Project 3

Generated by Doxygen 1.12.0

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

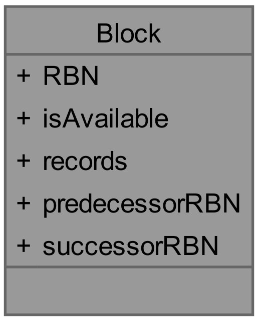
Class Documentation

3.1 Block Struct Reference

Represents a single block in the blocked sequence set.

`#include <Block.h>`

Collaboration diagram for Block:



Public Attributes

- int RBN

  Relative Block Number (unique identifier for the block)
- bool isAvailable

  Flag indicating whether the block is available.
- std::vector< std::string > records

  Records stored in the block.
- int predecessorRBN

  RBN of the predecessor block in the chain.
- int successorRBN

  RBN of the successor block in the chain.

3.1.1 Detailed Description

Represents a single block in the blocked sequence set.

A block can either be part of the active list or the available list. It contains metadata such as predecessor and successor links and a list of records. Each block is uniquely identified by a Relative Block Number (RBN).

Definition at line 27 of file Block.h.

3.1.2 Member Data Documentation

3.1.2.1 isAvailable

```
bool Block::isAvailable
```

Flag indicating whether the block is available.

Definition at line 29 of file Block.h.

3.1.2.2 predecessorRBN

```
int Block::predecessorRBN
```

RBN of the predecessor block in the chain.

Definition at line 31 of file Block.h.

3.1.2.3 RBN

```
int Block::RBN
```

Relative Block Number (unique identifier for the block)

Definition at line 28 of file Block.h.

3.1.2.4 records

```
std::vector<std::string> Block::records
```

Records stored in the block.

Definition at line 30 of file Block.h.

3.1.2.5 successorRBN

```
int Block::successorRBN
```

RBN of the successor block in the chain.

Definition at line 32 of file Block.h.

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

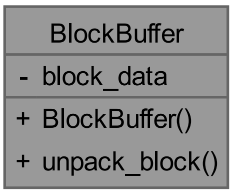
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.h

3.2 BlockBuffer Class Reference

A class to manage and process blocks of data.

```
#include <Buffer.h>
```

Collaboration diagram for BlockBuffer:



Public Member Functions

- BlockBuffer (const std::unordered_map< std::string, ZipCodeRecord > &block)
  A buffer class to manage individual blocks of data.
- std::vector< ZipCodeRecord > unpack_block () const
  Unpacks a block into a vector of records.

Private Attributes

- std::unordered_map< std::string, ZipCodeRecord > block_data

3.2.1 Detailed Description

A class to manage and process blocks of data.

Definition at line 25 of file Buffer.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 BlockBuffer()

```
BlockBuffer::BlockBuffer (
            const std::unordered_map< std::string, ZipCodeRecord > & block)  [explicit]
```

A buffer class to manage individual blocks of data.

Definition at line 14 of file Buffer.cpp.

3.2.3 Member Function Documentation

3.2.3.1 unpack_block()

```
std::vector< ZipCodeRecord > BlockBuffer::unpack_block () const
```

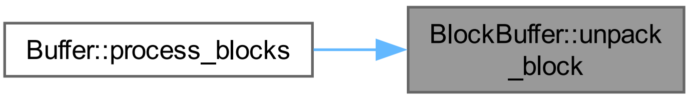
Unpacks a block into a vector of records.

Returns

A vector of ZipCodeRecords contained in the block.

Definition at line 17 of file Buffer.cpp.

Here is the caller graph for this function:



3.2.4 Member Data Documentation

3.2.4.1 block_data

```
std::unordered_map<std::string, ZipCodeRecord> BlockBuffer::block_data  [private]
```

Definition at line 36 of file Buffer.h.

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

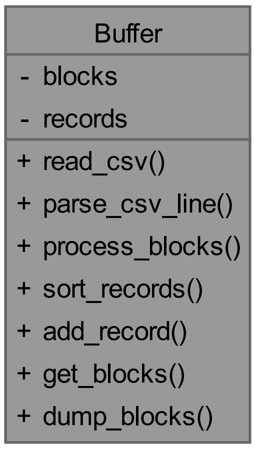
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.h
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.cpp

3.3 Buffer Class Reference

A buffer class to manage ZipCodeRecords and process blocks of data.

```
#include <Buffer.h>
```

Collaboration diagram for Buffer:



Public Member Functions

- bool read_csv (const std::string &csv_filename, size_t records_per_block)

  Reads a CSV file and stores the records in the buffer.
- ZipCodeRecord parse_csv_line (const std::string &line) const

  Parses a single line from the CSV file into a ZipCodeRecord.
- void process_blocks ()

  Processes the buffer block-by-block, unpacking records and fields.
- void sort_records ()

  Sorts all records in the buffer by zip code.
- void add_record (size_t block_number, const ZipCodeRecord &record)

  Adds a ZipCodeRecord to a specific block and the main records list.
- std::unordered_map< size_t, std::unordered_map< std::string, ZipCodeRecord > > get_blocks () const

  Retrieves all blocks of ZipCodeRecords.
- void dump_blocks () const

  Prints the contents of each block for debugging purposes.

Private Attributes

- std::unordered_map< size_t, std::unordered_map< std::string, ZipCodeRecord > > blocks
- std::vector< ZipCodeRecord > records

3.3.1 Detailed Description

A buffer class to manage ZipCodeRecords and process blocks of data.

Definition at line 68 of file Buffer.h.

3.3.2 Member Function Documentation

3.3.2.1 add_record()

```
void Buffer::add_record (
             size_t block_number,
             const ZipCodeRecord & record)
```

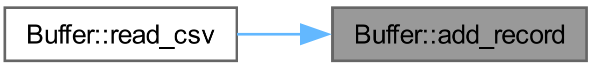
Adds a ZipCodeRecord to a specific block and the main records list.

Parameters

block_number	The block number to which the record should be added.
record	The ZipCodeRecord to be added.

Definition at line 157 of file Buffer.cpp.

Here is the caller graph for this function:



3.3.2.2 dump_blocks()

```
void Buffer::dump_blocks () const
```

Prints the contents of each block for debugging purposes.

Definition at line 176 of file Buffer.cpp.

3.3.2.3 get_blocks()

```
std::unordered_map< size_t, std::unordered_map< std::string, ZipCodeRecord > > Buffer::get_↩
blocks () const
```

Retrieves all blocks of ZipCodeRecords.

Returns

A map where the key is the block number, and the value is a map of ZipCodeRecