffer  [private]
```

Definition at line 118 of file BFile.h.

3.1.4.3 blockIndex

```
BlockIndex BFile::blockIndex  [private]
```

Definition at line 119 of file BFile.h.

3.1.4.4 firstRBN

```
int BFile::firstRBN  [private]
```

Definition at line 114 of file BFile.h.

3.1.4.5 inputData

```
ifstream BFile::inputData  [private]
```

Definition at line 117 of file BFile.h.

3.1.4.6 outputData

```
ofstream BFile::outputData  [private]
```

Definition at line 116 of file BFile.h.

3.1.4.7 totalBlocks

```
int BFile::totalBlocks  [private]
```

Definition at line 114 of file BFile.h.

3.1.4.8 totalRecords

```
int BFile::totalRecords  [private]
```

Definition at line 114 of file BFile.h.

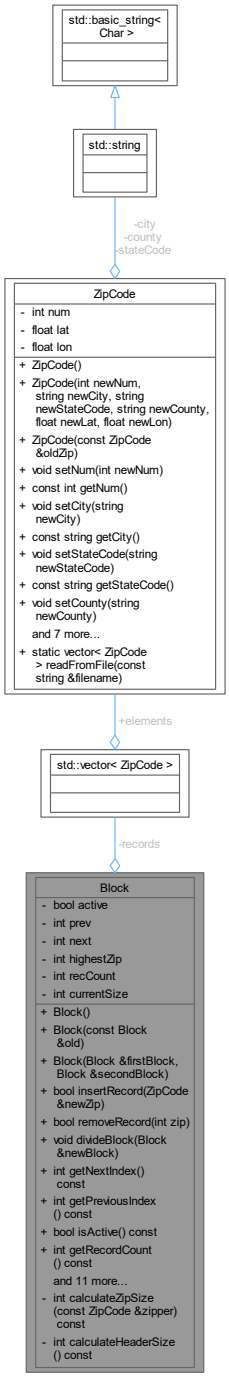
The documentation for this class was generated from the following files:

- BFile.h
- BFile.cpp

3.2 Block Class Reference

`#include <Block.h>`

Collaboration diagram for Block:



Public Member Functions

- Block ()

Default constructor.

- Block (const Block &old)

  Copy constructor.
- Block (Block &firstBlock, Block &secondBlock)

  Merge constructor, merges two Blocks into one.
- bool insertRecord (ZipCode &newZip)

  Inserts a new ZipCode record into the Block.
- bool removeRecord (int zip)

  Removes a ZipCode record from the Block.
- void divideBlock (Block &newBlock)

  Splits the Block into two parts.
- int getNextIndex () const

  Get the next block index.
- int getPreviousIndex () const

  Get the previous block index.
- bool isActive () const

  Check if the block is active.
- int getRecordCount () const

  Get the record count of the block.
- int getSize () const

  Get the current size of the block.
- int getMaximumZip () const

  Get the highest ZIP code in the block.
- void fetchRecords (vector< ZipCode > &recordsOut) const

  Retrieves all ZipCode records in the block.
- bool searchZip (ZipCode &resultZip, int target)

  Searches for a specific ZipCode in the block based on the given target number.
- void setActiveState (bool state)
- void setNextIndex (int next)
- void setPreviousIndex (int prev)
- void setRecordCount (int recCount)
- void setSize (int currentSize)
- void setMaximumZip (int highestZip)
- int calculateHighestZip ()

  Calculates the highest ZIP code among the records in the block.

Private Member Functions

- int calculateZipSize (const ZipCode &zipper) const

  Calculates the size of a ZipCode record.
- int calculateHeaderSize () const

  Calculates the size of the block header.

Private Attributes

- bool active
- int prev
- int next
- int highestZip
- int recCount
- int currentSize
- vector< ZipCode > records

3.2.1 Detailed Description

Definition at line 19 of file Block.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Block() [1/3]

```
Block::Block ( )
```

Default constructor.

Constructor that initializes a new, empty block.

Postcondition

Creates an empty Block.

Creates a block with default initial values and clears any existing records.

Definition at line 17 of file Block.cpp.

Here is the call graph for this function:

3.2.2.2 Block() [2/3]

```
Block::Block (
            const Block & old )
```

Copy constructor.

Copy constructor for creating a copy of an existing block.

Precondition

Requires an existing Block object.

Postcondition

Creates a new Block as a copy of the given Block.

Parameters

old	A constant reference to the Block object to be copied.

Postcondition

Creates a new Block object as a copy of the provided Block.

Definition at line 33 of file Block.cpp.

3.2.2.3 Block() [3/3]

```
Block::Block (
            Block & firstBlock,
            Block & secondBlock )
```

Merge constructor, merges two Blocks into one.

Merge constructor that merges two Blocks into one.

Precondition

Requires two Block objects to be merged.

Postcondition

Merges the contents of the two Blocks into one.

Parameters

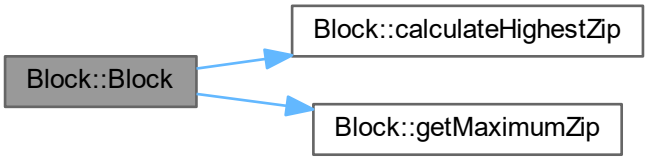
firstBlock	A reference to the first Block object to be merged.
secondBlock	A reference to the second Block object to be merged.

Postcondition

Merges the contents of both Blocks into a single new Block.

Definition at line 50 of file Block.cpp.

Here is the call graph for this function:



3.2.3 Member Function Documentation

3.2.3.1 calculateHeaderSize()

```
int Block::calculateHeaderSize ( ) const  [private]
```

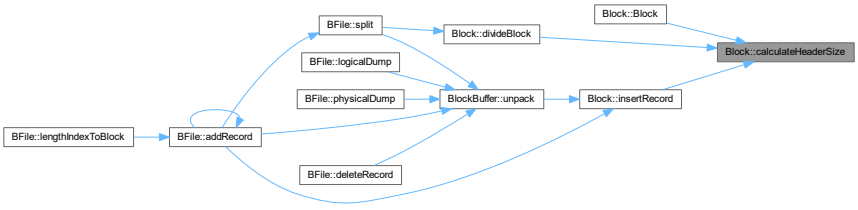
Calculates the size of the block header.

Returns

The size of the block header in bytes.

Definition at line 168 of file Block.cpp.

Here is the caller graph for this function:



3.2.3.2 calculateHighestZip()

```
int Block::calculateHighestZip ( )
```

Calculates the highest ZIP code among the records in the block.

Returns

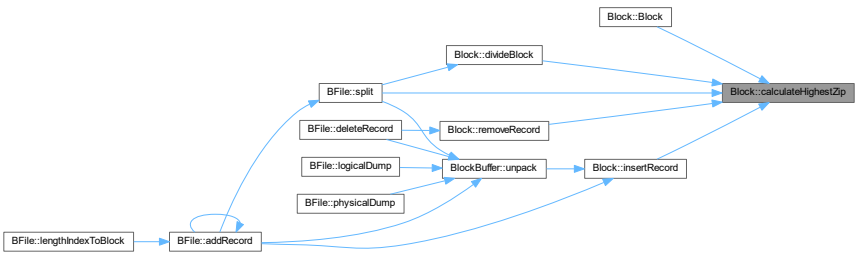
The highest ZIP code value in the block.

Postcondition

Sets the highestZip member to the highest ZIP code found.

Definition at line 144 of file Block.cpp.

Here is the caller graph for this function:



3.2.3.3 calculateZipSize()

```
int Block::calculateZipSize (
            const ZipCode & zipper ) const  [private]
```

Calculates the size of a ZipCode record.

Parameters

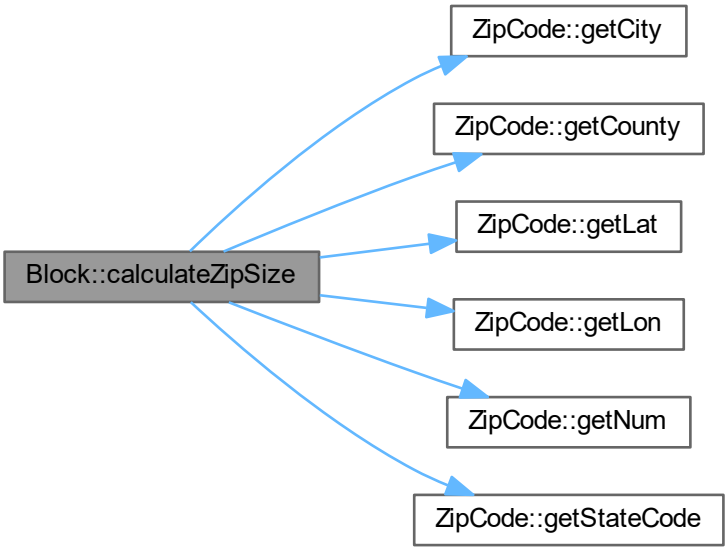
zipper	A constant reference to the ZipCode object.

Returns

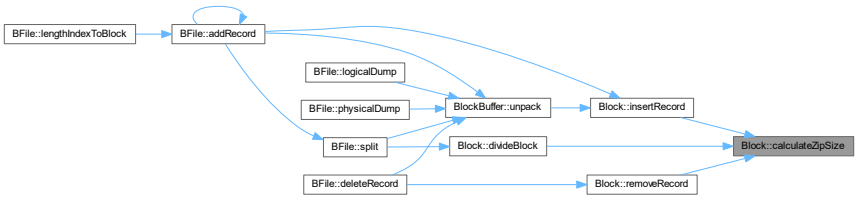
The size of the ZipCode record in bytes.

Definition at line 156 of file Block.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.2.3.4 divideBlock()

```
void Block::divideBlock (
            Block & newBlock )
```

Splits the Block into two parts.

Splits the current Block into two by dividing its records.

Precondition

Requires an empty Block to split the contents into.

Postcondition

Divides the records between the current and the new Block.

Parameters

newBlock	A reference to the Block where the second half of records will be moved.

Precondition

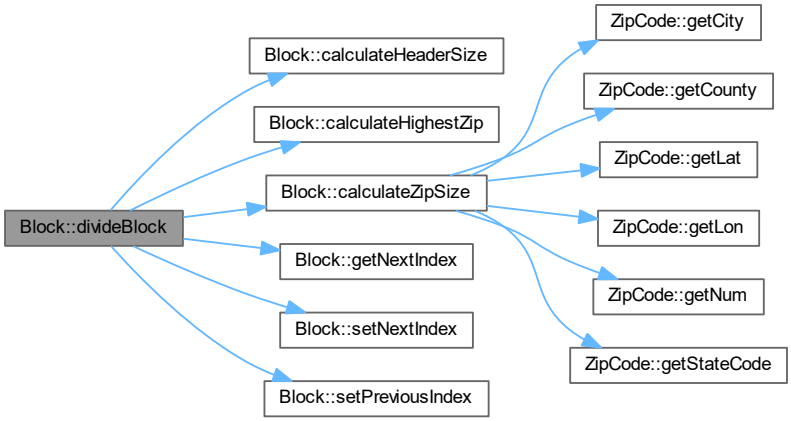
Assumes that newBlock is empty.

Postcondition

Divides the records between the current Block and newBlock.

Definition at line 100 of file Block.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

3.2.3.5 fetchRecords()

```
void Block::fetchRecords (
            vector< ZipCode > & recordsOut ) const
```

Retrieves all ZipCode records in the block.

Parameters

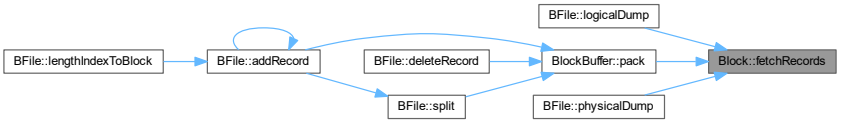
recordsOut	A vector reference to store the fetched records.

Postcondition

The recordsOut vector contains all the records from the block.

Definition at line 180 of file Block.cpp.

Here is the caller graph for this function:



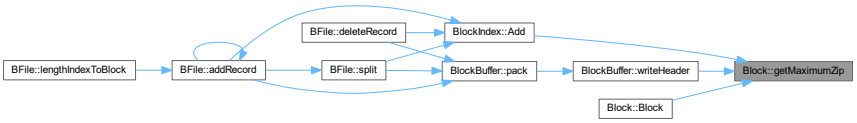
3.2.3.6 getMaximumZip()

```
int Block::getMaximumZip ( ) const  [inline]
```

Get the highest ZIP code in the block.

Definition at line 97 of file Block.h.

Here is the caller graph for this function:



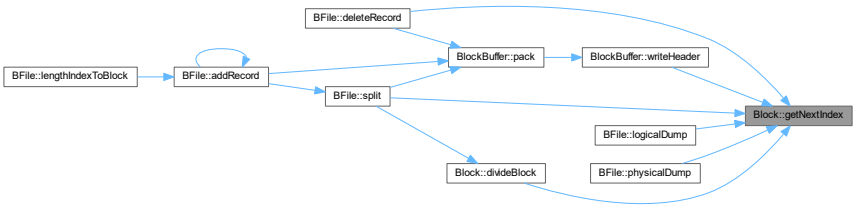
3.2.3.7 getNextIndex()

```
int Block::getNextIndex ( ) const  [inline]
```

Get the next block index.

Definition at line 72 of file Block.h.

Here is the caller graph for this function:



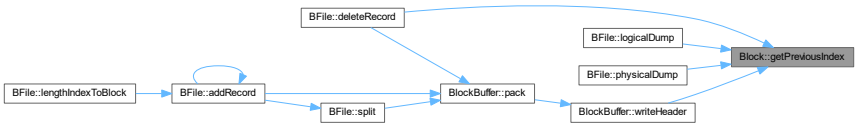
3.2.3.8 getPreviousIndex()

```
int Block::getPreviousIndex ( ) const  [inline]
```

Get the previous block index.

Definition at line 77 of file Block.h.

Here is the caller graph for this function:



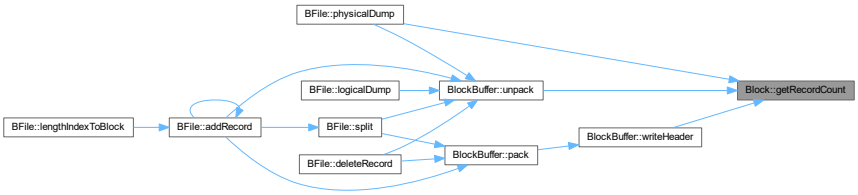
3.2.3.9 getRecordCount()

```
int Block::getRecordCount ( ) const  [inline]
```

Get the record count of the block.

Definition at line 87 of file Block.h.

Here is the caller graph for this function:



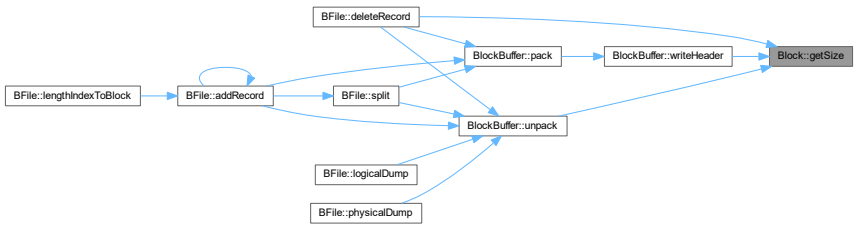
3.2.3.10 getSize()

```
int Block::getSize ( ) const  [inline]
```

Get the current size of the block.

Definition at line 92 of file Block.h.

Here is the caller graph for this function:



3.2.3.11 insertRecord()

```
bool Block::insertRecord (
            ZipCode & newZip )
```

Inserts a new ZipCode record into the Block.

Precondition

The Block should not exceed a certain size limit.

Postcondition

Adds a record to the Block, returns true if successful, false otherwise.

Parameters

newZip	A reference to the ZipCode object to be added.

Precondition

The Block should not exceed a certain size limit.

Postcondition

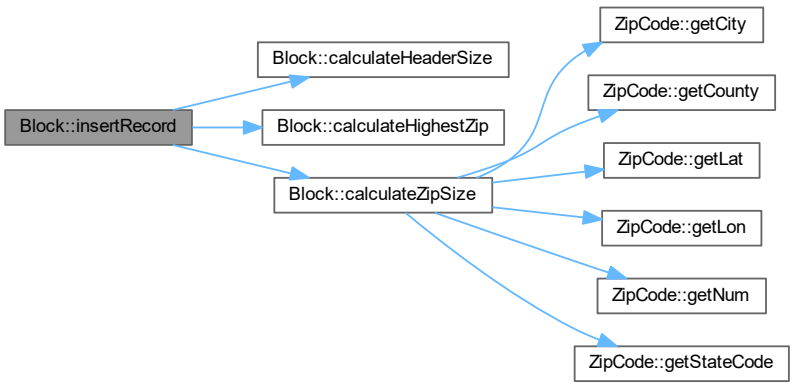
Adds a ZipCode record to the Block. Returns true if successful, false otherwise.

Returns

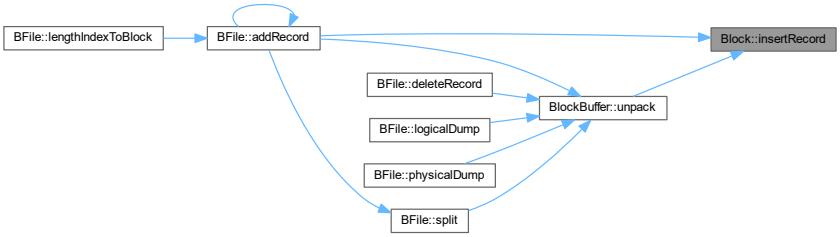
Boolean indicating whether the record was successfully added.

Definition at line 77 of file Block.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



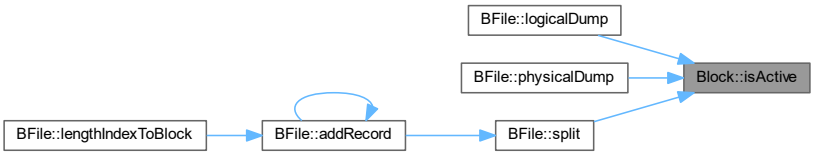
3.2.3.12 isActive()

```
bool Block::isActive ( ) const  [inline]
```

Check if the block is active.

Definition at line 82 of file Block.h.

Here is the caller graph for this function:



3.2.3.13 removeRecord()

```
bool Block::removeRecord (
            int zip )
```

Removes a ZipCode record from the Block.

Precondition

Requires a valid zip code to remove from the Block.

Postcondition

Removes a record from the Block, returns true if successful, false otherwise.

Parameters

zip	The zip code number of the record to remove.

Precondition

The Block must contain the ZipCode record to be removed.

Postcondition

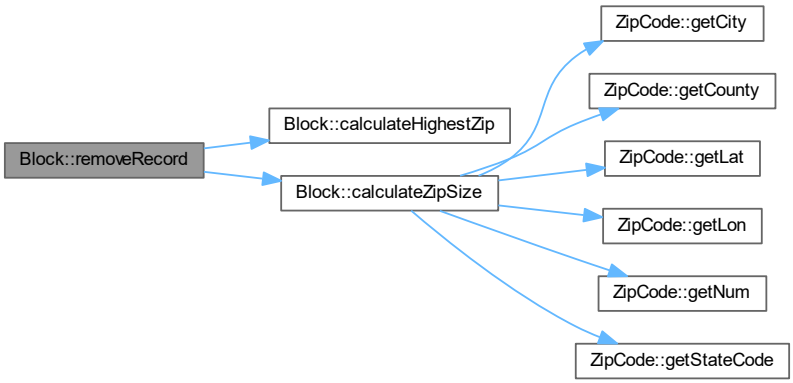
Removes the specified ZipCode record from the Block. Decrements the record count.

Returns

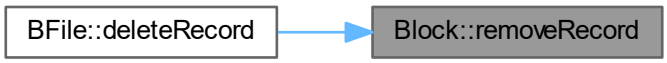
Boolean indicating whether the record was successfully removed.

Definition at line 124 of file Block.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.2.3.14 searchZip()

```
bool Block::searchZip (
            ZipCode & resultZip,
            int target )
```

Searches for a specific ZipCode in the block based on the given target number.

Parameters

resultZip	A reference to a ZipCode object to store the found record.
target	The target zip code number to search for.

Returns

Boolean indicating whether the zip code was found.

Postcondition

If found, resultZip is set to the matching ZipCode record.

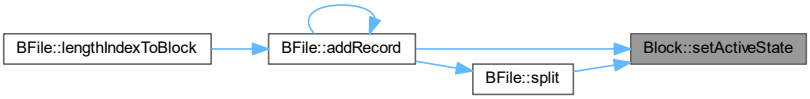
Definition at line 192 of file Block.cpp.

3.2.3.15 setActiveState()

```
void Block::setActiveState (
            bool state )  [inline]
```

Definition at line 104 of file Block.h.

Here is the caller graph for this function:

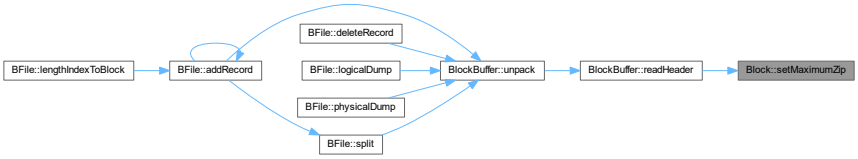


3.2.3.16 setMaximumZip()

```
void Block::setMaximumZip (
            int highestZip )  [inline]
```

Definition at line 109 of file Block.h.

Here is the caller graph for this function:

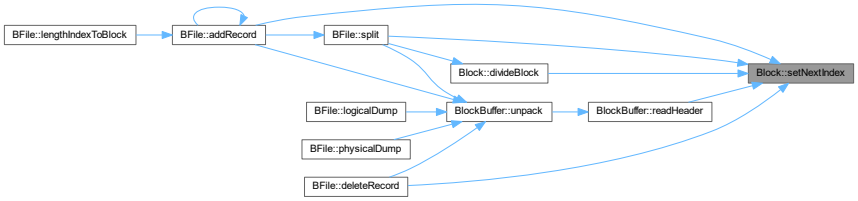


3.2.3.17 setNextIndex()

```
void Block::setNextIndex (
            int next )  [inline]
```

Definition at line 105 of file Block.h.

Here is the caller graph for this function:

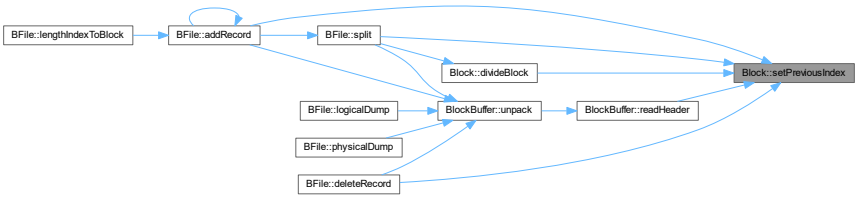


3.2.3.18 setPreviousIndex()

```
void Block::setPreviousIndex (
            int prev )  [inline]
```

Definition at line 106 of file Block.h.

Here is the caller graph for this function:

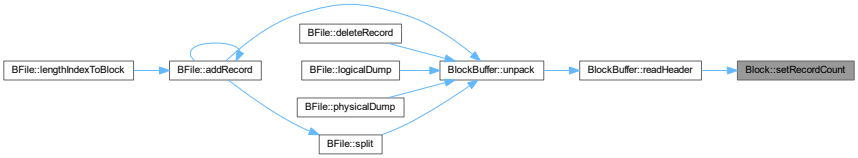


3.2.3.19 setRecordCount()

```
void Block::setRecordCount (
            int recCount )  [inline]
```

Definition at line 107 of file Block.h.

Here is the caller graph for this function:

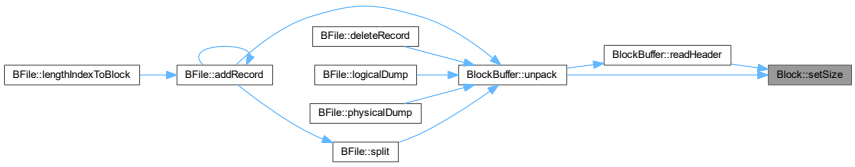


3.2.3.20 setSize()

```
void Block::setSize (
            int currentSize )  [inline]
```

Definition at line 108 of file Block.h.

Here is the caller graph for this function:



3.2.4 Member Data Documentation

3.2.4.1 active

```
bool Block::active  [private]
```

Definition at line 122 of file Block.h.

3.2.4.2 currentSize

```
int Block::currentSize  [private]
```

Definition at line 124 of file Block.h.

3.2.4.3 highestZip

```
int Block::highestZip  [private]
```

Definition at line 124 of file Block.h.

3.2.4.4 next

```
int Block::next  [private]
```

Definition at line 123 of file Block.h.

3.2.4.5 prev

```
int Block::prev  [private]
```

Definition at line 123 of file Block.h.

3.2.4.6 recCount

```
int Block::recCount  [private]
```

Definition at line 124 of file Block.h.

3.2.4.7 records

```
vector<ZipCode> Block::records  [private]
```

Definition at line 125 of file Block.h.

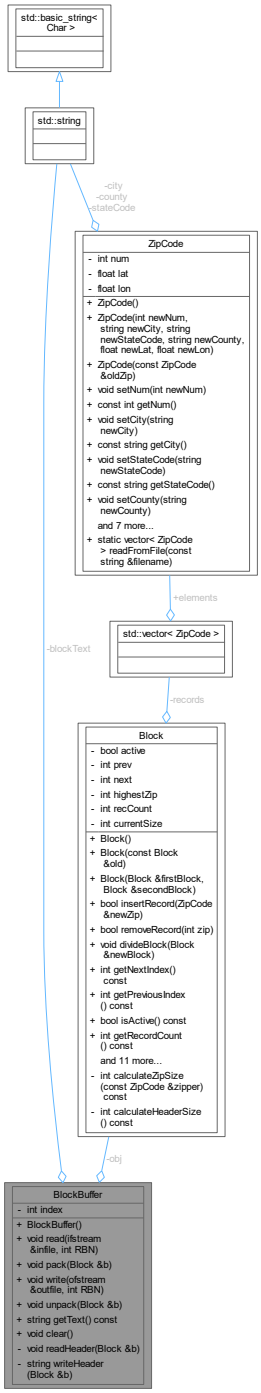
The documentation for this class was generated from the following files:

- Block.h
- Block.cpp

3.3 BlockBuffer Class Reference

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

Collaboration diagram for BlockBuffer:



Public Member Functions

- BlockBuffer ()

  Constructs a BlockBuffer with an empty text buffer.
- void read (ifstream &infile, int RBN)

  Reads a block from a file based on its relative block number.
- void pack (Block &b)

Converts a Block object into a text representation.

- void write (ofstream &outfile, int RBN)

  Writes the content of blockText to a file at a specific block position.

- void unpack (Block &b)

  Parses the blockText into a Block object.

- string getText () const

  Retrieves the content of the blockText buffer.

- void clear ()

  Clears the contents of the BlockBuffer.

Private Member Functions

- void readHeader (Block &b)

  Reads and parses the header data from a Block object.

- string writeHeader (Block &b)

  Generates a string representation of a Block object's header data.

Private Attributes

- string blockText
- Block obj
- int index

3.3.1 Detailed Description

Definition at line 18 of file BlockBuffer.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 BlockBuffer()

```
BlockBuffer::BlockBuffer ( )  [inline]
```

Constructs a BlockBuffer with an empty text buffer.

Definition at line 23 of file BlockBuffer.h.

3.3.3 Member Function Documentation

3.3.3.1 clear()

```
void BlockBuffer::clear ( )  [inline]
```

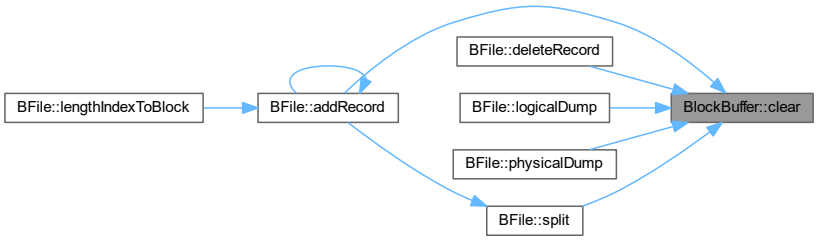
Clears the contents of the BlockBuffer.

Postcondition

blockText is cleared and the index is reset.

Definition at line 65 of file BlockBuffer.h.

Here is the caller graph for this function:



3.3.3.2 getText()

```
string BlockBuffer::getText ( ) const  [inline]
```

Retrieves the content of the blockText buffer.

Returns

A string containing the content of blockText.

Definition at line 59 of file BlockBuffer.h.

3.3.3.3 pack()

```
void BlockBuffer::pack (
            Block & b )
```

Converts a Block object into a text representation.

Parameters

b	The Block object to be converted into text.

Postcondition

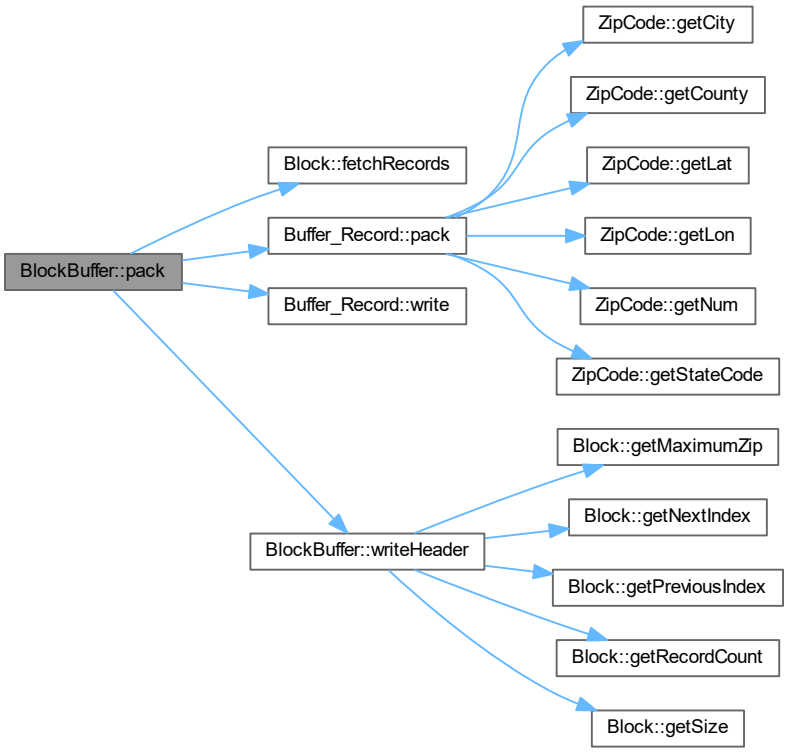
The Block object's data is converted into a string and stored in blockText.

Parameters

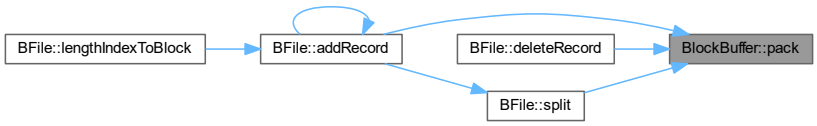
b	The Block object to be converted into text.

Definition at line 30 of file BlockBuffer.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.3.4 read()

```
void BlockBuffer::read (
            ifstream & inFile,
            int RBN )
```

Reads a block from a file based on its relative block number.

Parameters

infile	The input file stream to read from.
RBN	The relative block number indicating the specific block in the file.

Postcondition

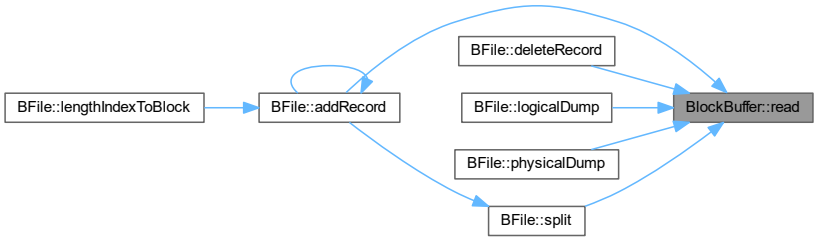
The content of the specified block is loaded into blockText.

Parameters

inFile	The input file stream to read from.
RBN	The relative block number indicating the specific block in the file.

Definition at line 14 of file BlockBuffer.cpp.

Here is the caller graph for this function:



3.3.3.5 readHeader()

```
void BlockBuffer::readHeader (
            Block & b )  [private]
```

Reads and parses the header data from a Block object.

Parameters

b	The Block object from which the header data will be read.

Postcondition

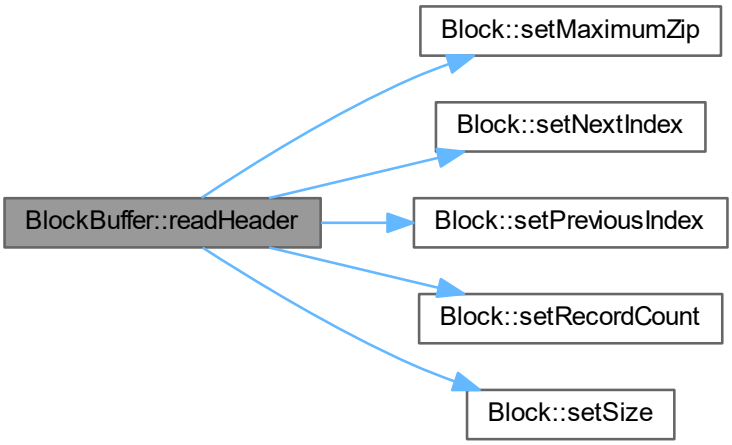
The header data is extracted and stored into the provided Block object.

Parameters

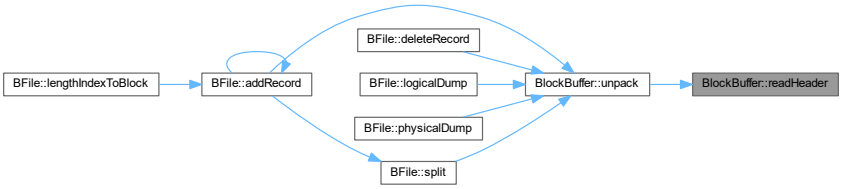
b	The Block object from which the header data will be read.

Definition at line 112 of file BlockBuffer.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.3.6 unpack()

```
void BlockBuffer::unpack (
            Block & b )
```

Parses the blockText into a Block object.

Parameters

b	An empty Block object that will be filled with data from blockText.

Postcondition

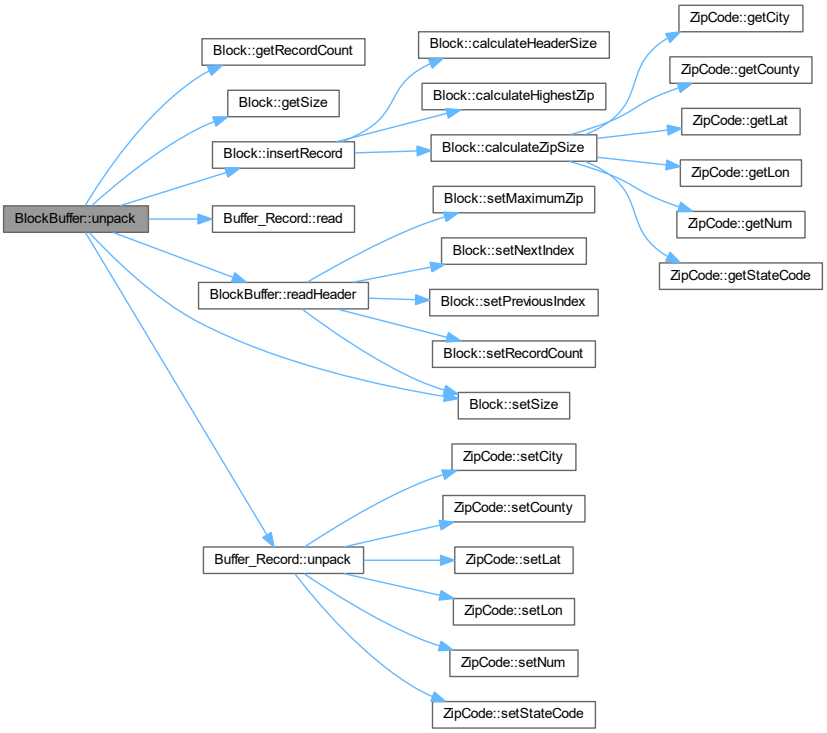
The blockText is parsed and its data is stored into the provided Block object.

Parameters

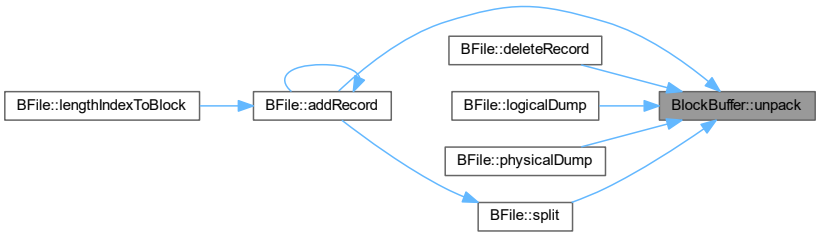
b	An empty Block object that will be filled with data from blockText.

Definition at line 65 of file BlockBuffer.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.3.7 write()

```
void BlockBuffer::write (
            ofstream & outfile,
            int RBN )
```

Writes the content of blockText to a file at a specific block position.

Parameters

outfile	The output file stream where the blockText will be written.
RBN	The relative block number indicating the position in the file to write.

Postcondition

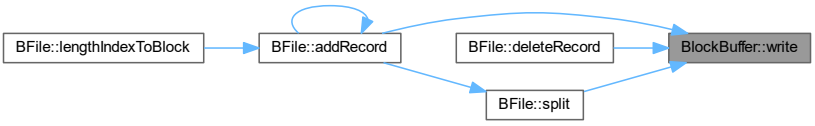
The content of blockText is written to the file at the specified block position.

Parameters

outfile	The output file stream where the blockText will be written.
RBN	The relative block number indicating the position in the file to write.

Definition at line 50 of file BlockBuffer.cpp.

Here is the caller graph for this function:



3.3.3.8 writeHeader()

```
string BlockBuffer::writeHeader (
            Block & b )  [private]
```

Generates a string representation of a Block object's header data.

Parameters

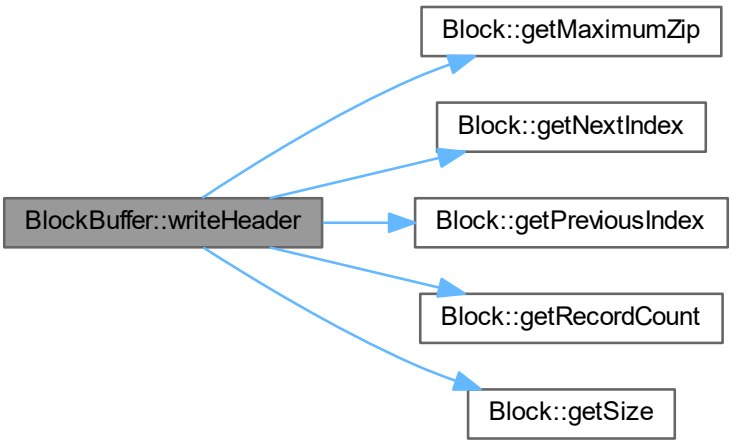
b	The Block object whose header data will be written to a string.

Returns

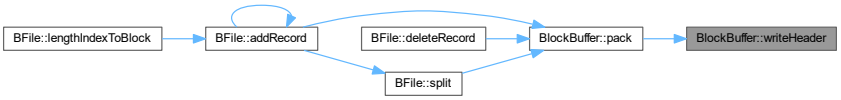
A string containing the header data of the Block object.

Definition at line 158 of file BlockBuffer.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.4 Member Data Documentation

3.3.4.1 blockText

```
string BlockBuffer::blockText  [private]
```

Definition at line 82 of file BlockBuffer.h.

3.3.4.2 index

```
int BlockBuffer::index  [private]
```

Definition at line 84 of file BlockBuffer.h.

3.3.4.3 obj

```
Block BlockBuffer::obj  [private]
```

Definition at line 83 of file BlockBuffer.h.

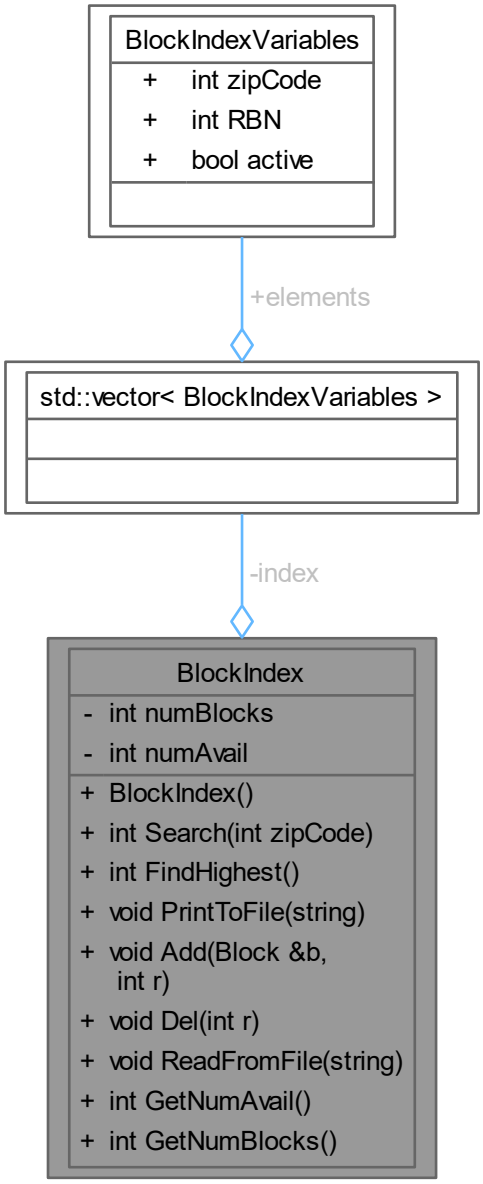
The documentation for this class was generated from the following files:

- BlockBuffer.h
- BlockBuffer.cpp

3.4 BlockIndex Class Reference

```
#include <BlockIndex.h>
```

Collaboration diagram for BlockIndex:



Public Member Functions

- BlockIndex ()
- int Search (int zipCode)
- int FindHighest ()
- void PrintToFile (string)
- void Add (Block &b, int r)
- void Del (int r)
- void ReadFromFile (string)
- int GetNumAvail ()
- int GetNumBlocks ()

Private Attributes

- int numBlocks
- int numAvail
- vector< BlockIndexVariables > index

3.4.1 Detailed Description

Definition at line 23 of file BlockIndex.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 BlockIndex()

```
BlockIndex::BlockIndex ( )  [inline]
```

Definition at line 35 of file BlockIndex.h.

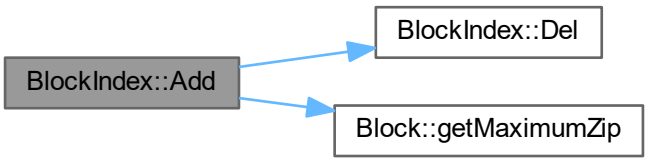
3.4.3 Member Function Documentation

3.4.3.1 Add()

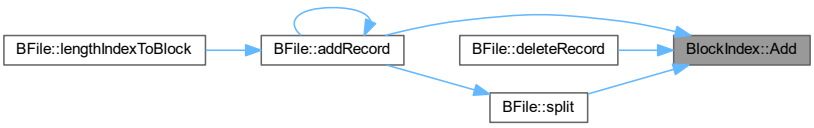
```
void BlockIndex::Add (
            Block & b,
            int r )
```

Definition at line 54 of file BlockIndex.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

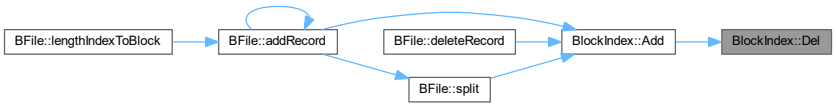


3.4.3.2 Del()

```
void BlockIndex::Del (
            int r )
```

Definition at line 63 of file BlockIndex.cpp.

Here is the caller graph for this function:



3.4.3.3 FindHighest()

```
int BlockIndex::FindHighest ( )
```

Definition at line 8 of file BlockIndex.cpp.

Here is the caller graph for this function:

3.4.3.4 GetNumAvail()

```
int BlockIndex::GetNumAvail ( )  [inline]
```

Definition at line 77 of file BlockIndex.h.

3.4.3.5 GetNumBlocks()

```
int BlockIndex::GetNumBlocks ( )  [inline]
```

Definition at line 85 of file BlockIndex.h.

3.4.3.6 PrintToFile()

```
void BlockIndex::PrintToFile (
            string out )
```

Definition at line 95 of file BlockIndex.cpp.

3.4.3.7 ReadFromFile()

```
void BlockIndex::ReadFromFile (
            string in )
```

Postcondition

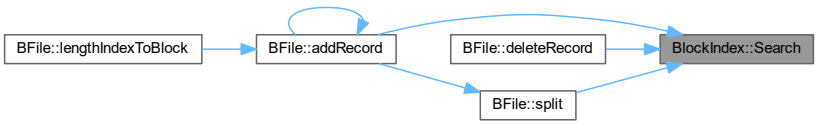
Definition at line 75 of file BlockIndex.cpp.

3.4.3.8 Search()

```
int BlockIndex::Search (
            int zipCode )
```

Definition at line 28 of file BlockIndex.cpp.

Here is the caller graph for this function:



3.4.4 Member Data Documentation

3.4.4.1 index

```
vector<BlockIndexVariables> BlockIndex::index  [private]
```

Definition at line 27 of file BlockIndex.h.

3.4.4.2 numAvail

```
int BlockIndex::numAvail  [private]
```

Definition at line 26 of file BlockIndex.h.

3.4.4.3 numBlocks

```
int BlockIndex::numBlocks  [private]
```

Definition at line 26 of file BlockIndex.h.

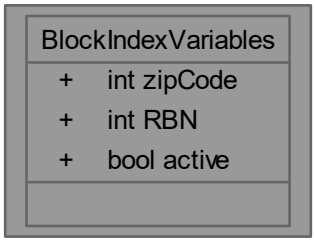
The documentation for this class was generated from the following files:

- BlockIndex.h
- BlockIndex.cpp

3.5 BlockIndexVariables Struct Reference

```
#include <BlockIndex.h>
```

Collaboration diagram for BlockIndexVariables:



Public Attributes

- int zipCode
- int RBN
- bool active

3.5.1 Detailed Description

Definition at line 17 of file BlockIndex.h.

3.5.2 Member Data Documentation

3.5.2.1 active

```
bool BlockIndexVariables::active
```

Definition at line 20 of file BlockIndex.h.

3.5.2.2 RBN

```
int BlockIndexVariables::RBN
```

Definition at line 19 of file BlockIndex.h.

3.5.2.3 zipCode

```
int BlockIndexVariables::zipCode
```

Definition at line 18 of file BlockIndex.h.

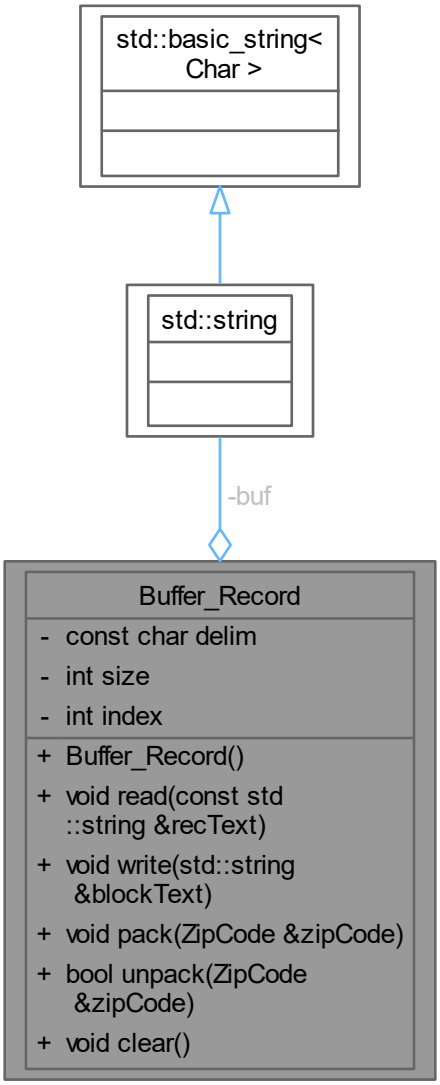
The documentation for this struct was generated from the following file:

- BlockIndex.h

3.6 Buffer_Record Class Reference

```
#include <Buffer_Record.h>
```

Collaboration diagram for Buffer_Record:



Public Member Functions

- Buffer_Record ()

  Default constructor.
- void read (const std::string &recText)

  Sets the buffer to a given string.
- void write (std::string &blockText)

  Writes the buffer to the blockText string.
- void pack (ZipCode &zipCode)

  Parses a ZipCode object onto the buffer.
- bool unpack (ZipCode &zipCode)

Parses the buffer and turns it into a ZipCode object.

- void clear ()

  Clears the buffer.

Private Attributes

- std::string buf
- const char delim = ','
- int size
- int index

3.6.1 Detailed Description

Definition at line 12 of file Buffer_Record.h.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Buffer_Record()

```
Buffer_Record::Buffer_Record ( )  [inline]
```

Default constructor.

Postcondition

Initializes a new Buffer_Record object.

Definition at line 19 of file Buffer_Record.h.

3.6.3 Member Function Documentation

3.6.3.1 clear()

```
void Buffer_Record::clear ( )  [inline]
```

Clears the buffer.

Clears the contents of the buffer.

Precondition

None.

Postcondition

Buffer is empty.

The buffer is cleared and reset.

Definition at line 64 of file Buffer_Record.h.

3.6.3.2 pack()

```
void Buffer_Record::pack (
            ZipCode & z )
```

Parses a ZipCode object onto the buffer.

Default constructor for Buffer_Record.

Parameters

zipCode	The ZipCode object to be packed into the buffer.

Precondition

Requires a ZipCode object.

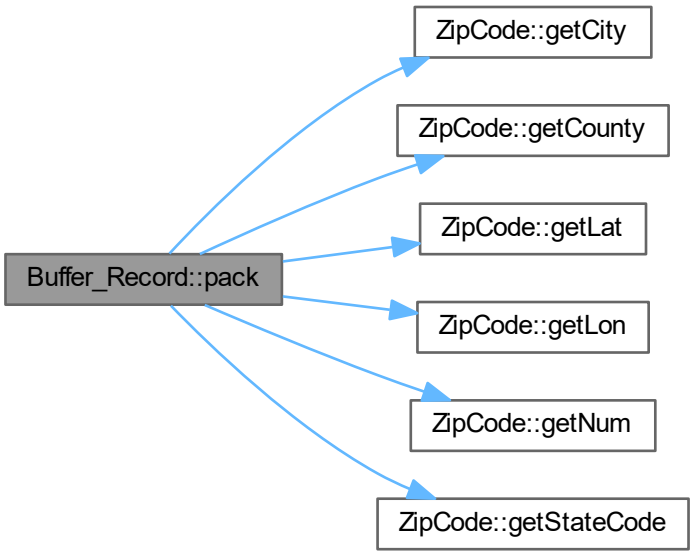
Postcondition

The buffer is filled with a record.

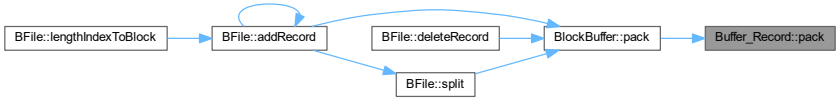
Initializes a new Buffer_Record object with an empty buffer.

Definition at line 16 of file Buffer_Record.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.6.3.3 read()

```
void Buffer_Record::read (
            const std::string & recText )
```

Sets the buffer to a given string.

Reads a string into the buffer.

Parameters

recText	The record text to be read into the buffer.

Precondition

Takes a string given by the BlockBuffer class.

Postcondition

Adds the string to the buffer.

Parameters

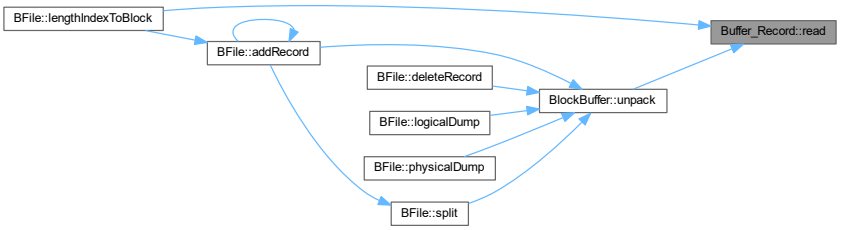
recText	The record text to be read into the buffer.

Postcondition

The buffer is set to the contents of recText.

Definition at line 41 of file Buffer_Record.cpp.

Here is the caller graph for this function:



3.6.3.4 unpack()

```
bool Buffer_Record::unpack (
            ZipCode & z )
```

Parses the buffer and turns it into a ZipCode object.

Unpacks the buffer into a ZipCode object.

Parameters

zipCode	An empty ZipCode object to be filled with unpacked data.

Returns

True if the buffer was successfully unpacked into a ZipCode object, false otherwise.

Precondition

Receives an empty ZipCode object.

Postcondition

Returns true if the buffer was made into a ZipCode object or false if it wasn't.

Parameters

z	A reference to a ZipCode object to be filled with unpacked data.

Returns

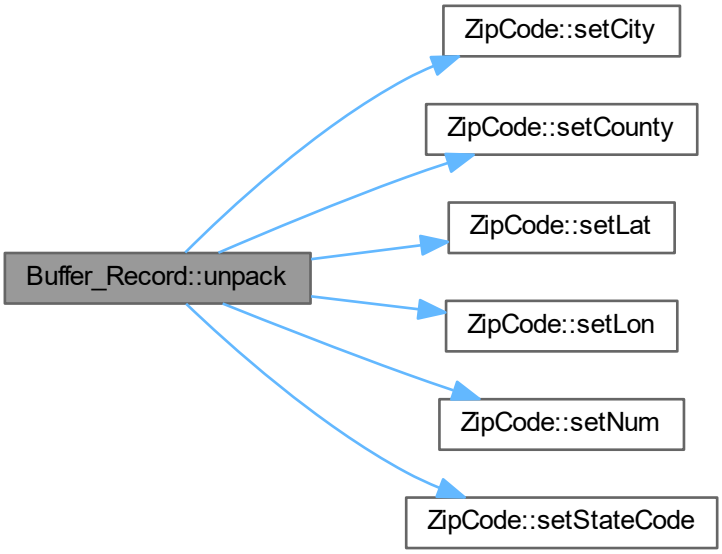
True if unpacking is successful, false otherwise.

Postcondition

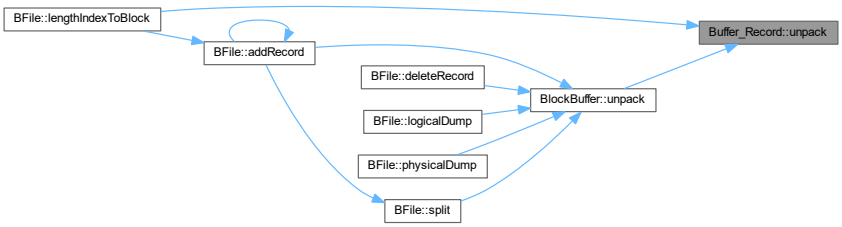
If successful, z is filled with the unpacked data from the buffer.

Definition at line 63 of file Buffer_Record.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.6.3.5 write()

```
void Buffer_Record::write (
            std::string & blockText )
```

Writes the buffer to the blockText string.

Writes the buffer content into a given string.

Parameters

blockText	The string where the buffer content will be written.

Precondition

Takes a string given by the BlockBuffer class to be written to.

Postcondition

Clears the buffer.

Parameters

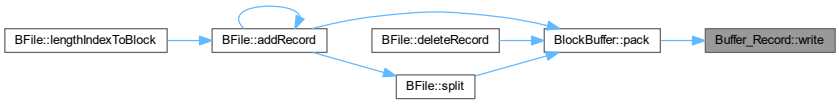
blockText	A reference to the string where the buffer content will be written.

Postcondition

The buffer's content is appended to blockText, and the buffer is cleared.

Definition at line 51 of file Buffer_Record.cpp.

Here is the caller graph for this function:



3.6.4 Member Data Documentation

3.6.4.1 buf

```
std::string Buffer_Record::buf  [private]
```

Definition at line 71 of file Buffer_Record.h.

3.6.4.2 delim

```
const char Buffer_Record::delim = ','  [private]
```

Definition at line 72 of file Buffer_Record.h.

3.6.4.3 index

```
int Buffer_Record::index  [private]
```

Definition at line 74 of file Buffer_Record.h.

3.6.4.4 size

```
int Buffer_Record::size  [private]
```

Definition at line 73 of file Buffer_Record.h.

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

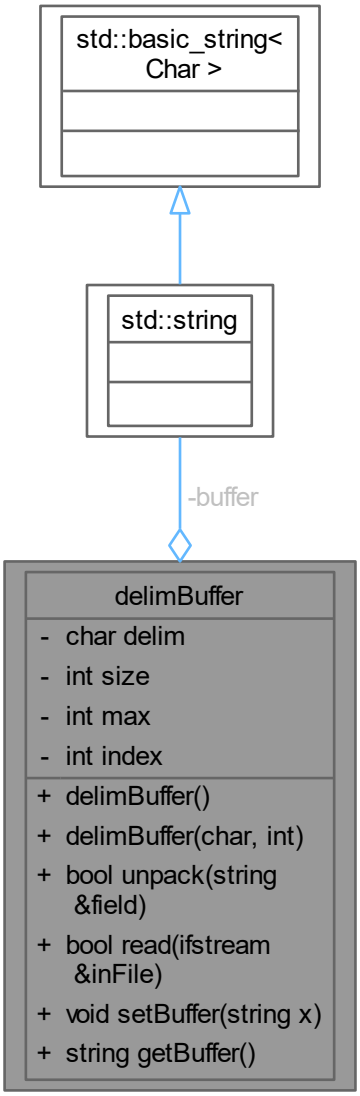
- Buffer_Record.h
- Buffer_Record.cpp

3.7 delimBuffer Class Reference

class to store each record and parse each field

```
#include <delimBuffer.h>
```

Collaboration diagram for delimBuffer:



Public Member Functions

- delimBuffer ()

  Constructor for the delimBuffer class.
- delimBuffer (char, int)
- bool unpack (string &field)

  Seperates each field from the line on the delimBuffer.
- bool read (ifstream &inFile)

  reads from csv file and places on string
- void setBuffer (string x)

  Gives the delimBuffer string
- string getBuffer ()

Private Attributes

- char delim
- int size
- int max
- int index
- string buffer

3.7.1 Detailed Description

class to store each record and parse each field

Definition at line 18 of file delimBuffer.h.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 delimBuffer() [1/2]

```
delimBuffer::delimBuffer ( )
```

Constructor for the delimBuffer class.

Precondition

Takes in the address to the us_postal_codes.csv file

Postcondition

inFile, index and buf are all initialized

delimBuffer.CPP Member function definitions for the delimBuffer class.

Definition at line 12 of file delimBuffer.cpp.

3.7.2.2 delimBuffer() [2/2]

```
delimBuffer::delimBuffer (
            char delim = ',',
            int maxsize = 1000 )
```

Definition at line 26 of file delimBuffer.cpp.

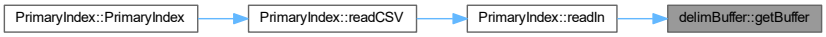
3.7.3 Member Function Documentation

3.7.3.1 getBuffer()

```
string delimBuffer::getBuffer ( )  [inline]
```

Definition at line 58 of file delimBuffer.h.

Here is the caller graph for this function:



3.7.3.2 read()

```
bool delimBuffer::read (
            ifstream & inFile )
```

reads from csv file and places on string

Postcondition

returns the string of one line of us_postal_codes.csv

Definition at line 38 of file delimBuffer.cpp.

Here is the caller graph for this function:

3.7.3.3 setBuffer()

```
void delimBuffer::setBuffer (
            string x )  [inline]
```

Gives the delimBuffer string

Postcondition

Returns the delimBuffer string

Definition at line 55 of file delimBuffer.h.

3.7.3.4 unpack()

```
bool delimBuffer::unpack (
            string & field )
```

Seperates each field from the line on the delimBuffer.

Precondition

delimBuffer must not be empty

Postcondition

Makes parameter string equal to correct field in record

Definition at line 52 of file delimBuffer.cpp.

Here is the caller graph for this function:

3.7.4 Member Data Documentation

3.7.4.1 buffer

```
string delimBuffer::buffer  [private]
```

Definition at line 25 of file delimBuffer.h.

3.7.4.2 delim

```
char delimBuffer::delim  [private]
```

Definition at line 21 of file delimBuffer.h.

3.7.4.3 index

```
int delimBuffer::index  [private]
```

Definition at line 24 of file delimBuffer.h.

3.7.4.4 max

```
int delimBuffer::max  [private]
```

Definition at line 23 of file delimBuffer.h.

3.7.4.5 size

```
int delimBuffer::size  [private]
```

Definition at line 22 of file delimBuffer.h.

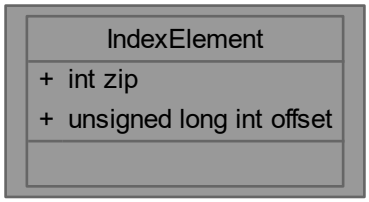
The documentation for this class was generated from the following files:

- delimBuffer.h
- delimBuffer.cpp

3.8 IndexElement Struct Reference

```
#include <PrimaryIndex.h>
```

Collaboration diagram for IndexElement:



Public Attributes

- int zip
- unsigned long int offset

3.8.1 Detailed Description

PrimaryIndex.h Class containing the primary index and the byte offset of the data file for the corresponding primary key.

Definition at line 19 of file PrimaryIndex.h.

3.8.2 Member Data Documentation

3.8.2.1 offset

```
unsigned long int IndexElement::offset
```

Definition at line 22 of file PrimaryIndex.h.

3.8.2.2 zip

```
int IndexElement::zip
```

Definition at line 21 of file PrimaryIndex.h.

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

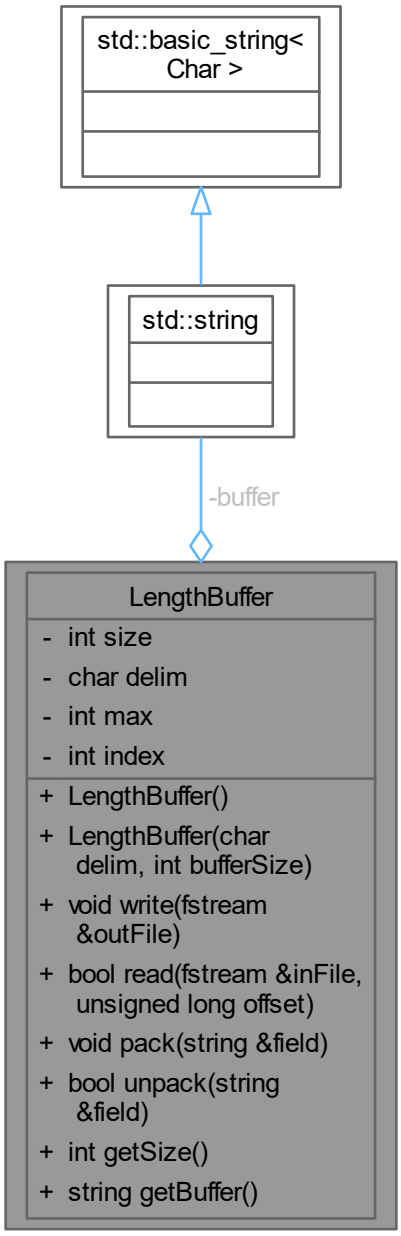
- PrimaryIndex.h

3.9 LengthBuffer Class Reference

Class to store each record and parse each field.

```
#include <LengthBuffer.h>
```

Collaboration diagram for LengthBuffer:



Public Member Functions

- LengthBuffer ()

  Constructor for the LengthBuffer class.
- LengthBuffer (char delim, int bufferSize)
- void write (fstream &outFile)
- bool read (fstream &inFile, unsigned long offset)

  Reads from the CSV file and places it in a string.

- void pack (string &field)
- bool unpack (string &field)

  Separates each field from the line in the LengthBuffer.
- int getSize ()
- string getBuffer ()

  Gives the LengthBuffer string.

Private Attributes

- int size
- char delim
- int max
- int index
- string buffer

3.9.1 Detailed Description

Class to store each record and parse each field.

Definition at line 18 of file LengthBuffer.h.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 LengthBuffer() [1/2]

```
LengthBuffer::LengthBuffer ( )
```

Constructor for the LengthBuffer class.

Precondition

Takes in the address of the us_postal_codes.csv file.

Postcondition

inFile, index, and buf are all initialized.

Definition at line 14 of file LengthBuffer.cpp.

3.9.2.2 LengthBuffer() [2/2]

```
LengthBuffer::LengthBuffer (
            char delim,
            int bufferSize )
```

Definition at line 28 of file LengthBuffer.cpp.

3.9.3 Member Function Documentation

3.9.3.1 getBuffer()

```
string LengthBuffer::getBuffer ( )  [inline]
```

Gives the LengthBuffer string.

Postcondition

Returns the LengthBuffer string.

Definition at line 62 of file LengthBuffer.h.

Here is the caller graph for this function:

3.9.3.2 getSize()

```
int LengthBuffer::getSize ( )  [inline]
```

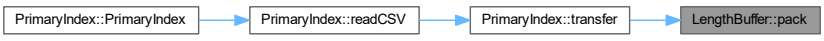
Definition at line 55 of file LengthBuffer.h.

3.9.3.3 pack()

```
void LengthBuffer::pack (
            string & field )
```

Definition at line 104 of file LengthBuffer.cpp.

Here is the caller graph for this function:



3.9.3.4 read()

```
bool LengthBuffer::read (
            fstream & inFile,
            unsigned long offset )
```

Reads from the CSV file and places it in a string.

Postcondition

Returns the string of one line from us_postal_codes.csv.

Definition at line 55 of file LengthBuffer.cpp.

Here is the caller graph for this function:

3.9.3.5 unpack()

```
bool LengthBuffer::unpack (
            string & field )
```

Separates each field from the line in the LengthBuffer.

Precondition

LengthBuffer must not be empty.

Postcondition

Makes the parameter string equal to the correct field in the record.

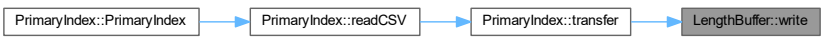
Definition at line 90 of file LengthBuffer.cpp.

3.9.3.6 write()

```
void LengthBuffer::write (
            fstream & outFile )
```

Definition at line 43 of file LengthBuffer.cpp.

Here is the caller graph for this function:



3.9.4 Member Data Documentation

3.9.4.1 buffer

```
string LengthBuffer::buffer  [private]
```

Definition at line 25 of file LengthBuffer.h.

3.9.4.2 delim

```
char LengthBuffer::delim  [private]
```

Definition at line 22 of file LengthBuffer.h.

3.9.4.3 index

```
int LengthBuffer::index  [private]
```

Definition at line 24 of file LengthBuffer.h.

3.9.4.4 max

```
int LengthBuffer::max  [private]
```

Definition at line 23 of file LengthBuffer.h.

3.9.4.5 size

```
int LengthBuffer::size  [private]
```

Definition at line 21 of file LengthBuffer.h.

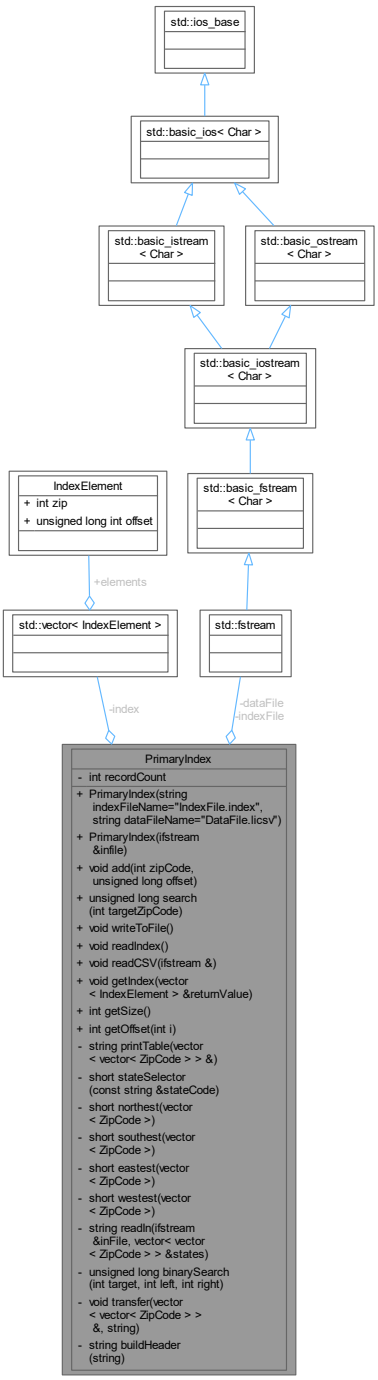
The documentation for this class was generated from the following files:

- LengthBuffer.h
- LengthBuffer.cpp

3.10 PrimaryIndex Class Reference

`#include <PrimaryIndex.h>`

Collaboration diagram for PrimaryIndex:



Public Member Functions

- PrimaryIndex (string indexFileName="IndexFile.index", string dataFileName="DataFile.licsv")

- PrimaryIndex (ifstream &infile)
- void add (int zipCode, unsigned long offset)
- unsigned long search (int targetZipCode)
- void writeToFile ()
- void readIndex ()
- void readCSV (ifstream &)
- void getIndex (vector< IndexElement > &returnValue)
- int getSize ()
- int getOffset (int i)

Private Member Functions

- string printTable (vector< vector< ZipCode > > &)
- short stateSelector (const string &stateCode)

  Chooses which state array index is correct with the use of a switch statement.
- short northest (vector< ZipCode >)
- short southest (vector< ZipCode >)
- short eastest (vector< ZipCode >)
- short westest (vector< ZipCode >)
- string readIn (ifstream &inFile, vector< vector< ZipCode > > &states)
- unsigned long binarySearch (int target, int left, int right)
- void transfer (vector< vector< ZipCode > > &, string)
- string buildHeader (string)

Private Attributes

- vector< IndexElement > index
- int recordCount
- fstream dataFile
- fstream indexFile

3.10.1 Detailed Description

Definition at line 25 of file PrimaryIndex.h.

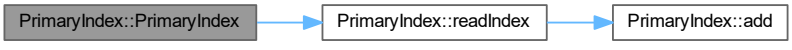
3.10.2 Constructor & Destructor Documentation

3.10.2.1 PrimaryIndex() [1/2]

```
PrimaryIndex::PrimaryIndex (
            string indexFileName = "IndexFile.index",
            string dataFileName = "DataFile.licsv" )  [inline]
```

Definition at line 55 of file PrimaryIndex.h.

Here is the call graph for this function:

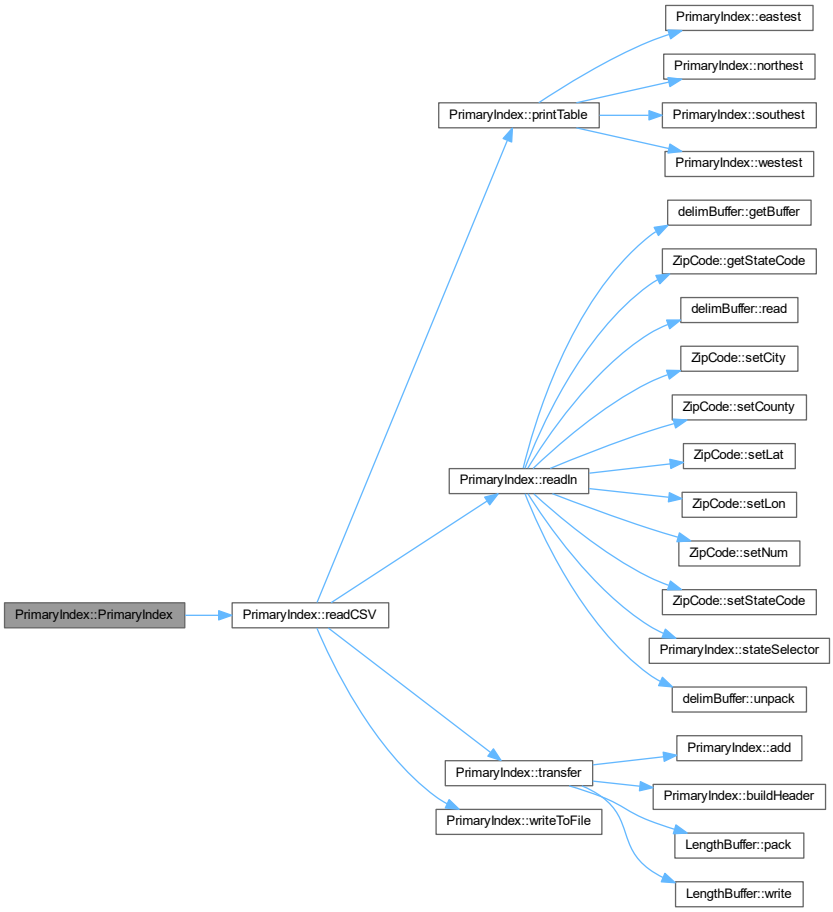


3.10.2.2 PrimaryIndex() [2/2]

```
PrimaryIndex::PrimaryIndex (
            ifstream & infile )  [inline]
```

Definition at line 58 of file PrimaryIndex.h.

Here is the call graph for this function:



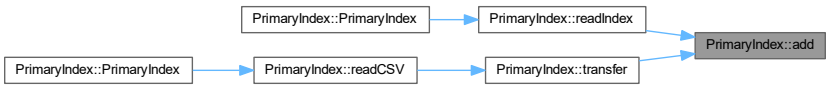
3.10.3 Member Function Documentation

3.10.3.1 add()

```
void PrimaryIndex::add (
            int zipCode,
            unsigned long offset )
```

Definition at line 25 of file PrimaryIndex.cpp.

Here is the caller graph for this function:



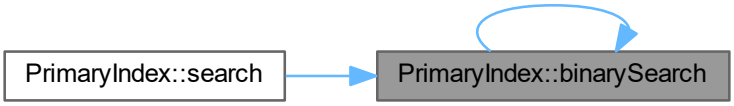
3.10.3.2 binarySearch()

```
unsigned long PrimaryIndex::binarySearch (
            int target,
            int left,
            int right )  [private]
```

Definition at line 54 of file PrimaryIndex.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:



3.10.3.3 buildHeader()

```
string PrimaryIndex::buildHeader (
            string headerData )  [private]
```

Definition at line 115 of file PrimaryIndex.cpp.

Here is the caller graph for this function:

3.10.3.4 eastest()

```
short PrimaryIndex::eastest (
            vector< ZipCode > state )  [private]
```

Definition at line 306 of file PrimaryIndex.cpp.

Here is the caller graph for this function:

3.10.3.5 getIndex()

```
void PrimaryIndex::getIndex (
            vector< IndexElement > & returnValue )
```

Definition at line 12 of file PrimaryIndex.cpp.

Here is the caller graph for this function:

3.10.3.6 getOffset()

```
int PrimaryIndex::getOffset (
            int i )  [inline]
```

Definition at line 76 of file PrimaryIndex.h.

3.10.3.7 getSize()

```
int PrimaryIndex::getSize ( )  [inline]
```

Definition at line 73 of file PrimaryIndex.h.

3.10.3.8 northest()

```
short PrimaryIndex::northest (
            vector< ZipCode > state )  [private]
```

Definition at line 286 of file PrimaryIndex.cpp.

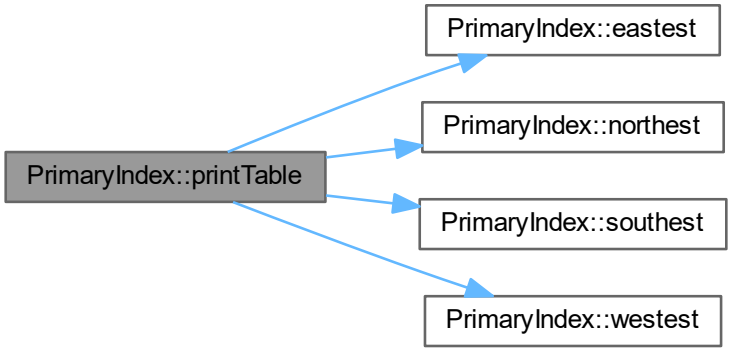
Here is the caller graph for this function:

3.10.3.9 printTable()

```
string PrimaryIndex::printTable (
            vector< vector< ZipCode > > & states )  [private]
```

Definition at line 259 of file PrimaryIndex.cpp.

Here is the call graph for this function:



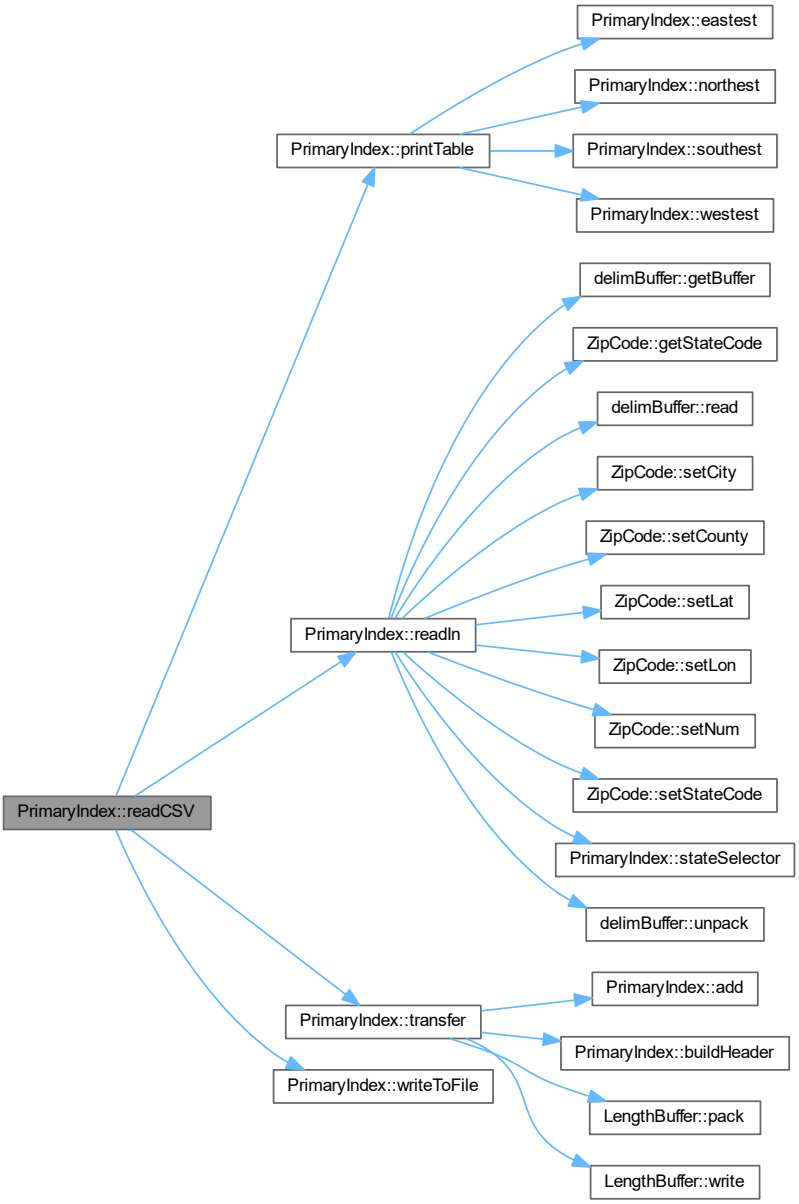
Here is the caller graph for this function:

3.10.3.10 readCSV()

```
void PrimaryIndex::readCSV (
            ifstream & infile )
```

Definition at line 101 of file PrimaryIndex.cpp.

Here is the call graph for this function:



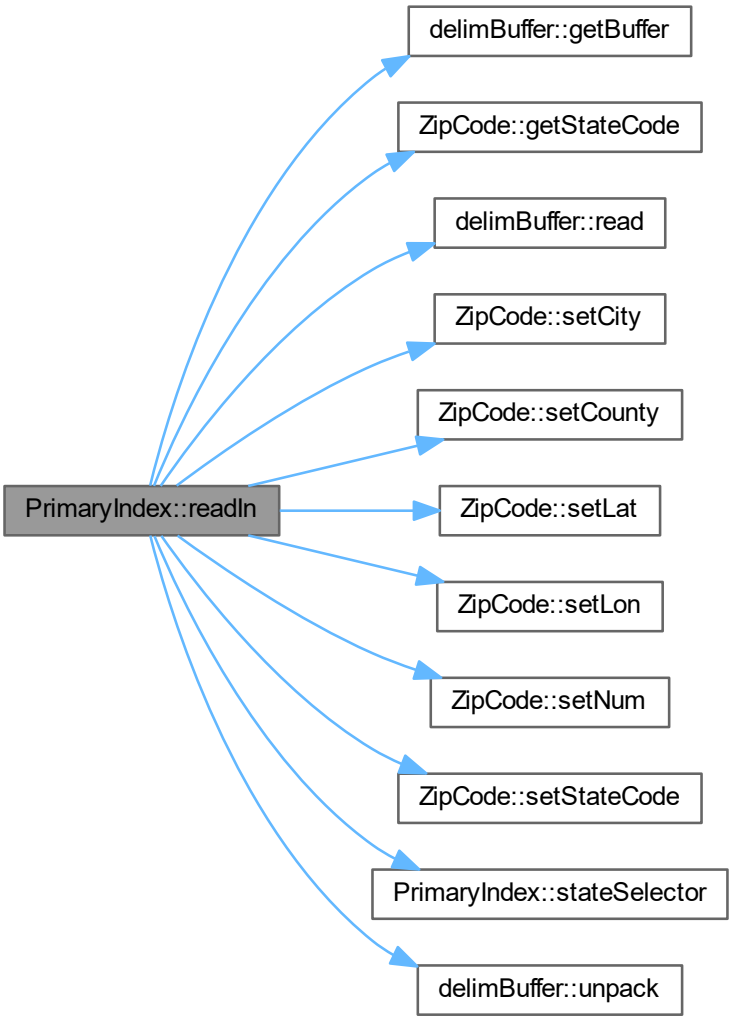
Here is the caller graph for this function:

3.10.3.11 readIn()

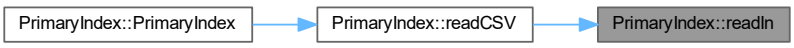
```
string PrimaryIndex::readIn (
            ifstream & inFile,
            vector< vector< ZipCode > > & states )  [private]
```

Definition at line 203 of file PrimaryIndex.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.10.3.12 readIndex()

```
void PrimaryIndex::readIndex ( )
```

Definition at line 73 of file PrimaryIndex.cpp.

Here is the call graph for this function:

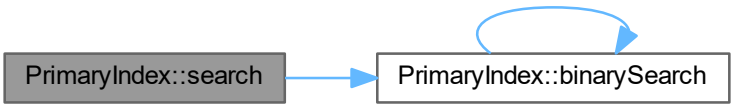
Here is the caller graph for this function:

3.10.3.13 search()

```
unsigned long PrimaryIndex::search (
            int targetZipCode )
```

Definition at line 49 of file PrimaryIndex.cpp.

Here is the call graph for this function:



3.10.3.14 southest()

```
short PrimaryIndex::southest (
            vector< ZipCode > state )  [private]
```

Definition at line 296 of file PrimaryIndex.cpp.

Here is the caller graph for this function:

3.10.3.15 stateSelector()

```
short PrimaryIndex::stateSelector (
            const string & stateCode )  [private]
```

Chooses which state array index is correct with the use of a switch statement.

Precondition

two character state code in a string is used as parameter

Postcondition

Returns the correct array index as an int

Definition at line 331 of file PrimaryIndex.cpp.

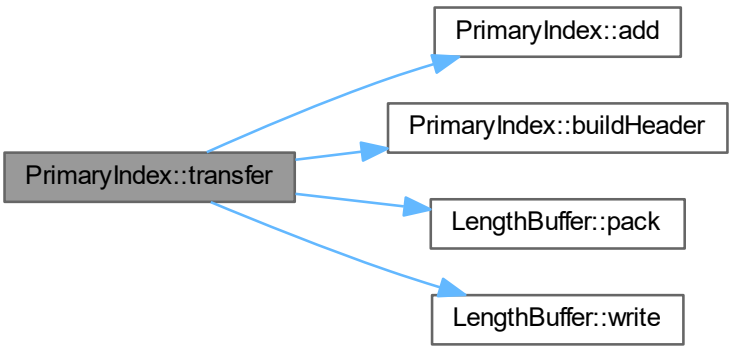
Here is the caller graph for this function:

3.10.3.16 transfer()

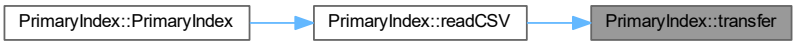
```
void PrimaryIndex::transfer (
            vector< vector< ZipCode > > & states,
            string headerData )  [private]
```

Definition at line 163 of file PrimaryIndex.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



3.10.3.17 westest()

```
short PrimaryIndex::westest (
            vector< ZipCode > state )  [private]
```

Definition at line 316 of file PrimaryIndex.cpp.

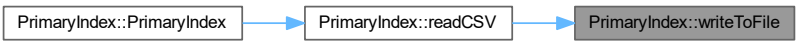
Here is the caller graph for this function:

3.10.3.18 writeToFile()

```
void PrimaryIndex::writeToFile ( )
```

Definition at line 90 of file PrimaryIndex.cpp.

Here is the caller graph for this function:



3.10.4 Member Data Documentation

3.10.4.1 dataFile

```
fstream PrimaryIndex::dataFile  [private]
```

Definition at line 51 of file PrimaryIndex.h.

3.10.4.2 index

```
vector<IndexElement> PrimaryIndex::index  [private]
```

Definition at line 49 of file PrimaryIndex.h.

3.10.4.3 indexFile

```
fstream PrimaryIndex::indexFile  [private]
```

Definition at line 51 of file PrimaryIndex.h.

3.10.4.4 recordCount

```
int PrimaryIndex::recordCount  [private]
```

Definition at line 50 of file PrimaryIndex.h.

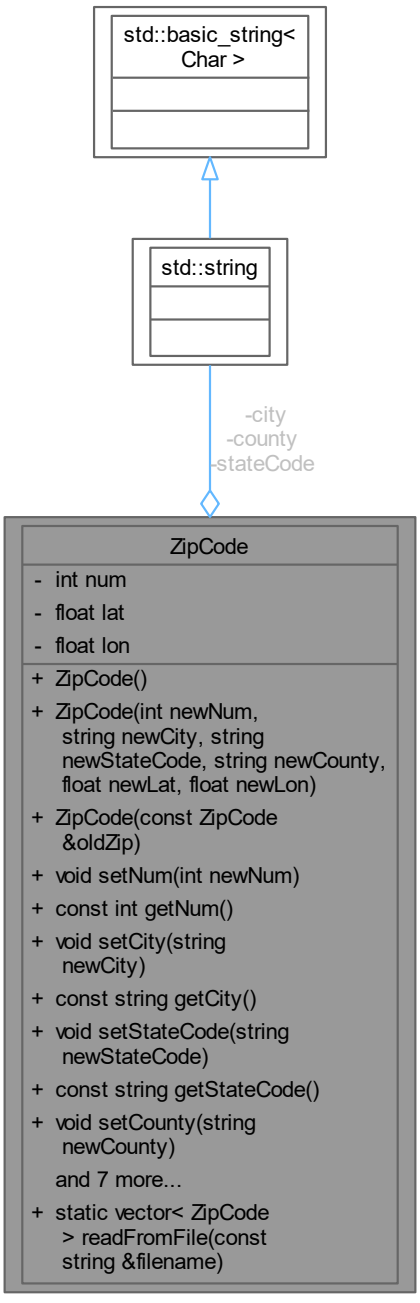
The documentation for this class was generated from the following files:

- PrimaryIndex.h
- PrimaryIndex.cpp

3.11 ZipCode Class Reference

`#include <zipCode.h>`

Collaboration diagram for ZipCode:



Public Member Functions

- ZipCode ()

- ZipCode (int newNum, string newCity, string newStateCode, string newCounty, float newLat, float newLon)
- ZipCode (const ZipCode &oldZip)
- void setNum (int newNum)
- const int getNum ()
- void setCity (string newCity)
- const string getCity ()
- void setStateCode (string newStateCode)
- const string getStateCode ()
- void setCounty (string newCounty)
- const string getCounty ()
- void setLat (float newLat)
- const float getLat ()
- void setLon (float newLon)
- const float getLon ()
- const int getSize ()
- void print ()

Static Public Member Functions

- static vector< ZipCode > readFromFile (const string &filename)

Private Attributes

- int num
- float lat
- float lon
- string stateCode
- string city
- string county

3.11.1 Detailed Description

Definition at line 14 of file zipCode.h.

3.11.2 Constructor & Destructor Documentation

3.11.2.1 ZipCode() [1/3]

```
ZipCode::ZipCode ( )
```

Definition at line 15 of file zipCode.cpp.

3.11.2.2 ZipCode() [2/3]

```
ZipCode::ZipCode (
            int newNum,
            string newCity,
            string newStateCode,
            string newCounty,
            float newLat,
            float newLon )
```

Definition at line 27 of file zipCode.cpp.

3.11.2.3 ZipCode() [3/3]

```
ZipCode::ZipCode (
            const ZipCode & oldZip )
```

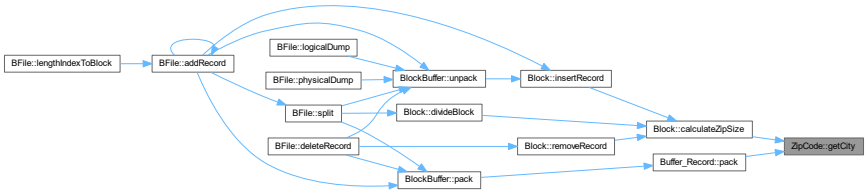
Definition at line 40 of file zipCode.cpp.

3.11.3 Member Function Documentation

3.11.3.1 getCity()

```
const string ZipCode::getCity ( )
```

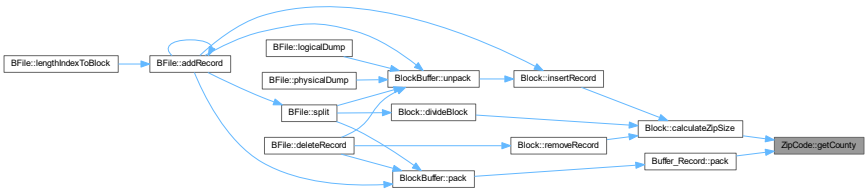
Here is the caller graph for this function:



3.11.3.2 getCounty()

```
const string ZipCode::getCounty ( )
```

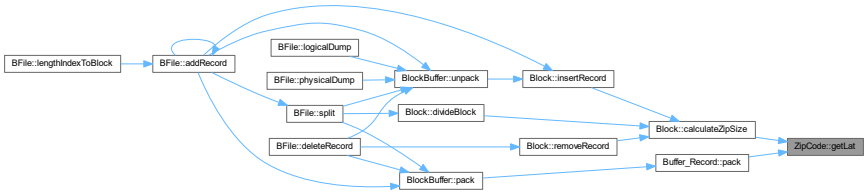
Here is the caller graph for this function:



3.11.3.3 getLat()

```
const float ZipCode::getLat ( )
```

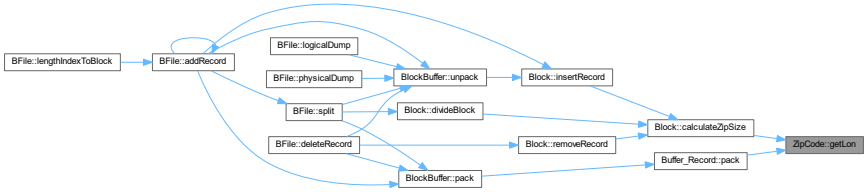
Here is the caller graph for this function:



3.11.3.4 getLon()

```
const float ZipCode::getLon ( )
```

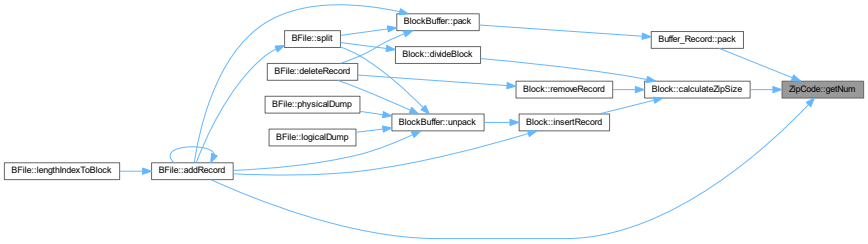
Here is the caller graph for this function:



3.11.3.5 getNum()

```
const int ZipCode::getNum ( )
```

Here is the caller graph for this function:



3.11.3.6 getSize()

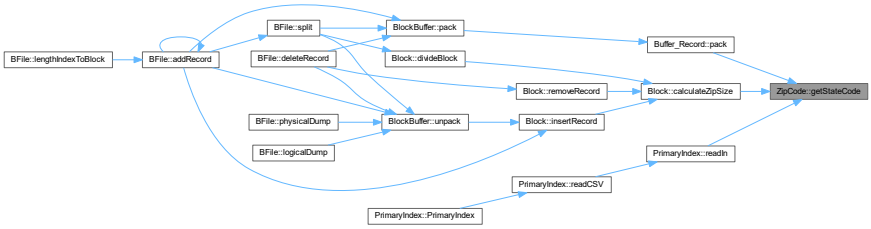
```
const int ZipCode::getSize ( )
```

Definition at line 53 of file zipCode.cpp.

3.11.3.7 getStateCode()

```
const string ZipCode::getStateCode ( )
```

Here is the caller graph for this function:



3.11.3.8 print()

```
void ZipCode::print ( )
```

3.11.3.9 readFromFile()

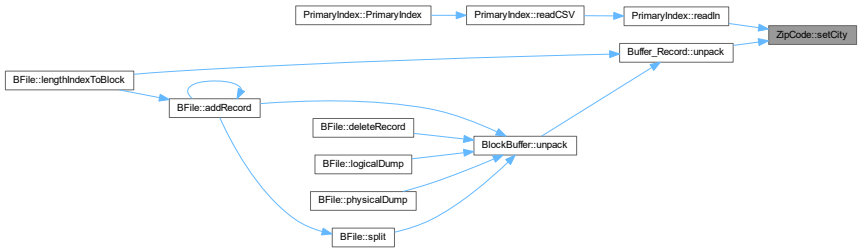
```
vector< ZipCode > ZipCode::readFromFile (
            const string & filename )  [static]
```

Definition at line 79 of file zipCode.cpp.

3.11.3.10 setCity()

```
void ZipCode::setCity (
            string newCity )
```

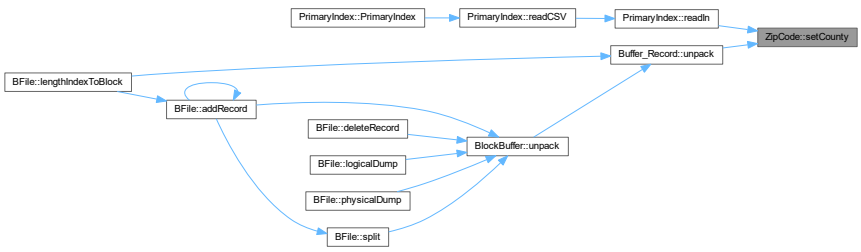
Here is the caller graph for this function:



3.11.3.11 setCounty()

```
void ZipCode::setCounty (
            string newCounty )
```

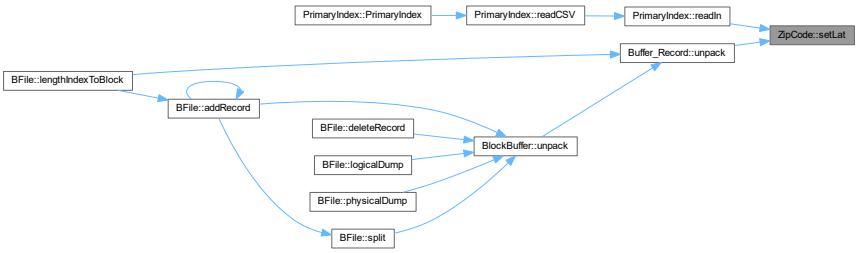
Here is the caller graph for this function:



3.11.3.12 setLat()

```
void ZipCode::setLat (
            float newLat )
```

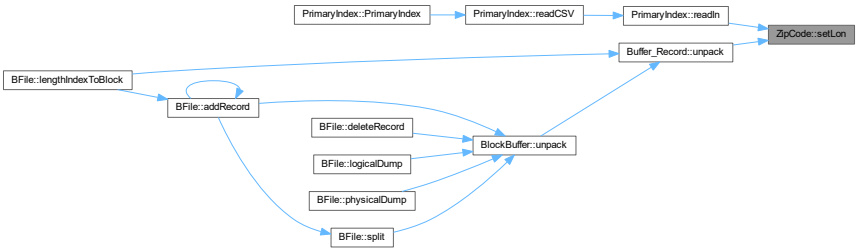
Here is the caller graph for this function:



3.11.3.13 setLon()

```
void ZipCode::setLon (
            float newLon )
```

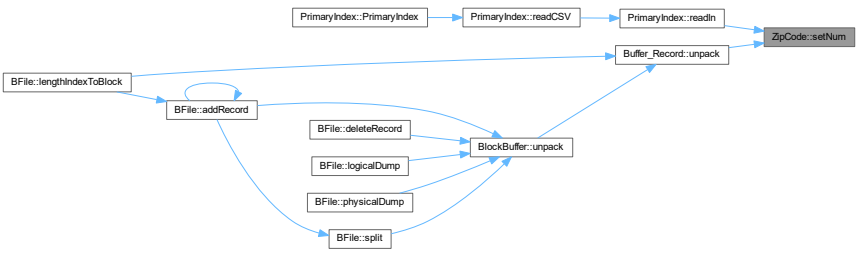
Here is the caller graph for this function:



3.11.3.14 setNum()

```
void ZipCode::setNum (
            int newNum )
```

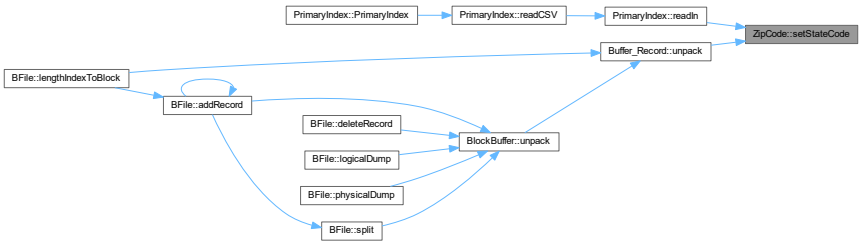
Here is the caller graph for this function:



3.11.3.15 setStateCode()

```
void ZipCode::setStateCode (
            string newStateCode )
```

Here is the caller graph for this function:



3.11.4 Member Data Documentation

3.11.4.1 city

```
string ZipCode::city  [private]
```

Definition at line 65 of file zipCode.h.

3.11.4.2 county

```
string ZipCode::county  [private]
```

Definition at line 66 of file zipCode.h.

3.11.4.3 lat

```
float ZipCode::lat  [private]
```

Definition at line 62 of file zipCode.h.

3.11.4.4 lon

```
float ZipCode::lon  [private]
```

Definition at line 63 of file zipCode.h.

3.11.4.5 num

```
int ZipCode::num  [private]
```

Definition at line 61 of file zipCode.h.

3.11.4.6 stateCode

```
string ZipCode::stateCode  [private]
```

Definition at line 64 of file zipCode.h.

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

- zipCode.h
- zipCode.cpp

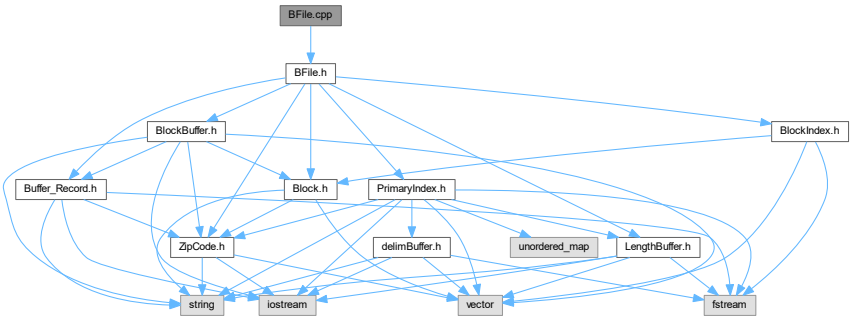
Chapter 4

File Documentation

4.1 BFile.cpp File Reference

Implementation of the BFile class for managing block file operations.

`#include "BFile.h"`
Include dependency graph for BFile.cpp:



4.1.1 Detailed Description

Implementation of the BFile class for managing block file operations.

Definition in file BFile.cpp.

4.2 BFile.cpp

Go to the documentation of this file.

```
00001
00006 #include "BFile.h"
00007
00012 BFile::BFile() {
00013     string index = "IndexFile.index";
00014     string data = "data.txt";
00015
```

```
00016     outputData.open("DataFile.txt");
00017     inputData.open("DataFile.txt");
00018
00019     for (int i = 0; i < 1024; i++) {
00020         outputData « '0';
00021     }
00022
00023     lengthIndexToBlock(index, data);
00024 }
00025
00031 // Converts a length index to a block structure.
00032 void BFile::lengthIndexToBlock(string indexString, string lengthData) {
00033     Buffer_Record recParser;
00034     LengthBuffer libuf;
00035     ZipCode z;
00036     PrimaryIndex pi(indexString, lengthData);
00037
00038     vector<IndexElement> ind;
00039     pi.getIndex(ind);
00040
00041     fstream lid;
00042     lid.open(lengthData);
00043
00044     for (int i = 0; i < ind.size(); i++) {
00045         libuf.read(lid, ind[i].offset);
00046         recParser.read(libuf.getBuffer());
00047         recParser.unpack(z);
00048         addRecord(z);
00049     }
00050     outputData.seekp(0);
00051     outputData « writeHeader();
00052 }
00053
00059 // Deletes a record based on address.
00060 bool BFile::deleteRecord(string zipCode) {
00061     Block currentBlock, previousBlock, nextBlock;
00062     int rbn = blockIndex.Search(stoi(zipCode));
00063     int prevRbn, nextRbn;
00064
00065     if (rbn == 0)
00066         return false;
00067     else {
00068         blockBuffer.read(inputData, rbn);
00069         blockBuffer.unpack(currentBlock);
00070         prevRbn = currentBlock.getPreviousIndex();
00071         nextRbn = currentBlock.getNextIndex();
00072
00073         if (currentBlock.removeRecord(stoi(zipCode))) {
00074             if (currentBlock.getSize() < 256) {
00075                 if (prevRbn != 0) {
00076                     blockBuffer.clear();
00077                     blockBuffer.read(inputData, prevRbn);
00078                     blockBuffer.unpack(previousBlock);
00079                 }
00080                 if (nextRbn != 0) {
00081                     blockBuffer.clear();
00082                     blockBuffer.read(inputData, nextRbn);
00083                     blockBuffer.unpack(nextBlock);
00084                 }
00085                 if (previousBlock.getSize() < 256 && currentBlock.getSize() < 256) {
00086                     Block mergedBlock(previousBlock, currentBlock);
00087                     blockIndex.Add(mergedBlock, prevRbn);
00088                     blockBuffer.pack(mergedBlock);
00089                     blockBuffer.write(outputData, prevRbn);
00090                     nextBlock.setPreviousIndex(prevRbn);
00091                     blockBuffer.clear();
00092                     blockBuffer.pack(nextBlock);
00093                     blockBuffer.write(outputData, nextRbn);
00094                     return true;
00095                 }
00096                 if (nextBlock.getSize() < 256 && currentBlock.getSize() < 256) {
00097                     Block mergedBlock(currentBlock, nextBlock);
00098                     blockIndex.Add(mergedBlock, nextRbn);
00099                     blockBuffer.pack(mergedBlock);
00100                     blockBuffer.write(outputData, nextRbn);
00101                     previousBlock.setNextIndex(nextRbn);
00102                     blockBuffer.clear();
00103                     blockBuffer.pack(previousBlock);
00104                     blockBuffer.write(outputData, prevRbn);
00105                     return true;
00106                 }
00107             }
00108             return true;
00109         } else
00110             return false;
00111     }
00112 }
```

```
00113
00119 // Adds a new ZipCode record to the file.
00120 bool BFile::addRecord(ZipCode &z) {
00121     Block tempBlock, newBlock;
00122     tempBlock.setActiveState(true);
00123     int rbn = blockIndex.Search(z.getNum());
00124
00125     if (rbn == 0) {
00126         if (totalBlocks == 0) totalBlocks++;
00127         rbn = blockIndex.FindHighest();
00128
00129         if (rbn == 0) {
00130             tempBlock.insertRecord(z);
00131             tempBlock.setPreviousIndex(0);
00132             tempBlock.setNextIndex(0);
00133             blockIndex.Add(tempBlock, 1);
00134
00135             blockBuffer.pack(tempBlock);
00136             blockBuffer.write(outputData, 1);
00137             blockBuffer.clear();
00138
00139             return true;
00140         } else {
00141             blockBuffer.read(inputData, rbn);
00142             blockBuffer.unpack(tempBlock);
00143             blockBuffer.clear();
00144             blockIndex.Add(tempBlock, rbn);
00145
00146             if (!tempBlock.insertRecord(z)) {
00147                 split(tempBlock);
00148                 return addRecord(z);
00149             } else {
00150                 blockBuffer.pack(tempBlock);
00151                 blockBuffer.write(outputData, rbn);
00152                 blockBuffer.clear();
00153                 return true;
00154             }
00155         }
00156     } else {
00157         blockBuffer.read(inputData, rbn);
00158         blockBuffer.unpack(tempBlock);
00159         blockBuffer.clear();
00160         blockIndex.Add(tempBlock, rbn);
00161
00162         if (!tempBlock.insertRecord(z)) {
00163             split(tempBlock);
00164             return addRecord(z);
00165         } else {
00166             blockIndex.Add(tempBlock, rbn);
00167             blockBuffer.pack(tempBlock);
00168             blockBuffer.write(outputData, rbn);
00169             blockBuffer.clear();
00170             return true;
00171         }
00172     }
00173     return false;
00174 }
00175
00179 // Reads the file header.
00180 void BFile::readHeader() {
00181     string temp;
00182     inputData.seekg(0);
00183     for (int i = 0; i < FILESIZE; i++) {
00184         temp.push_back(inputData.get());
00185     }
00186 }
00187
00192 // Writes header information to the file.
00193 string BFile::writeHeader() {
00194     string header;
00195     // File Structure Type
00196     header.append("File Structure: Blocked sequence set with comma separated fields, and
      length-indicated records\n");
00197
00198     // File Version
00199     header.append("Version: 1.0\n");
00200
00201     // Header record size
00202     header.append("Header Size: 512 bytes\n");
00203
00204     // Size Format
00205     header.append("Format: ASCII\n");
00206
00207     // Block size
00208     header.append("Block Size: 512 bytes\n");
00209
00210     // Minimum block capacity
```

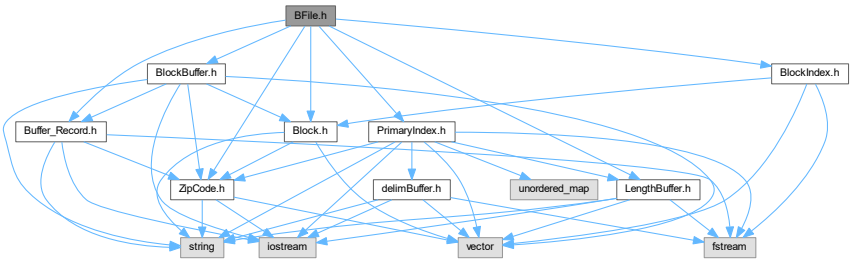
```
00211     header.append("Min Block Capacity: 256 bytes\n");
00212
00213     // Index File Name
00214     header.append("Index File: Index\n");
00215
00216     // Index File Schema
00217     header.append("File Schema: Highest zip code and RBN\n");
00218
00219     // Record count
00220     header.append("Record Count: ");
00221     header.append(to_string(totalRecords));
00222     header.push_back('\n');
00223
00224     // Block count
00225     header.append("Block Count: ");
00226     header.append(to_string(totalBlocks));
00227     header.push_back('\n');
00228
00229     // Fields per record
00230     header.append("Fields: 6\n");
00231
00232     // Fields
00233     header.append("ZipCode, Place Name, State, County, Lat, Long\n");
00234
00235     // Type Schema
00236     header.append("Type Schema: Zip Code is an integer. Lon and Lat are floating point decimals.
      County, State Code, and City are strings.\n");
00237
00238     // Primary key
00239     header.append("Zip code is the first key\n");
00240
00241     // RBN link to avail list
00242     header.append("First Available Block: ");
00243     header.append(to_string(availableSpace));
00244     header.push_back('\n');
00245
00246     // RBN link to active list
00247     header.append("First Active Block: ");
00248     header.append(to_string(firstRBN));
00249     header.push_back('\n');
00250
00251     // Stale flag
00252     header.append("Stale: true");
00253     header.push_back('\n');
00254
00255     return header;
00256 }
00257
00262 // Provides a physical dump of the file's data.
00263 string BFile::physicalDump() {
00264     string output = "";
00265     output.append("List Head: ");
00266     output.append(to_string(getFirstRBN()));
00267     output.append("\nAvail Head: ");
00268     output.append(to_string(getAvailableSpace()));
00269     output.append("\n");
00270
00271     Block tempBlock;
00272     vector<ZipCode> records;
00273
00274     for (int i = 1; i <= totalBlocks; ++i) {
00275         blockBuffer.read(inputData, i);
00276         blockBuffer.unpack(tempBlock);
00277         blockBuffer.clear();
00278
00279         if (tempBlock.isActive()) {
00280             output.append("RBN Prev: ");
00281             output.append(to_string(tempBlock.getPreviousIndex()));
00282             tempBlock.fetchRecords(records);
00283
00284             for (int j = 0; j < tempBlock.getRecordCount(); j++) {
00285                 output.push_back(' ');
00286                 output.append(to_string(records[j].getNum()));
00287                 output.push_back(' ');
00288             }
00289
00290             output.append("RBN Next: ");
00291             output.append(to_string(tempBlock.getNextIndex()));
00292             output.append("\n");
00293         } else {
00294             output.append("RBN Prev:0\t*AVAILABLE*\tRBN Next: 0\n");
00295         }
00296     }
00297
00298     return output;
00299 }
00300
```

```
00305 // Provides a logical dump of the file's data.
00306 string BFile::logicalDump() {
00307     int rbn = 1;
00308     Block tempBlock;
00309     string zips;
00310     vector<ZipCode> records;
00311
00312     zips.append("List Head: ");
00313     zips.append(to_string(getFirstRBN()));
00314     zips.append("\nAvail Head: ");
00315     zips.append(to_string(getAvailableSpace()));
00316     zips.append("\n");
00317
00318     for (int i = 1; i <= totalBlocks; ++i) {
00319         if (rbn == 0) break;
00320         blockBuffer.read(inputData, rbn);
00321         blockBuffer.unpack(tempBlock);
00322         blockBuffer.clear();
00323
00324         if (tempBlock.isActive()) {
00325             zips.append("RBN Prev: ");
00326             zips.append(to_string(tempBlock.getPreviousIndex()));
00327             tempBlock.fetchRecords(records);
00328
00329             for (int j = 0; j < records.size(); j++) {
00330                 zips.append(to_string(records[j].getNum()));
00331                 zips.push_back(' ');
00332             }
00333
00334             zips.append("RBN Prev: ");
00335             zips.append(to_string(tempBlock.getNextIndex()));
00336             zips.push_back('\n');
00337             rbn = tempBlock.getNextIndex();
00338         } else {
00339             zips.append("RBN Prev:0\t*AVAILABLE*\tRBN Next: 0\n");
00340         }
00341     }
00342
00343     return zips;
00344 }
00345
00351 // Splits a block into two parts.
00352 bool BFile::split(Block& b) {
00353     if (b.isActive()) {
00354         int rbn, tempIndex;
00355         Block tempBlock1, tempBlock2;
00356
00357         rbn = blockIndex.Search(b.calculateHighestZip());
00358         b.divideBlock(tempBlock1);
00359
00360         if (b.getNextIndex() == 0) {
00361             tempBlock1.setNextIndex(0);
00362         } else {
00363             tempIndex = b.getNextIndex();
00364             blockBuffer.read(inputData, tempIndex);
00365             blockBuffer.unpack(tempBlock2);
00366
00367             tempBlock1.setNextIndex(tempIndex);
00368             tempBlock2.setPreviousIndex(totalBlocks);
00369             blockBuffer.clear();
00370         }
00371
00372         b.setNextIndex(++totalBlocks);
00373         tempBlock1.setPreviousIndex(rbn);
00374         tempBlock1.setActiveState(true);
00375
00376         blockBuffer.pack(b);
00377         blockBuffer.write(outputData, rbn);
00378         blockBuffer.clear();
00379         blockIndex.Add(b, rbn);
00380
00381         blockBuffer.pack(tempBlock1);
00382         blockBuffer.write(outputData, totalBlocks);
00383         blockBuffer.clear();
00384         blockIndex.Add(tempBlock1, totalBlocks);
00385
00386         return true;
00387     }
00388     return false;
00389 }
```

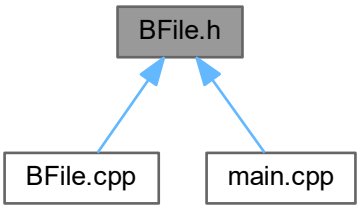
4.3 BFile.h File Reference

```
#include "BlockBuffer.h"
#include "Buffer_Record.h"
#include "zipCode.h"
#include "Block.h"
#include "BlockIndex.h"
#include "LengthBuffer.h"
#include "PrimaryIndex.h"
```

Include dependency graph for BFile.h:



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



Classes

- class BFile

Macros

- #define BFILE

Variables

- const int FILESIZE = 512

4.3.1 Macro Definition Documentation

4.3.1.1 BFILE

```
#define BFILE
```

Definition at line 4 of file BFile.h.

4.3.2 Variable Documentation

4.3.2.1 FILESIZE

```
const int FILESIZE = 512
```

Definition at line 14 of file BFile.h.

4.4 BFile.h

Go to the documentation of this file.

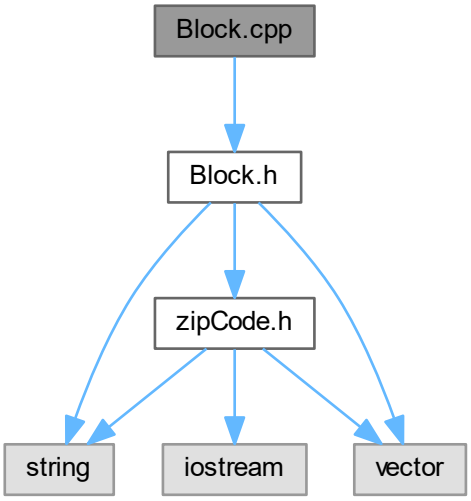
```
00001 #pragma once
00002
00003 #ifndef BFILE
00004 #define BFILE
00005
00006 #include "BlockBuffer.h"
00007 #include "Buffer_Record.h"
00008 #include "zipCode.h"
00009 #include "Block.h"
00010 #include "BlockIndex.h"
00011 #include "LengthBuffer.h"
00012 #include "PrimaryIndex.h"
00013
00014 const int FILESIZE = 512;
00015
00016 class BFile {
00017 public:
00021     BFile() {
00022         // Default constructor implementation
00023     }
00024
00029     BFile(string fileName) {
00030         open(fileName);
00031     }
00032
00038     void lengthIndexToBlock(string indexString, string lengthData);
00039
00044     void open(string fileName) {
00045         inputData.open(fileName);
00046         outputData.open(fileName);
00047     }
00048
00052     void close() {
00053         inputData.close();
00054         outputData.close();
00055     }
00056
00060     void readHeader();
00061
00066     string writeHeader();
00067
00072     string physicalDump();
00073
00078     string logicalDump();
00079
00085     bool split(Block& b);
00086
00092     bool addRecord(ZipCode& zipCode);
00093
00099     bool deleteRecord(string zipCode);
```

```
00100
00105     int getFirstRBN() const { return firstRBN; }
00106
00111     int getAvailableSpace() const { return availableSpace; }
00112
00113 private:
00114     int firstRBN, availableSpace, totalBlocks, totalRecords;
00115
00116     ofstream outputData;
00117     ifstream inputData;
00118     BlockBuffer blockBuffer;
00119     BlockIndex blockIndex;
00120 };
00121
00122 #endif //BFILE
```

4.5 Block.cpp File Reference

Implementation of the Block class for managing collections of ZipCode records.

`#include "Block.h"`
Include dependency graph for Block.cpp:



4.5.1 Detailed Description

Implementation of the Block class for managing collections of ZipCode records.

Author

Group 7

This file contains the implementation of the Block class, which includes methods for manipulating blocks of ZipCode records, such as adding, removing, and splitting blocks.

Definition in file Block.cpp.

4.6 Block.cpp

Go to the documentation of this file.

```
00001
00010 #include "Block.h"
00011
00016 // Constructor: Initializes a new, empty block
00017 Block::Block() {
00018     active = false;
00019     recCount = 0;
00020     currentSize = calculateHeaderSize() + 1;
00021     highestZip = 0;
00022     prev = 0;
00023     next = 0;
00024     records.clear();
00025 }
00026
00032 // Copy constructor: Creates a copy of an existing block
00033 Block::Block(const Block& old) {
00034     active = old.active;
00035     recCount = old.recCount;
00036     currentSize = old.currentSize;
00037     highestZip = old.highestZip;
00038     prev = old.prev;
00039     next = old.next;
00040     records = old.records; // Using direct assignment for vector copy
00041 }
00042
00049 // Merge constructor: Merges two blocks into one
00050 Block::Block(Block& firstBlock, Block& secondBlock) {
00051     if (secondBlock.getMaximumZip() < firstBlock.getMaximumZip()) {
00052         swap(firstBlock, secondBlock);
00053     }
00054
00055     records.reserve(firstBlock.records.size() + secondBlock.records.size());
00056     records.insert(records.end(), firstBlock.records.begin(), firstBlock.records.end());
00057     records.insert(records.end(), secondBlock.records.begin(), secondBlock.records.end());
00058
00059     active = true;
00060     recCount = firstBlock.recCount + secondBlock.recCount;
00061     currentSize = firstBlock.currentSize + secondBlock.currentSize;
00062     prev = firstBlock.prev;
00063     next = secondBlock.next;
00064     secondBlock.active = false;
00065
00066     calculateHighestZip();
00067 }
00068
00076 // Inserts a new ZipCode record into the block
00077 bool Block::insertRecord(ZipCode& newZip) {
00078     int tempsize = calculateHeaderSize();
00079     int count = calculateZipSize(newZip);
00080
00081     if (count + currentSize < 512) {
00082         auto position = lower_bound(records.begin(), records.end(), newZip,
00083                                     [](const ZipCode& a, const ZipCode& b) { return a.getNum() <
      b.getNum(); });
00084         records.insert(position, newZip);
00085         recCount++;
00086         calculateHighestZip();
00087         currentSize = (currentSize - tempsize) + calculateHeaderSize();
00088         return true;
00089     }
00090     return false;
00091 }
00092
00099 // Splits the block into two blocks
00100 void Block::divideBlock(Block& newBlock) {
00101     int midpoint = records.size() / 2;
00102     newBlock.records.assign(records.begin() + midpoint, records.end());
00103     newBlock.recCount = newBlock.records.size();
00104     newBlock.calculateHighestZip();
00105     newBlock.currentSize = newBlock.calculateHeaderSize() +
      newBlock.calculateZipSize(newBlock.records);
00106
00107     records.resize(midpoint);
00108     recCount = records.size();
00109     calculateHighestZip();
00110     currentSize = calculateHeaderSize() + calculateZipSize(records);
00111
00112     newBlock.setNextIndex(getNextIndex());
00113     newBlock.setPreviousIndex(this); // Assuming 'prev' is an int, not a pointer
00114 }
00115
00123 // Removes a ZipCode record from the block
```

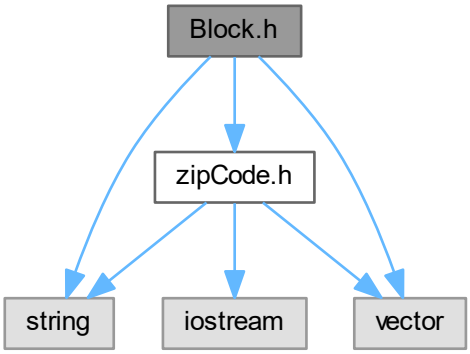
```
00124 bool Block::removeRecord(int zip) {
00125     auto it = find_if(records.begin(), records.end(), [zip](const ZipCode& record) { return
      record.getNum() == zip; });
00126     if (it != records.end()) {
00127         int count = calculateZipSize(*it);
00128         records.erase(it);
00129         recCount--;
00130         currentSize -= count;
00131         calculateHighestZip();
00132         active = recCount > 0;
00133         return true;
00134     }
00135     return false;
00136 }
00137
00143 // Calculates the highest ZIP code in the block
00144 int Block::calculateHighestZip() {
00145     highestZip = records.empty() ? 0 : max_element(records.begin(), records.end(),
00146         [](const ZipCode& a, const ZipCode& b) { return a.getNum() < b.getNum(); })->getNum();
00147     return highestZip;
00148 }
00149
00155 // Calculates the size of a ZipCode record
00156 int Block::calculateZipSize(const ZipCode& zipper) const {
00157     ostringstream oss;
00158     oss « zipper.getNum() « ',' « zipper.getCity() « ',' « zipper.getStateCode() « ','
00159         « zipper.getCounty() « ',' « zipper.getLat() « ',' « zipper.getLon();
00160     return oss.str().size();
00161 }
00162
00167 // Calculates the size of the block header
00168 int Block::calculateHeaderSize() const {
00169     ostringstream oss;
00170     oss « prev « ',' « next « ',' « recCount « ',' « currentSize « ',' « highestZip « ';';
00171     return oss.str().size();
00172 }
00173
00179 // Retrieves all ZipCode records in the block
00180 void Block::fetchRecords(vector<ZipCode>& recordsOut) const {
00181     recordsOut = records;
00182 }
00183
00191 // Searches for a specific ZipCode in the block
00192 bool Block::searchZip(ZipCode& resultZip, int target) {
00193     auto it = find_if(records.begin(), records.end(), [target](const ZipCode& record) { return
      record.getNum() == target; });
00194     if (it != records.end()) {
00195         resultZip = *it;
00196         return true;
00197     }
00198     return false;
00199 }
```

4.7 Block.h File Reference

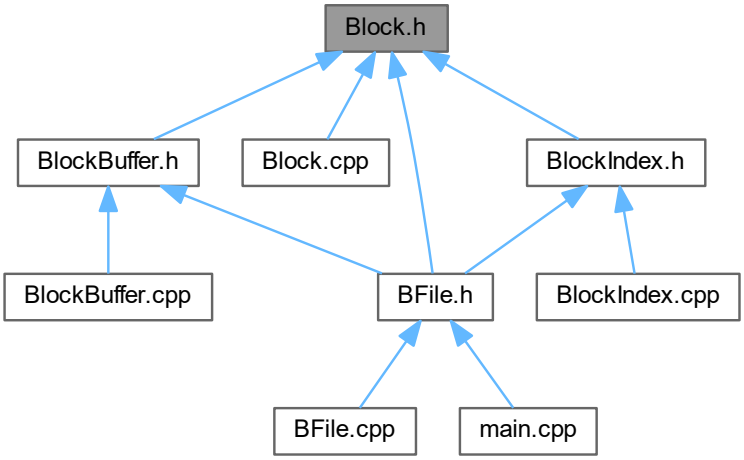
Defines the Block class for managing collections of ZipCode records.

```
#include "zipCode.h"
#include <string>
#include <vector>
```

Include dependency graph for Block.h:



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



Classes

- class Block

Macros

- #define BLOCK

4.7.1 Detailed Description

Defines the Block class for managing collections of ZipCode records.

Definition in file Block.h.

4.7.2 Macro Definition Documentation

4.7.2.1 BLOCK

```
#define BLOCK
```

Definition at line 15 of file Block.h.

4.8 Block.h

Go to the documentation of this file.

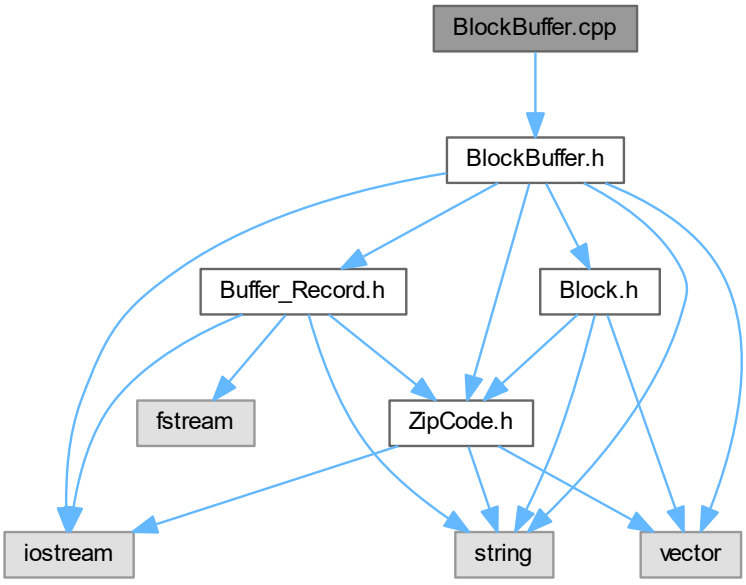
```
00001
00005 #pragma once
00006
00007
00008 #include "zipCode.h"
00009 #include <string>
00010 #include <vector>
00011 using namespace std;
00012
00013
00014 #ifndef BLOCK
00015 #define BLOCK
00016
00017
00018
00019 class Block {
00020 public:
00025     // Default constructor
00026     Block();
00027
00033     // Copy constructor
00034     Block(const Block& old);
00035
00041     // Merge constructor
00042     Block(Block& firstBlock, Block& secondBlock);
00043
00049     // Inserts a new ZipCode record
00050     bool insertRecord(ZipCode& newZip);
00051
00057     // Removes a ZipCode record
00058     bool removeRecord(int zip);
00059
00065     // Splits the Block into two parts
00066     void divideBlock(Block& newBlock);
00067
00068     // Getters with Doxygen @brief tags
00072     int getNextIndex() const { return next; };
00073
00077     int getPreviousIndex() const { return prev; };
00078
00082     bool isActive() const { return active; };
00083
00087     int getRecordCount() const { return recCount; };
00088
00092     int getSize() const { return currentSize; };
00093
00097     int getMaximumZip() const { return highestZip; };
00098
00099     // Other methods
00100     void fetchRecords(vector<ZipCode>& recordsOut) const;
00101     bool searchZip(ZipCode& resultZip, int target);
00102
00103     // Setters
00104     void setActiveState(bool state) { active = state; };
```

```
00105     void setNextIndex(int next) { this->next = next; };
00106     void setPreviousIndex(int prev) { this->prev = prev; };
00107     void setRecordCount(int recCount) { this->recCount = recCount; };
00108     void setSize(int currentSize) { this->currentSize = currentSize; };
00109     void setMaximumZip(int highestZip) { this->highestZip = highestZip; };
00110
00111     // Calculate the highest ZIP code
00112     int calculateHighestZip();
00113
00114 private:
00115     // Calculate the size of a ZipCode record
00116     int calculateZipSize(const ZipCode& zipper) const;
00117
00118     // Calculate the size of the Block header
00119     int calculateHeaderSize() const;
00120
00121     // Member variables
00122     bool active;
00123     int prev, next;
00124     int highestZip, recCount, currentSize;
00125     vector<ZipCode> records;
00126 };
```

4.9 BlockBuffer.cpp File Reference

Implementation of the BlockBuffer class for handling block operations in files.

`#include "BlockBuffer.h"`
Include dependency graph for BlockBuffer.cpp:



4.9.1 Detailed Description

Implementation of the BlockBuffer class for handling block operations in files.

Definition in file BlockBuffer.cpp.

4.10 BlockBuffer.cpp

Go to the documentation of this file.

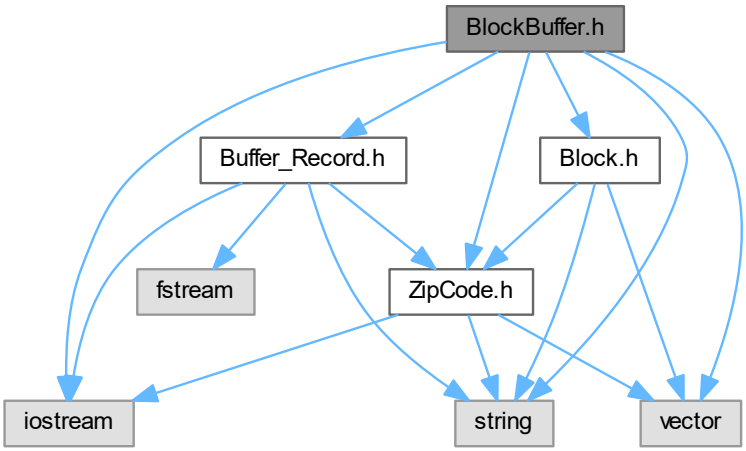
```
00001
00006 #include "BlockBuffer.h"
00007
00014 void BlockBuffer::read(ifstream& inFile, int RBN) {
00015     unsigned long NBR = static_cast<unsigned long>(RBN) * 512;
00016     int index = 0;
00017     inFile.seekg(NBR);
00018     while (!inFile.eof() && index != BUFSIZE) {
00019         blockText.push_back(inFile.get());
00020         index++;
00021     }
00022     index = 0;
00023 }
00024
00029 // pack & storerecords
00030 void BlockBuffer::pack(Block& b) {
00031     Buffer_Record rec;
00032     vector<ZipCode> records;
00033
00034     string temp = writeHeader(b);
00035     blockText.append(temp);
00036
00037
00038     b.fetchRecords(records);
00039     for (auto& record : records) {
00040         rec.pack(record);
00041         rec.write(blockText);
00042     }
00043 }
00044
00050 void BlockBuffer::write(ofstream& outfile, int RBN) {
00051     unsigned long NBR = static_cast<unsigned long>(RBN) * 512;
00052     outfile.seekp(NBR);
00053
00054     for (int i = 0; i < 512; i++) {
00055         outfile « (i < blockText.size() ? blockText[i] : ' ');
00056     }
00057     outfile.flush();
00058     blockText = "";
00059 }
00060
00065 void BlockBuffer::unpack(Block& b) {
00066     readHeader(b);
00067     // Unpack blockText into Block object
00068     ZipCode tempZip;
00069     string temp;
00070     Buffer_Record rec;
00071     int recSize;
00072     int count;
00073     int numRecs = b.getRecordCount();
00074     int recCounter = 0;
00075     int tempCurrentSize = b.getSize();
00076
00077     b.setSize(0);
00078
00079     while (index != tempCurrentSize) {
00080         if (recCounter == numRecs)
00081             break;
00082
00083         while (blockText[index] != ',') {
00084             temp.push_back(blockText[index++]);
00085         }
00086         index++;
00087         recSize = stoi(temp) - 3;
00088         temp = "";
00089         count = 0;
00090         while (index != blockText.size() && count != recSize) {
00091             temp.push_back(blockText[index++]);
00092             count++;
00093         }
00094
00095         rec.read(temp);
00096         rec.unpack(tempZip);
00097
00098         b.insertRecord(tempZip);
00099         recCounter++;
00100         temp = "";
00101     }
00102     index = 0;
00103     blockText = "";
00104     b.setSize(tempCurrentSize);
00105 }
```

```
00106
00111 // Parse header data from blockText
00112 void BlockBuffer::readHeader(Block& b) {
00113     int headerSize;
00114     index = 0;
00115     string temp;
00116
00117     for (index; blockText[index] != ',' ; index++) {
00118         temp.push_back(blockText[index]);
00119     }
00120     index++;
00121     b.setPreviousIndex(stoi(temp));
00122     temp = "";
00123
00124     for (index; blockText[index] != ',' ; index++) {
00125         temp.push_back(blockText[index]);
00126     }
00127     index++;
00128     b.setNextIndex(stoi(temp));
00129     temp = "";
00130
00131     for (index; blockText[index] != ',' ; index++) {
00132         temp.push_back(blockText[index]);
00133     }
00134     index++;
00135     b.setRecordCount(stoi(temp));
00136     temp = "";
00137
00138     for (index; blockText[index] != ',' ; index++) {
00139         temp.push_back(blockText[index]);
00140     }
00141     index++;
00142     b.setSize(stoi(temp));
00143     temp = "";
00144
00145     for (index; blockText[index] != ';' ; index++) {
00146         temp.push_back(blockText[index]);
00147     }
00148     index++;
00149     b.setMaximumZip(stoi(temp));
00150     temp = "";
00151 }
00152
00158 string BlockBuffer::writeHeader(Block& b) {
00159 // Create header string from Block object's attributes
00160     string header, temp;
00161 // Store block attributes as ASCII
00162     header.append(to_string(b.getPreviousIndex()));
00163     header.push_back(',');
00164     header.append(to_string(b.getNextIndex()));
00165     header.push_back(',');
00166     header.append(to_string(b.getRecordCount()));
00167     header.push_back(',');
00168     header.append(to_string(b.getSize()));
00169     header.push_back(',');
00170     header.append(to_string(b.getMaximumZip()));
00171     header.push_back(';');
00172     return header;
00173 }
```

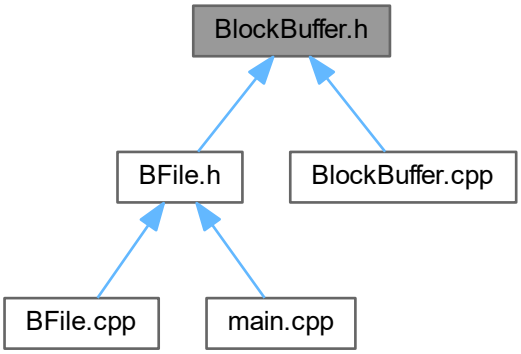
4.11 BlockBuffer.h File Reference

```
#include <vector>
#include <iostream>
#include <string>
#include "ZipCode.h"
#include "Buffer_Record.h"
#include "Block.h"
```

Include dependency graph for BlockBuffer.h:



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



Classes

- class BlockBuffer

Macros

- #define BLOCKBUFFER

Variables

- const int BUFSIZE = 512

4.11.1 Macro Definition Documentation

4.11.1.1 BLOCKBUFFER

```
#define BLOCKBUFFER
```

Definition at line 5 of file BlockBuffer.h.

4.11.2 Variable Documentation

4.11.2.1 BUFSIZE

```
const int BUFSIZE = 512
```

Definition at line 16 of file BlockBuffer.h.

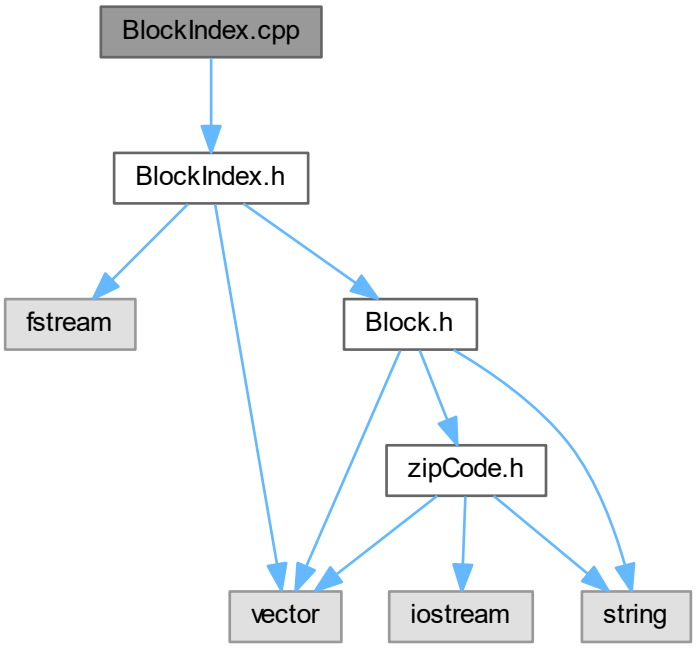
4.12 BlockBuffer.h

Go to the documentation of this file.

```
00001 // BlockBuffer.h
00002 #pragma once
00003
00004 #ifndef BLOCKBUFFER
00005 #define BLOCKBUFFER
00006
00007 #include <vector>
00008 #include <iostream>
00009 #include <string>
00010 #include "ZipCode.h"
00011 #include "Buffer_Record.h"
00012 #include "Block.h"
00013
00014 using namespace std;
00015
00016 const int BUFSIZE = 512;
00017
00018 class BlockBuffer {
00019 public:
00023     BlockBuffer() : blockText(""), index(0) {}
00024
00031     void read(ifstream& infile, int RBN);
00032
00038     void pack(Block& b);
00039
00046     void write(ofstream& outfile, int RBN);
00047
00053     void unpack(Block& b);
00054
00059     string getText() const { return blockText; };
00060
00065     void clear() { blockText.clear(); index = 0; };
00066
00067 private:
00073     void readHeader(Block& b);
00074
00080     string writeHeader(Block& b);
00081
00082     string blockText;  // Text buffer for storing block content
00083     Block obj;         // Block object for temporary storage
00084     int index;         // Index used in reading and writing operations
00085 };
00086
00087 #endif // BLOCKBUFFER
```

4.13 BlockIndex.cpp File Reference

`#include "BlockIndex.h"`
Include dependency graph for BlockIndex.cpp:



4.14 BlockIndex.cpp

Go to the documentation of this file.

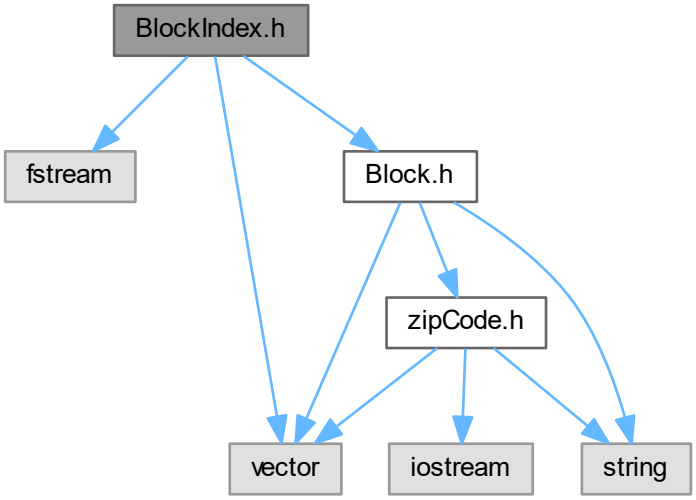
```
00001 /*
00002 BlockIndex.cpp
00003 */
00004 #include "BlockIndex.h"
00005
00006 using namespace std;
00007
00008 int BlockIndex::FindHighest() {
00009     int tempZip = 0;
00010     int tempRBN = 0;
00011
00012     if (index.size() == 0) {
00013         return 0;
00014     }
00015
00016     int i = 0;
00017     while(i < index.size()){
00018         if (index[i].zipCode > tempZip) {
00019             tempZip = index[i].zipCode;
00020             tempRBN = index[i].RBN;
00021         }
00022         i++;
00023     }
00024
00025     return tempRBN;
00026 }
00027
```

```
00028 int BlockIndex::Search(int zip) {
00029
00030     if (index.size() == 0)
00031         return 0;
00032
00033     int tempRBN = 0;
00034     bool found = false;
00035     int tempZip;
00036
00037     int i = 0;
00038     while (i < index.size()) {
00039     if (found) {
00040         if (index[i].zipCode >= zip && index[i].zipCode < tempZip) {
00041             tempRBN = index[i].RBN;
00042             tempZip = index[i].zipCode;
00043         }
00044     } else if (index[i].zipCode >= zip) {
00045         tempRBN = index[i].RBN;
00046         tempZip = index[i].zipCode;
00047         found = true;
00048     }
00049     i++;
00050   }
00051   return tempRBN;
00052 }
00053
00054 void BlockIndex::Add(Block& b, int r) {
00055     BlockIndexVariables temp;
00056     temp.zipCode = b.getMaximumZip();
00057     temp.RBN = r;
00058     temp.active = true;
00059     Del(r);
00060     index.push_back(temp);
00061 }
00062
00063 void BlockIndex::Del(int r) {
00064     if (index.size() == 0) {
00065         return;
00066     }
00067     for (int i = 0; i < index.size(); i++) {
00068         if (index[i].RBN == r) {
00069             index.erase(index.begin() + i);
00070             break;
00071         }
00072     }
00073 }
00074
00075 void BlockIndex::ReadFromFile(string in) {
00076
00077     ifstream iFile;
00078     iFile.open(in);
00079     char trash;
00080     BlockIndexVariables temp;
00081
00082     if (iFile » numBlocks » trash » numAvail » trash) {
00083
00084         index.resize(1);
00085         iFile » index[0].zipCode » trash » index[0].RBN » trash » index[0].active » trash;
00086
00087         for (int i = 0; i < numBlocks && !iFile.eof(); i++) {
00088
00089             iFile » temp.zipCode » trash » temp.RBN » trash » temp.active » trash;
00090             index.push_back(temp);
00091         }
00092     }
00093 }
00094
00095 void BlockIndex::PrintToFile(string out) {
00096
00097     ofstream oFile;
00098     oFile.open(out);
00099
00100     oFile « numBlocks « ',' « numAvail « ';';
00101
00102     int i = 0;
00103     while (i < index.size()) {
00104         oFile « index[i].zipCode « ',' « index[i].RBN « ',' « index[i].active « ';';
00105         i++;
00106     }
00107
00108 }
```

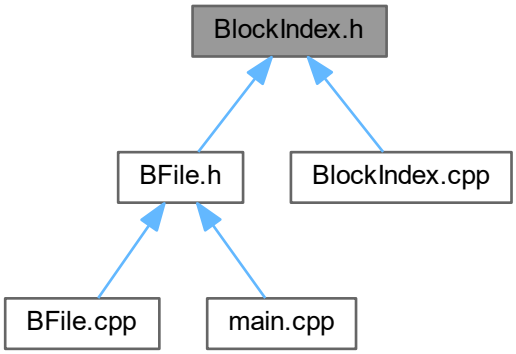
4.15 BlockIndex.h File Reference

```
#include <fstream>
#include <vector>
#include "Block.h"
```

Include dependency graph for BlockIndex.h:



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



Classes

- struct BlockIndexVariables
- class BlockIndex

Macros

- #define BLOCKINDEX_H

4.15.1 Macro Definition Documentation

4.15.1.1 BLOCKINDEX_H

```
#define BLOCKINDEX_H
```

BlockIndex.h Class containing the block index for the highest zip in each block

Definition at line 9 of file BlockIndex.h.

4.16 BlockIndex.h

Go to the documentation of this file.

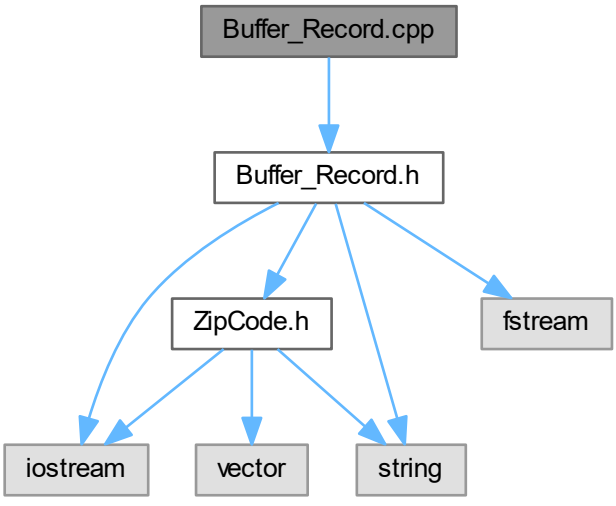
```
00001
00006 #pragma once
00007
00008 #ifndef BLOCKINDEX_H
00009 #define BLOCKINDEX_H
00010
00011 #include <fstream>
00012 #include <vector>
00013 #include "Block.h"
00014
00015 using namespace std;
00016
00017 struct BlockIndexVariables {
00018     int zipCode;
00019     int RBN;
00020     bool active;
00021 };
00022
00023 class BlockIndex {
00024
00025 private:
00026     int numBlocks, numAvail;
00027     vector<BlockIndexVariables> index;
00028
00029 public:
00030     /*
00031     * @brief Default constructor
00032     * @pre
00033     * @post
00034     */
00035     BlockIndex() {
00036         index.clear();
00037          }
00038
00039     /*
00040     * @brief Search function
00041     * @pre Takes an integer ZIP to search through the index
00042     * @post Returns the block number that would contain this zip as an int
00043     */
00044     int Search(int zipCode);
00045
00046     int FindHighest();
00047
00048     /*
00049     * @brief Print to file function
00050     * @post Outputs the content of the index to the file
00051     */
00052     void PrintToFile(string);
00053
00054     /*
00055     * @brief Add function
00056     * @pre Adds the first parameter as a zip and the second as a RBN onto the back of the index
00057     * @post Returns void
00058     */
00059     void Add(Block& b, int r);
```

```
00060
00061     /*
00062     * @brief Delete function
00063     * @pre Takes a block number to delete from the index
00064     * @post Returns void
00065     */
00066     void Del(int r);
00067
00072     void ReadFromFile(string);
00073     /*
00074     * @brief Get number of available function
00075     * @post Returns the number of blocks available as an int
00076     */
00077     int GetNumAvail() {
00078         return numAvail;
00079         }
00080
00081     /*
00082     * @brief Get number of blocks function
00083     * @post Returns the number of blocks as an int
00084     */
00085     int GetNumBlocks() {
00086         return numBlocks;
00087          };
00088
00089 };
00090
00091 #endif // BLOCKINDEX_H
```

4.17 Buffer_Record.cpp File Reference

Implementation of the Buffer_Record class for handling record buffering.

`#include "Buffer_Record.h"`
Include dependency graph for Buffer_Record.cpp:



4.17.1 Detailed Description

Implementation of the Buffer_Record class for handling record buffering.

Author

Group 7

This file implements the Buffer_Record class, providing functionality to pack, unpack, read, and write ZipCode records into a string buffer.

Definition in file Buffer_Record.cpp.

4.18 Buffer_Record.cpp

Go to the documentation of this file.

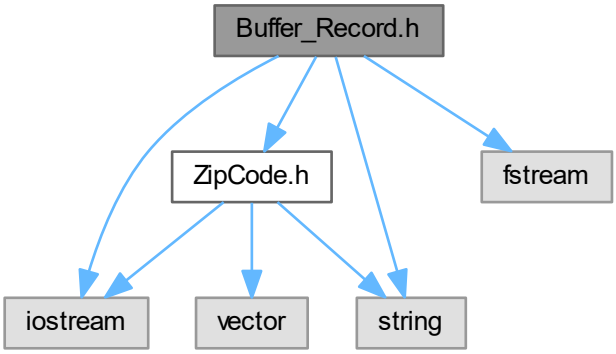
```
00001
00009 #include "Buffer_Record.h"
00010
00015 // Packs a ZipCode object into the buffer
00016 void Buffer_Record::pack(ZipCode& z) {
00017     std::string temp;
00018
00019     temp.push_back(',');
00020     temp.append(std::to_string(z.getNum()));
00021     temp.push_back(',');
00022     temp.append(z.getCity());
00023     temp.push_back(',');
00024     temp.append(z.getStateCode());
00025     temp.push_back(',');
00026     temp.append(z.getCounty());
00027     temp.push_back(',');
00028     temp.append(std::to_string(z.getLat()));
00029     temp.push_back(',');
00030     temp.append(std::to_string(z.getLon()));
00031     buf.append(std::to_string(temp.size() + 2));
00032     buf.append(temp);
00033 }
00034
00040 // Reads a string into the buffer
00041 void Buffer_Record::read(const std::string& recText) {
00042     buf = recText;
00043 }
00044
00050 // Writes buffer content into a given string
00051 void Buffer_Record::write(std::string& blockText) {
00052     blockText.append(buf);
00053     buf = "";
00054 }
00055
00062 // Unpacks the buffer into a ZipCode object
00063 bool Buffer_Record::unpack(ZipCode& z) {
00064     std::string temp;
00065     int size = buf.size();
00066     temp = "";
00067     index = 0;
00068
00069     int fieldNumber = 1;
00070
00071     if (index != size) { // execute only when buf is not empty
00072         for (int i = 0; i < size && index < buf.size() + 1; i++) {
00073             if (buf[index] != delim) {
00074                 temp.push_back(buf[index]);
00075                 if (buf[index + 1] == delim || index + 1 == size) {
00076                     if (fieldNumber == 1) {
00077                         z.setNum(std::stoi(temp));
00078                     } else if (fieldNumber == 2) {
00079                         z.setCity(temp);
00080                     } else if (fieldNumber == 3) {
00081                         z.setStateCode(temp);
00082                     } else if (fieldNumber == 4) {
00083                         z.setCounty(temp);
00084                     } else if (fieldNumber == 5) {
00085                         z.setLat(std::stof(temp));
00086                     } else {
00087                         z.setLon(std::stof(temp));
00088                     }
00089                     fieldNumber++;
00090                     temp = "";
00091                 }
00092             }
00093             index++;
```

```
00094         }
00095         return true;
00096     }
00097
00098     return false;
00099 }
00104 void Buffer_Record::clear() {
00105     // Implementation
00106 }
```

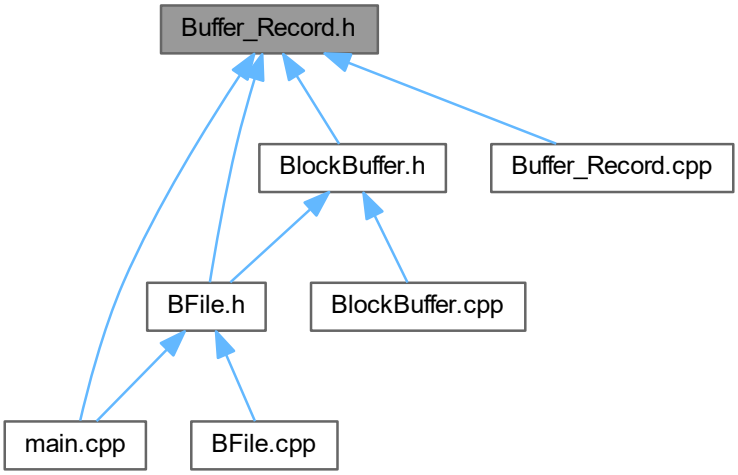
4.19 Buffer_Record.h File Reference

```
#include <iostream>
#include <string>
#include <fstream>
#include "ZipCode.h"
```

Include dependency graph for Buffer_Record.h:



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



Classes

- class Buffer_Record

Macros

- #define BUFFER_RECORD

4.19.1 Macro Definition Documentation

4.19.1.1 BUFFER_RECORD

```
#define BUFFER_RECORD
```

Definition at line 5 of file Buffer_Record.h.

4.20 Buffer_Record.h

Go to the documentation of this file.

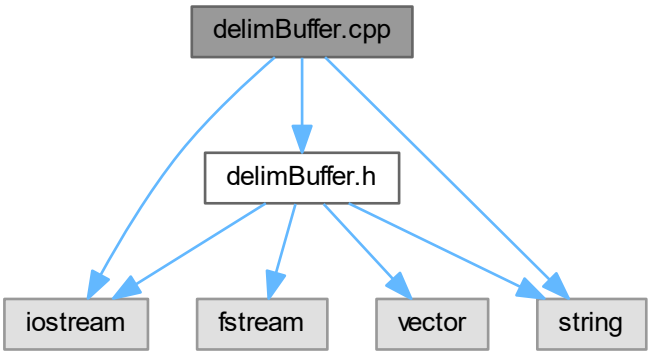
```
00001
00002 //Buffer_Record.h
00003 #pragma once
00004 #ifndef BUFFER_RECORD
00005 #define BUFFER_RECORD
00006
00007 #include <iostream>
00008 #include <string>
00009 #include <fstream>
00010 #include "ZipCode.h"
00011
```

```
00012 class Buffer_Record {
00013 public:
00018     // Default Constructor - Initializes a new Buffer_Record object
00019     Buffer_Record() : buf(""), size(buf.size()), index(0) {}
00020
00027     // Reads record text into the buffer
00028     void read(const std::string& recText);
00029
00036     // Writes buffer content into blockText
00037     void write(std::string& blockText);
00038
00045     // Packs a ZipCode object into the buffer
00046     void pack(ZipCode& zipCode);
00047
00055     // Unpacks the buffer into a ZipCode object
00056     bool unpack(ZipCode& zipCode);
00057
00063     // Clears the contents of the buffer
00064     void clear() {
00065         buf = "";
00066         size = 0;
00067         index = 0;
00068     }
00069
00070 private:
00071     std::string buf;      // The buffer for storing record data
00072     const char delim = ','; // Delimiter for separating record fields
00073     int size;              // Size of the buffer
00074     int index;             // Index for tracking read/write operations
00075 };
00076
00077 #endif // BUFFER_RECORD
```

4.21 delimBuffer.cpp File Reference

```
#include "delimBuffer.h"
#include <iostream>
#include <string>
```

Include dependency graph for delimBuffer.cpp:



4.22 delimBuffer.cpp

Go to the documentation of this file.

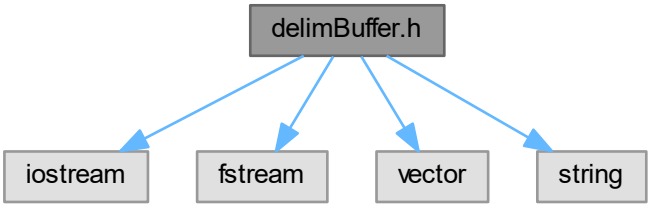
```
00001
```

```
00005 #include "delimBuffer.h"
00006 #include <iostream>
00007 #include <string>
00008
00009 /*
00010 @brief General constructor for the DelimBuffer
00011 */
00012 delimBuffer::delimBuffer() {
00013     delim = ',';
00014     index = 0;
00015     buffer = "";
00016     size = 0;
00017     max = 1000;
00018 }
00019
00020 /*
00021 @brief Builds the specified constructor.
00022 @pre Receives the delimeter, and the maximum integer.
00023 @param1 delim a character which is a comma.
00024 @param2 maxsize an int which is the maxsizef the buffer.
00025 */
00026 delimBuffer::delimBuffer(char delim = ',', int maxsize = 1000) {
00027     index = 0;
00028     buffer = "";
00029     size = 0;
00030 }
00031
00032 /*
00033 @brief Reads from the delim file.
00034 @pre Receives a file and the and the offset for a record.
00035 @param1 inFile an ifstream variable which contains the address of the length-indicated file.
00036 @param2 offset an integer variable which contains the offset for the specific record.
00037 */
00038 bool delimBuffer::read(ifstream& inFile) {
00039     index = 0;
00040     buffer = "";
00041     if (inFile.is_open() && !inFile.eof()) {                // execute only when the file is open and
      not at the end of the file
00042         getline(inFile,buffer);                               // pull everything up to the next
      newline
00043         size = buffer.size();
00044         return true;
00045     }
00046     else
00047         return false;
00048
00049 }
00050
00051
00052 bool delimBuffer::unpack(string& field) {
00053     if (index != size && size != 0) { // execute only when delimBuffer is not empty
00054
00055         while (index < size && buffer[index] != delim) { //
00056             field.push_back(buffer[index++]);
00057         }
00058         if (buffer[index] == delim)
00059             index++;
00060         return true;
00061
00062     }
00063     return false;
00064 }
```

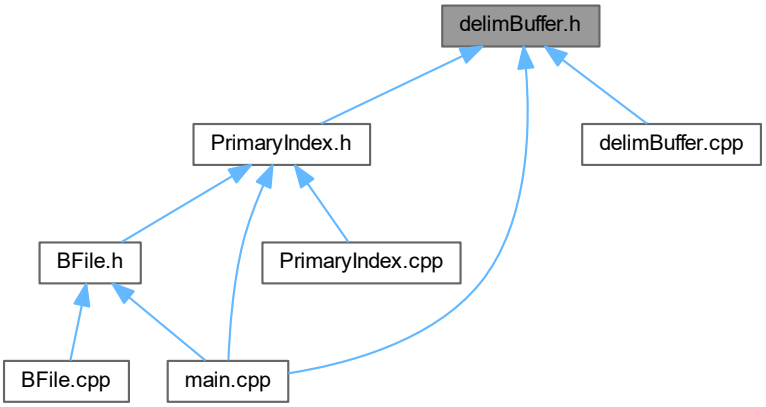
4.23 delimBuffer.h File Reference

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

Include dependency graph for delimBuffer.h:



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



Classes

- class delimBuffer

  class to store each record and parse each field

4.24 delimBuffer.h

Go to the documentation of this file.

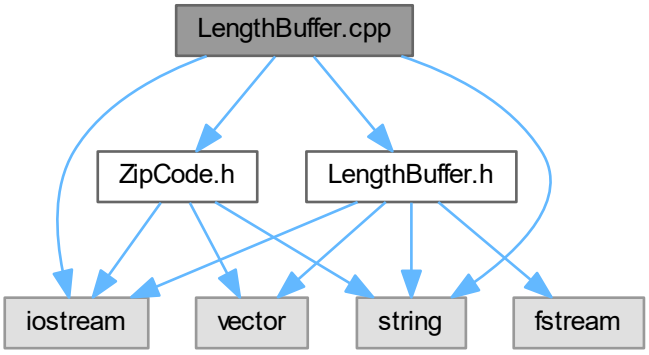
```
00001
00006 #ifndef DELIMBUFFER_h
00007 #define DELIMBUFFER_h
00008
00009 #include <iostream>
00010 #include <fstream>
00011 #include <vector>
00012 #include <string>
00013 using namespace std;
00014
```

```
00018 class delimBuffer {
00019
00020 private:
00021     char delim;
00022     int size;
00023     int max;
00024     int index;
00025     string buffer;
00026
00027 public:
00028
00034     delimBuffer();
00035
00036     delimBuffer(char, int);
00037
00043     bool unpack(string & field);
00044
00049     bool read(ifstream& inFile);
00050
00055     void setBuffer(string x) {
00056         buffer = x; };
00057
00058     string getBuffer() {
00059         return buffer; };
00060
00061 };
00062 #endif
```

4.25 LengthBuffer.cpp File Reference

```
#include "LengthBuffer.h"
#include "ZipCode.h"
#include <iostream>
#include <string>
```

Include dependency graph for LengthBuffer.cpp:



4.26 LengthBuffer.cpp

Go to the documentation of this file.

```
00001
00005 #include "LengthBuffer.h"
00006 #include "ZipCode.h"
00007 #include <iostream>
00008 #include <string>
```

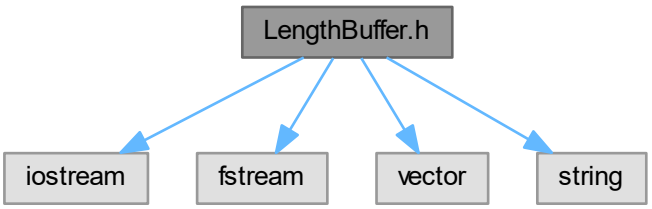
```
00009
00010 using namespace std;
00011 /*
00012  * @brief General constructor for the LengthBuffer.
00013  */
00014 LengthBuffer::LengthBuffer() {
00015     delim = ',';
00016     index = 0;
00017     buffer = "";
00018     size = 0;
00019     max = 1000;
00020 }
00021
00022 /*
00023  * @brief Builds the specified constructor.
00024  * @pre Receives the delimiter and the maximum buffer size.
00025  * @param1 delimiter a character, default is a comma.
00026  * @param2 maxsize an integer, the maximum size of the buffer.
00027  */
00028 LengthBuffer::LengthBuffer(char delim, int max) {
00029     this->delim = delim;
00030     this->index = 0;
00031     this->buffer = "";
00032     this->size = 0;
00033     this->max = max;
00034 }
00035
00036 /*
00037  * @brief Writes to the index file.
00038  * @pre Receives the index file and an instance of the ZipCode class.
00039  * @param1 outFile an ofstream variable containing the index file address.
00040  * @param2 record a ZipCode class variable.
00041  * @post Returns true or false if the file wrote correctly.
00042  */
00043 void LengthBuffer::write(fstream& outFile) {
00044     buffer.insert(0, to_string(buffer.size()));
00045     outFile « buffer;
00046     buffer = "";
00047 }
00048
00049 /*
00050  * @brief Reads from the length-indicated file.
00051  * @pre Receives a file and the offset for a record.
00052  * @param1 inFile an ifstream variable containing the address of the length-indicated file.
00053  * @param2 offset an integer variable containing the offset for the specific record.
00054  */
00055 bool LengthBuffer::read(fstream& inFile, unsigned long offset) {
00056     char temp1, temp2;
00057     int length;
00058     index = 0;
00059     buffer = "";
00060     size = 0;
00061
00062     inFile.seekg(offset);    // seek to start of record
00063
00064     temp1 = inFile.get();   // get first two characters to decode length
00065     temp2 = inFile.get();
00066     buffer.push_back(temp1);   // add characters to buffer
00067     buffer.push_back(temp2);
00068
00069     length = stoi(buffer);     // convert length ascii to int
00070
00071     buffer = "";
00072
00073     int i = 0;
00074     while (i < length) {
00075         inFile.get(temp1);
00076         buffer.push_back(temp1);
00077         size++;
00078         i++;
00079     }
00080
00081     return true;
00082 }
00083
00084 /*
00085  * @brief Unpacks a string of record fields.
00086  * @pre Receives a string of fields.
00087  * @param1 field a string reference of record fields.
00088  * @post Returns true if the string was unpacked; otherwise, it returns false.
00089  */
00090 bool LengthBuffer::unpack(string& field) {
00091     if (index != size && size != 0) {   // execute only when LengthBuffer is not empty
00092
00093         while (index < size && buffer[index] != delim) {
00094             field.push_back(buffer[index++]);
00095         }
```

```
00096         if (buffer[index] == delim)
00097             index++;
00098         return true;
00099
00100     }
00101     return false;
00102 }
00103
00104 void LengthBuffer::pack(string& field) {
00105     buffer.append(field);
00106 }
```

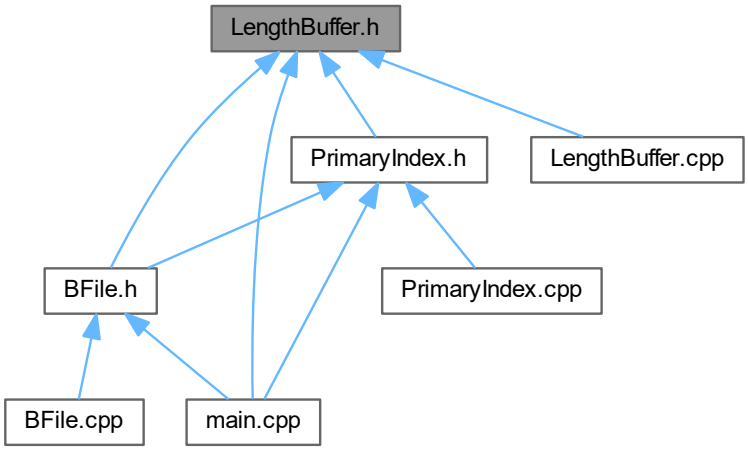
4.27 LengthBuffer.h File Reference

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

Include dependency graph for LengthBuffer.h:



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



Classes

- class LengthBuffer

  Class to store each record and parse each field.

4.28 LengthBuffer.h

Go to the documentation of this file.

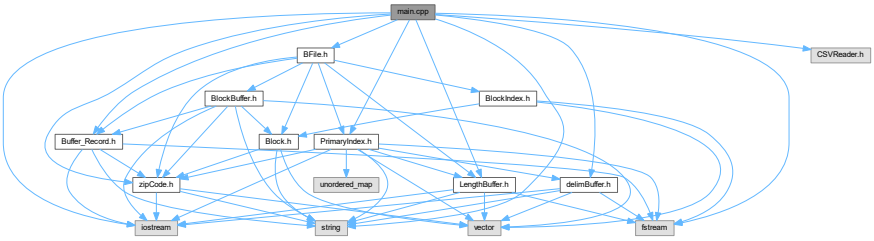
```
00001
00006 #ifndef LENGTHBUFFER_H
00007 #define LENGTHBUFFER_H
00008
00009 #include <iostream>
00010 #include <fstream>
00011 #include <vector>
00012 #include <string>
00013 using namespace std;
00014
00018 class LengthBuffer {
00019
00020 private:
00021     int size;
00022     char delim;
00023     int max;
00024     int index;
00025     string buffer;
00026
00027 public:
00028
00034     LengthBuffer();
00035
00036     LengthBuffer(char delim, int bufferSize);
00037
00038     void write(fstream& outFile);
00039
00044     bool read(fstream& inFile, unsigned long offset);
00045
00046     void pack(string& field);
00047
00053     bool unpack(string& field);
00054
00055     int getSize() {
00056         return buffer.size(); }
00057
00062     string getBuffer() {
00063         return buffer; }
00064
00065 };
00066 #endif
```

4.29 main.cpp File Reference

```
#include "PrimaryIndex.h"
#include "delimBuffer.h"
#include "LengthBuffer.h"
#include "ZipCode.h"
#include "BFile.h"
#include "Buffer_Record.h"
#include "CSVReader.h"
#include <iostream>
#include <fstream>
```

```
#include <string>
```

Include dependency graph for main.cpp:



Functions

- void analyzeCSV (CSVReader &csvReader)
- void addRecord (BFile &bf)
- void delRecord (BFile &bf, const string &arg)
- void handleFileImport (const string &filename)
- void searchDatabase (const PrimaryIndex &indexList)
- void displayRecordFromOffset (ifstream &FS, unsigned long offset)
- int main (int argc, char ∗argv[ ])

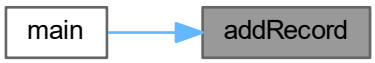
  Main function to process user commands and manage the postal code database.

4.29.1 Function Documentation

4.29.1.1 addRecord()

```
void addRecord (
            BFile & bf )
```

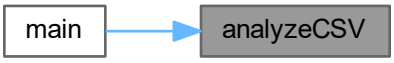
Here is the caller graph for this function:



4.29.1.2 analyzeCSV()

```
void analyzeCSV (
            CSVReader & csvReader )
```

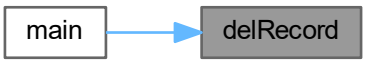
Here is the caller graph for this function:



4.29.1.3 delRecord()

```
void delRecord (
            BFile & bf,
            const string & arg )
```

Here is the caller graph for this function:



4.29.1.4 displayRecordFromOffset()

```
void displayRecordFromOffset (
            ifstream & FS,
            unsigned long offset )
```

4.29.1.5 handleFileImport()

```
void handleFileImport (
            const string & filename )
```

Here is the caller graph for this function:

4.29.1.6 main()

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

Main function to process user commands and manage the postal code database.

Processes command-line arguments for different operations such as physical and logical data dump, record addition, deletion, file importing, and database searching.

Parameters

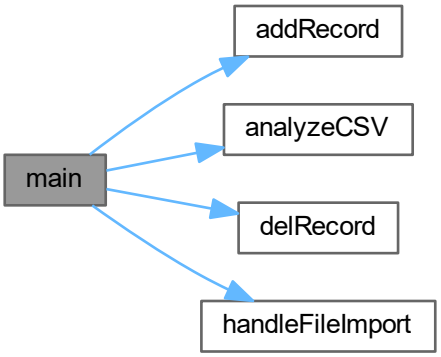
argc	Number of command-line arguments.
argv	Array of command-line arguments.

Returns

int Exit status of the program.

Definition at line 47 of file main.cpp.

Here is the call graph for this function:



4.29.1.7 searchDatabase()

```
void searchDatabase (
            const PrimaryIndex & indexList )
```

4.30 main.cpp

Go to the documentation of this file.

```
00001
00013 //imports
00014
00015 #include "PrimaryIndex.h"
00016 #include "delimBuffer.h"
00017 #include "LengthBuffer.h"
00018 #include "ZipCode.h"
00019 #include "BFile.h"
00020 #include "Buffer_Record.h"
00021 #include "CSVReader.h"
00022 #include <iostream>
00023 #include <fstream>
00024 #include <string>
00025
00026 using namespace std;
00027
00028 // Declarations for helper functions
00029 void analyzeCSV(CSVReader &csvReader);
00030 void addRecord(BFile& bf);
00031 void delRecord(BFile& bf, const string& arg);
00032 void handleFileImport(const string& filename);
00033 void searchDatabase(const PrimaryIndex& indexList);
00034 void displayRecordFromOffset(ifstream& FS, unsigned long offset);
00035
00046 // main to process user commands and manage the postal code database.
00047 int main(int argc, char* argv[]) {
00048     // Process CSV file at the beginning
00049     std::string csvFile = "us_postal_codes.csv";
00050     std::cout « "Processing us_postal_codes.csv.\n" « std::endl;
00051     CSVReader csvReader(csvFile);
00052     analyzeCSV(csvReader);
00053
00054     // Process command-line arguments
00055     if (argc < 2) {
00056         cout « "Usage: " « argv[0] « " -option [additional arguments]" « endl;
00057         return 1;
00058     }
00059
00060     string option = argv[1];
00061     bFile bf;
00062
00063     if (option == "-pd") {
00064         cout « bf.physicalDump();  // Updated method name
00065     } else if (option == "-ld") {
00066         cout « bf.logicalDump();  // Updated method name
00067     } else if (option == "-a") {
00068         addRecord(bf);  // Updated function call
00069     } else if (option == "-d" && argc == 3) {
00070         delRecord(bf, argv[2]);  // Updated function call
00071     } else if (option == "-r" && argc == 3) {
00072         handleFileImport(argv[2]);  // Unchanged
00073     } else if (option == "-z" && argc == 3) {
00074         PrimaryIndex indexList("primIndex.index", "data.txt);
00075         ifstream FS("data.txt");
00076         unsigned long offset = indexList.search(stoi(argv[2]));
00077         displayRecordFromOffset(FS, offset);
00078     } else {
00079         cout « "Invalid arguments" « endl;
00080         return -1;
00081     }
00082
00083     return 0;
00084 }
00092 //add record
00093 void addRecord(BFile& b) {
00094     ZipCode address;  // Correct class name
00095     string temporary;
00096     float lat, lon;
00097     int zip;
00098
00099     cout « "Zip Code: ";
```

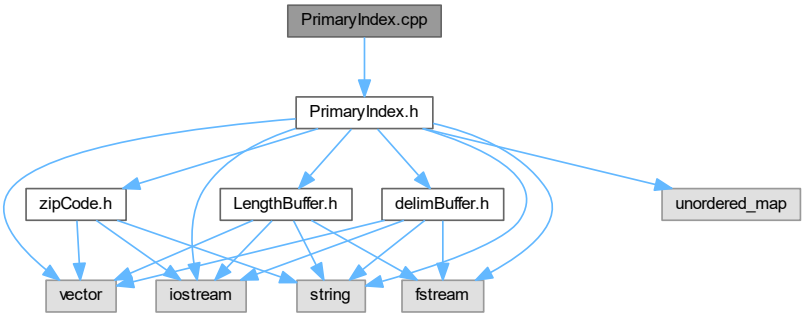
```
00100     cin » temporary;
00101     zip = stoi(temporary);  // Convert string to int
00102     address.setNum(zip);
00103
00104     cout « "City: ";
00105     cin » temporary;
00106     address.setCity(temporary);
00107
00108     cout « "State Code: ";
00109     cin » temporary;
00110     address.setStateCode(temporary);
00111
00112     cout « "County: ";
00113     cin » temporary;
00114     address.setCounty(temporary);
00115
00116     cout « "Latitude: ";
00117     cin » lat;  // Directly read as float
00118     address.setLat(lat);
00119
00120     cout « "Longitude: ";
00121     cin » lon;  // Directly read as float
00122     address.setLon(lon);
00123
00124     if (b.addRecord(address))  // Correct object name
00125         cout « "Record added\n";
00126     else
00127         cout « "Record not added\n";
00128 }
00129
00137 void delRecord(blockFile& b, const string& arg) {
00138     if (b.delRecord(arg))
00139         cout « "Record deleted \n";
00140     else
00141         cout « "Failed to delete \n";
00142 }
00143
00149 void handleFileImport(const string& filename) {
00150     ifstream inFile(filename);
00151     primaryIndex indexList(inFile);
00152     cout « "File imported successfully" « endl;
00153     cout « "Do you want to search the database? (Y/N): ";
00154     char response;
00155     cin » response;
00156     if (tolower(response) == 'y') {
00157         searchDatabase(indexList);
00158     }
00159 }
00160
00166 //Searches the database for a specific address and displays
00167 void searchDatabase(const PrimaryIndex& indexList) {
00168     int valid_zip;
00169     cout « "Please enter a valid zip: ";
00170     cin » valid_zip;
00171     unsigned long offset = indexList.search(valid_zip;);
00172     if (offset == 0) {
00173         cout « "cant find zip" « endl;
00174         return;
00175     }
00176     ifstream dFile("data.txt");
00177     displayRecordFromOffset(FS, offset);
00178 }
00179
00186 //display @offset
00187 void displayRecordFromOffset(ifstream& FS, unsigned long offset) {
00188     LengthBuffer showing_addr;
00189     indicated.read(FS, offset);
00190     for (int i = 0; i < 6; ++i) {
00191         string temp;
00192         showing_addr.unpack(temp);
00193         cout « (i == 0 ? "Zip Code: " : i == 1 ? "Place Name: " : i == 2 ? "State: " : i == 3 ?
      "County: " : i == 4 ? "Lat: " : "Long: ") « temp « endl;
00194     }
00195 }
00202 //Analyzes and displays state statistics from a CSVReader object.
00203 void analyzeCSV(CSVReader &csvReader) {
00204     //CSVReader csvReader(fileName);
00205     if (!csvReader.isOpen()) {
00206         std::cerr « "Failed to open CSV file." « std::endl;
00207         return;
00208     }
00209     // Read and process the CSV file.
00210     csvReader.ReadFile();
00211
00212
00213     // Close the CSV file.
00214     csvReader.close();
```

```
00215 }
```

4.31 PrimaryIndex.cpp File Reference

```
#include "PrimaryIndex.h"
```

Include dependency graph for PrimaryIndex.cpp:



Variables

- static const short NumStates = 57

4.31.1 Variable Documentation

4.31.1.1 NumStates

```
const short NumStates = 57  [static]
```

Definition at line 10 of file PrimaryIndex.cpp.

4.32 PrimaryIndex.cpp

Go to the documentation of this file.

```
00001
00006 #include "PrimaryIndex.h"
00007
00008 using namespace std;
00009
00010 static const short NumStates = 57; // Number of possible states/regions
00011
00012 void PrimaryIndex::getIndex(vector<IndexElement>& returnValue) {
00013     IndexElement temp;
00014
00015     int i = 0;
00016 while (i < index.size()) {
00017     temp.zip = index[i].zip;
00018     temp.offset = index[i].offset;
00019     returnValue.push_back(temp);
00020     i++;
00021 }
```

```
00022
00023 }
00024
00025 void PrimaryIndex::add(int zipCode, unsigned long offset) {
00026     IndexElement temp = {zipCode, offset};
00027     int i = 0;
00028
00029     if (recordCount == 0) {
00030         index.push_back(temp);
00031     } else {
00032         while (i < recordCount && index[i].zip < zipCode) {
00033             i++;
00034         }
00035         if (i == recordCount) {
00036             index.push_back(temp);
00037         } else if (i == 0) {
00038             index.insert(index.begin(), temp);
00039         } else {
00040             vector<IndexElement>::iterator it;
00041             it = index.begin();
00042             index.insert(it + i, temp);
00043         }
00044     }
00045
00046     recordCount++;
00047 }
00048
00049 unsigned long PrimaryIndex::search(int targetZipCode) {
00050     int offset = binarySearch(targetZipCode, 0, recordCount - 1);
00051     return offset;
00052 }
00053
00054 unsigned long PrimaryIndex::binarySearch(int target, int left, int right) {
00055     if (left > right) {
00056         return 0;
00057     }
00058
00059     if (right >= left) {
00060         int mid = (left + right) / 2;
00061
00062         if (index[mid].zip == target) {
00063             cout « "Target hit!: " « index[mid].zip « ' ' « index[mid].offset « endl;
00064             return index[mid].offset;
00065         } else if (index[mid].zip > target) {
00066             binarySearch(target, left, mid - 1);
00067         } else {
00068             binarySearch(target, mid + 1, right);
00069         }
00070     }
00071 }
00072
00073 void PrimaryIndex::readIndex() {
00074     if (!indexFile.eof()) {
00075         int itemp;
00076         indexFile » itemp;
00077         cout « itemp « " records in the file." « endl;
00078
00079         int zip;
00080         unsigned long int offset;
00081         char temp;
00082
00083         while (!indexFile.eof()) {
00084             indexFile » zip » temp » offset;
00085             add(zip, offset);
00086         }
00087     }
00088 }
00089
00090 void PrimaryIndex::writeToFile() {
00091     ofstream outFile;
00092     outFile.open("IndexFile.index");
00093
00094     outFile « recordCount « "\n";
00095
00096     for (int i = 0; i < recordCount; i++) {
00097         outFile « index[i].zip « "," « index[i].offset « "\n";
00098     }
00099 }
00100
00101 void PrimaryIndex::readCSV(ifstream& infile) {
00102     indexFile.open("IndexFile.index");
00103     dataFile.open("DataFile.licsv");
00104
00105     vector<vector<ZipCode» states;
00106     states.resize(NumStates);
00107     string headerData = readIn(infile, states);
00108     cout « endl « printTable(states) « endl;
```

```
00109
00110     transfer(states, headerData);
00111
00112     writeToFile();
00113 }
00114
00115 string PrimaryIndex::buildHeader(string headerData) {
00116     string record;
00117     int count = 1, temp;
00118
00119     record.append("Structure Type: Length Indicated Comma Separated File\n");
00120     record.append("Version: 1.0\n");
00121
00122     // Determine size of record
00123     record.append("Record Size: ");
00124     record.append(to_string(headerData.size()));
00125     record.push_back('\n');
00126
00127     // Size format type
00128     record.append("Size Format: 2-digit ASCII\n");
00129
00130     // Index File Name
00131     record.append("Index File: IndexFile.index\n");
00132
00133     // Index File Schema Information
00134     record.append("Index File Schema: Listed by zip code then corresponding offset, sorted by zip
      code\n");
00135
00136     // Record Count
00137     record.append("Record Count: 40933\n");
00138
00139     // Count of fields per record
00140     for (int j = 0; j < headerData.size(); j++) {
00141         if (headerData[j] == ',') {
00142             count++;
00143         }
00144     }
00145
00146     record.append("Fields per Record: ");
00147     record.append(to_string(count));
00148     record.push_back('\n');
00149
00150     // Name of each field
00151     record.append("Name of Fields: ");
00152     record.append(headerData);
00153     record.push_back('\n');
00154
00155     // Type Schema
00156     record.append("Type Schema: Zip Code is an integer. Lon and Lat are floating point decimals.
      County, State Code, and City are strings.\n");
00157
00158     // Indicate which field is primary key
00159     record.append("First Key: Zip Code\n");
00160     return record;
00161 }
00162
00163 void PrimaryIndex::transfer(vector<vector<ZipCode»& states, string headerData) {
00164     if (!dataFile.is_open()) {
00165         dataFile.open("DataFile.licsv");
00166     }
00167
00168     string header = buildHeader(headerData);
00169     dataFile « header;
00170
00171     string temp;
00172     LengthBuffer buf;
00173     unsigned long count = 0;
00174     unsigned long offsetSum = header.size() + 11;
00175
00176     for (int i = 0; i < NumStates; i++) {
00177         for (int j = 0; j < states[i].size(); j++) {
00178             count = 0;
00179
00180             temp = to_string(states[i][j].getNum());
00181             temp.push_back(',');
00182             temp.append(states[i][j].getCity());
00183             temp.push_back(',');
00184             temp.append(states[i][j].getStateCode());
00185             temp.push_back(',');
00186             temp.append(states[i][j].getCounty());
00187             temp.push_back(',');
00188             temp.append(to_string(states[i][j].getLat()));
00189             temp.push_back(',');
00190             temp.append(to_string(states[i][j].getLon()));
00191
00192             count = temp.size();
00193
```

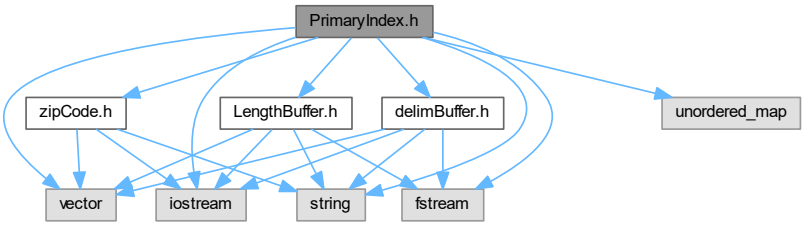
```
00194             buf.pack(temp);
00195             buf.write(dataFile);
00196 
00197             add(states[i][j].getNum(), offsetSum);
00198             offsetSum += count + 2;
00199         }
00200     }
00201 }
00202 
00203 string PrimaryIndex::readIn(ifstream& inFile, vector<vector<ZipCode>>& states) {
00204     ZipCode temp;
00205     string item;
00206     string headerData;
00207     vector<string> headerElement;
00208     delimBuffer b;
00209 
00210     for (int i = 0; i < 3; ++i) {
00211         b.read(inFile);
00212         headerData.append(b.getBuffer());
00213     }
00214 
00215     for (int i = 0; i < headerData.size(); i++) {
00216         headerData[i] = tolower(headerData[i]);
00217     }
00218     for (int i = 0, j = 0, k = 0; i < headerData.size(); ++i, k = 0) {
00219         string temps = "";
00220         while (headerData[i] != ',' && i < headerData.size()) {
00221             if (isalpha(headerData[i])) {
00222                 temps.push_back(headerData[i]);
00223             }
00224             i++;
00225         }
00226 
00227         headerElement.resize(headerElement.size() + 1);
00228         headerElement[j++] = temps;
00229     }
00230 
00231     int tag = 0;
00232 
00233     while (b.read(inFile)) {
00234         while (b.unpack(item)) {
00235             if (tag > 5) tag = 0;
00236 
00237             if (headerElement[tag] == "zipcode") {
00238                 temp.setNum(stoi(item));
00239             } else if (headerElement[tag] == "placename") {
00240                 temp.setCity(item);
00241             } else if (headerElement[tag] == "state") {
00242                 temp.setStateCode(item);
00243             } else if (headerElement[tag] == "county") {
00244                 temp.setCounty(item);
00245             } else if (headerElement[tag] == "lat") {
00246                 temp.setLat(stof(item));
00247             } else if (headerElement[tag] == "long") {
00248                 temp.setLon(stof(item));
00249             }
00250             ++tag;
00251             item = "";
00252         }
00253         states[stateSelector(temp.getStateCode())].push_back(ZipCode(temp));
00254     }
00255 
00256     return headerData;
00257 }
00258 
00259 string PrimaryIndex::printTable(vector<vector<ZipCode>>& states) {
00260     string output;
00261 
00262     output.append("*************************************************\n");
00263     output.append("*State\t|East\t\t|West\t\t|North\t\t|South\t*\n");
00264     output.append("*************************************************\n");
00265 
00266     for (int i = 0; i < NumStates; i++) {
00267         output.append("*");
00268         output.append(states[i][0].getStateCode());
00269         output.append("\t|");
00270         output.append(to_string(states[i][eastest(states[i])].getNum()));
00271         output.append("\t\t|");
00272         output.append(to_string(states[i][westest(states[i])].getNum()));
00273         output.append("\t\t|");
00274         output.append(to_string(states[i][northest(states[i])].getNum()));
00275         output.append("\t\t|");
00276         output.append(to_string(states[i][southest(states[i])].getNum()));
00277         output.append("\t*\n");
00278     }
00279     output.append("*************************************************\n");
00280     output.append("*State\t|East\t\t|West\t\t|North\t\t|South\t*\n");
```

```
00281     output.append("*************************************************");
00282
00283     return output;
00284 }
00285
00286 short PrimaryIndex::northest(vector<ZipCode> state) {
00287     short x = 0;
00288     for (int i = 1; i < state.size(); i++) {
00289         if (state[i].getLat() > state[i - 1].getLat()) {
00290             x = i;
00291         }
00292     }
00293     return x;
00294 }
00295
00296 short PrimaryIndex::southest(vector<ZipCode> state) {
00297     short x = 0;
00298     for (int i = 1; i < state.size(); i++) {
00299         if (state[i].getLat() < state[i - 1].getLat()) {
00300             x = i;
00301         }
00302     }
00303     return x;
00304 }
00305
00306 short PrimaryIndex::eastest(vector<ZipCode> state) {
00307     short x = 0;
00308     for (int i = 1; i < state.size(); i++) {
00309         if (state[i].getLon() < state[i - 1].getLon()) {
00310             x = i;
00311         }
00312     }
00313     return x;
00314 }
00315
00316 short PrimaryIndex::westest(vector<ZipCode> state) {
00317     short x = 0;
00318     for (int i = 1; i < state.size(); i++) {
00319         if (state[i].getLon() > state[i - 1].getLon())
00320             x = i;
00321     }
00322     return x;
00323 }
00324
00331 short PrimaryIndex::stateSelector(const std::string& stateCode) {
00332     static const std::unordered_map<std::string, short> stateMap = {
00333         {"AA", 0}, {"AK", 1}, {"AL", 2}, {"AP", 3}, {"AR", 4},
00334         {"AZ", 5}, {"CA", 6}, {"CO", 7}, {"CT", 8}, {"DC", 9},
00335         {"DE", 10}, {"FL", 11}, {"FM", 12}, {"GA", 13}, {"HI", 14},
00336         {"IA", 15}, {"ID", 16}, {"IL", 17}, {"IN", 18}, {"KS", 19},
00337         {"KY", 20}, {"LA", 21}, {"MA", 22}, {"MD", 23}, {"ME", 24},
00338         {"MH", 25}, {"MI", 26}, {"MN", 27}, {"MO", 28}, {"MP", 29},
00339         {"MS", 30}, {"MT", 31}, {"NC", 32}, {"ND", 33}, {"NE", 34},
00340         {"NH", 35}, {"NJ", 36}, {"NM", 37}, {"NV", 38}, {"NY", 39},
00341         {"OH", 40}, {"OK", 41}, {"OR", 42}, {"PA", 43}, {"PW", 44},
00342         {"RI", 45}, {"SC", 46}, {"SD", 47}, {"TN", 48}, {"TX", 49},
00343         {"UT", 50}, {"VA", 51}, {"VT", 52}, {"WA", 53}, {"WI", 54},
00344         {"WV", 55}, {"WY", 56}
00345     };
00346
00347     auto it = stateMap.find(stateCode);
00348     if (it != stateMap.end()) {
00349         return it->second;
00350     }
00351     else {
00352         return -1;
00353     }
00354 }
```

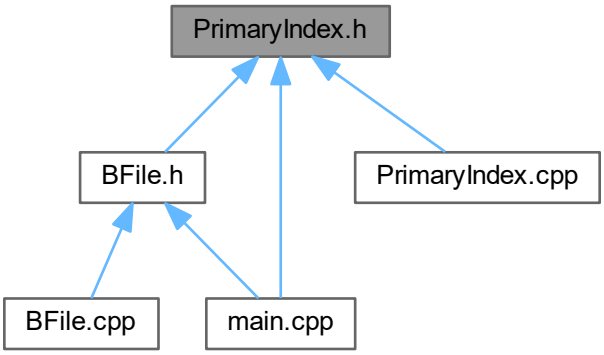
4.33 PrimaryIndex.h File Reference

```
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
#include <unordered_map>
#include "LengthBuffer.h"
#include "zipCode.h"
```

`#include "delimBuffer.h"`
Include dependency graph for PrimaryIndex.h:



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



Classes

- struct IndexElement
- class PrimaryIndex

4.34 PrimaryIndex.h

Go to the documentation of this file.

```
00001
00007 #ifndef PRIMARYINDEX_H
00008 #define PRIMARYINDEX_H
00009
00010 #include <vector>
00011 #include <iostream>
00012 #include <fstream>
00013 #include <string>
00014 #include <unordered_map>
00015 #include "LengthBuffer.h"
00016 #include "zipCode.h"
00017 #include "delimBuffer.h"
```

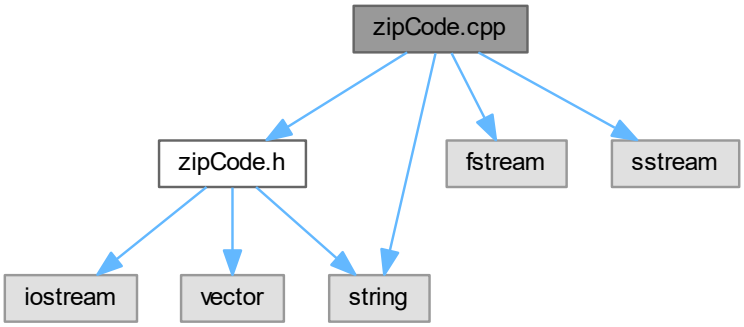
```
00018
00019 struct IndexElement {
00020
00021     int zip;
00022     unsigned long int offset;
00023 };
00024
00025 class PrimaryIndex {
00026
00027 private:
00028
00029     string printTable(vector<vector<ZipCode»&); // output data table
00030
00031     short stateSelector(const string& stateCode);    // return index of state with the given 2-letter
      code
00032
00033     short northest(vector<ZipCode>); // searches a given state to find the most northern zip code
00034
00035     short southest(vector<ZipCode>); // searches a given state to find the most southern zip code
00036
00037     short eastest(vector<ZipCode>); // searches a given state to find the most eastern zip code
00038
00039     short westest(vector<ZipCode>); // searches a given state to find the most western zip code
00040
00041     string readIn(ifstream& inFile, vector<vector<ZipCode»& states);
00042
00043     unsigned long binarySearch(int target, int left, int right);
00044
00045     void transfer(vector<vector<ZipCode»&, string);
00046
00047     string buildHeader(string);
00048
00049     vector<IndexElement> index;
00050     int recordCount;
00051     fstream dataFile, indexFile;
00052
00053 public:
00054
00055     PrimaryIndex(string indexFileName = "IndexFile.index", string dataFileName = "DataFile.licsv") {
00056         indexFile.open(indexFileName); dataFile.open(dataFileName); readIndex(); indexFile.close();
      dataFile.close(); }
00057
00058     PrimaryIndex(ifstream& infile) {
00059         readCSV(infile); }
00060
00061     void add(int zipCode, unsigned long offset);
00062
00063     unsigned long search(int targetZipCode);
00064
00065     void writeToFile();
00066
00067     void readIndex();
00068
00069     void readCSV(ifstream&);
00070
00071     void getIndex(vector<IndexElement>& returnValue);
00072
00073     int getSize() {
00074         return index.size(); }
00075
00076     int getOffset(int i) { return index[i].offset; };
00077
00078 };
00079
00080 #endif
```

4.35 zipCode.cpp File Reference

Implementation of member functions for the ZipCode class.

```
#include "zipCode.h"
#include <fstream>
#include <sstream>
#include <string>
```

Include dependency graph for zipCode.cpp:



4.35.1 Detailed Description

Implementation of member functions for the ZipCode class.

Author

Group7

Definition in file zipCode.cpp.

4.36 zipCode.cpp

Go to the documentation of this file.

```
00001
00007 #include "zipCode.h"
00008 #include <fstream>
00009 #include <sstream>
00010 #include <string>
00011
00012 using namespace std;
00013
00014 // Default Constructor
00015 ZipCode::ZipCode() {
00016     // @brief Default Constructor
00017     // @post Initializes a ZipCode object with default (empty or zero) values.
00018     num = -1;
00019     city = "";
00020     stateCode = "";
00021     county = "";
00022     lat = 0;
00023     lon = 0;
00024 }
00025
00026 // Parameterized Constructor
00027 ZipCode::ZipCode(int newNum, string newCity, string newStateCode, string newCounty, float newLat,
    float newLon) {
00028     // @brief Parameterized Constructor
00029     // @pre Accepts individual parameters for each member variable.
00030     // @post Initializes a ZipCode object with provided values.
00031     num = newNum;
00032     city = newCity;
00033     stateCode = newStateCode;
00034     county = newCounty;
```

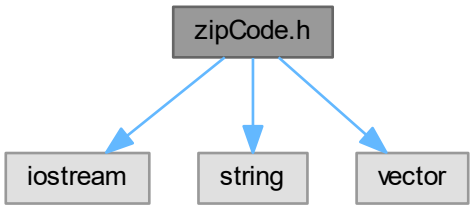
```
00035     lat = newLat;
00036     lon = newLon;
00037 }
00038
00039 // Copy Constructor
00040 ZipCode::ZipCode(const ZipCode& oldZip) {
00041     // @brief Copy Constructor
00042     // @pre Accepts an existing ZipCode object.
00043     // @post Creates a new ZipCode object as a copy of the existing one.
00044     num = oldZip.num;
00045     city = oldZip.city;
00046     stateCode = oldZip.stateCode;
00047     county = oldZip.county;
00048     lat = oldZip.lat;
00049     lon = oldZip.lon;
00050 }
00051
00052 // Function to get the size of the ZipCode data
00053 const int ZipCode::getSize() {
00054     // @brief Gets the size of the ZipCode data.
00055     // @return The size of the ZipCode data as an integer.
00056     string size = "";
00057     string temp = "";
00058
00059     temp.append(to_string(num));
00060     size.push_back(',');
00061     temp.append(city);
00062     size.push_back(',');
00063     temp.append(stateCode);
00064     size.push_back(',');
00065     temp.append(county);
00066     size.push_back(',');
00067     temp.append(to_string(lat));
00068     size.push_back(',');
00069     temp.append(to_string(lon));
00070
00071     size.append(to_string(temp.size()));
00072     size.push_back(',');
00073     size.append(temp);
00074
00075     return size.size();
00076 }
00077
00078 // Static method to read zip codes from a file
00079 vector<ZipCode> ZipCode::readFromFile(const string& filename) {
00080     // @brief Reads zip codes from a file and creates ZipCode objects.
00081     // @pre Expects a filename as a string.
00082     // @return A vector of ZipCode objects.
00083     vector<ZipCode> zipCodes;
00084     ifstream file(filename);
00085     string line;
00086
00087     while (getline(file, line)) {
00088         stringstream ss(line);
00089         int num;
00090         string city, stateCode, county;
00091         float lat, lon;
00092
00093         ss » num;
00094         getline(ss, city, ',');
00095         getline(ss, stateCode, ',');
00096         getline(ss, county, ',');
00097         ss » lat » lon;
00098
00099         ZipCode zip(num, city, stateCode, county, lat, lon);
00100         zipCodes.push_back(zip);
00101     }
00102
00103     return zipCodes;
00104 }
```

4.37 zipCode.h File Reference

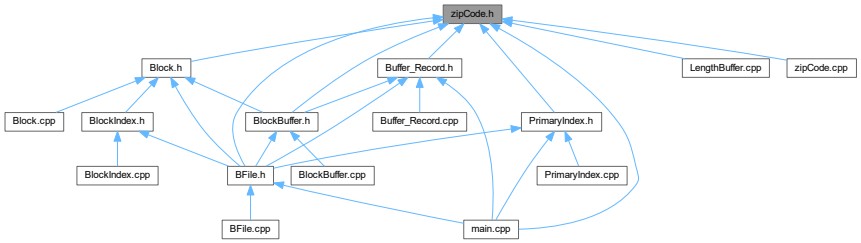
Class representing a zip code along with city name, state code, county, latitude, and longitude for each postal code.

```
#include <iostream>
#include <string>
```

`#include <vector>`
Include dependency graph for zipCode.h:



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



Classes

- class ZipCode

4.37.1 Detailed Description

Class representing a zip code along with city name, state code, county, latitude, and longitude for each postal code.

Definition in file zipCode.h.

4.38 zipCode.h

Go to the documentation of this file.

```
00001
00006 #ifndef ZIP_CODE
00007 #define ZIP_CODE
00008
00009 #include <iostream>
00010 #include <string>
00011 #include <vector>
00012 using namespace std;
00013
00014 class ZipCode {
00015 public:
```

```
00016     // Default constructor
00017     // @brief Initializes a ZipCode object with empty values.
00018     ZipCode();
00019
00020     // Parameterized constructor
00021     // @brief Initializes a ZipCode object with specific values.
00022     // @pre Requires zip code, city, state code, county, latitude, and longitude.
00023     // @post ZipCode object initialized with given values.
00024     ZipCode(int newNum, string newCity, string newStateCode, string newCounty, float newLat, float
      newLon);
00025
00026     // Copy constructor
00027     // @brief Initializes a ZipCode object as a copy of another ZipCode object.
00028     ZipCode(const ZipCode& oldZip);
00029
00030     // Setters and Getters
00031     // @brief Set and get methods for ZipCode properties.
00032     void setNum(int newNum);
00033     const int getNum();
00034     void setCity(string newCity);
00035     const string getCity();
00036     void setStateCode(string newStateCode);
00037     const string getStateCode();
00038     void setCounty(string newCounty);
00039     const string getCounty();
00040     void setLat(float newLat);
00041     const float getLat();
00042     void setLon(float newLon);
00043     const float getLon();
00044
00045     // Method to get the size of the ZipCode data
00046     // @brief Gets the size of the ZipCode data.
00047     // @return The size of the ZipCode data as an integer.
00048     const int getSize();
00049
00050     // Method to print the ZipCode information
00051     // @brief Prints the details of the ZipCode object.
00052     void print();
00053
00054     // Static method for reading zip codes from a file
00055     // @brief Reads zip codes from a file and returns them as a vector of ZipCode objects.
00056     // @pre Requires a filename as a string.
00057     // @return A vector of ZipCode objects.
00058     static vector<ZipCode> readFromFile(const string& filename);
00059
00060 private:
00061     int num;
00062     float lat;
00063     float lon;
00064     string stateCode;
00065     string city;
00066     string county;
00067 };
00068
00069 #endif // ZIP_CODE
```

Index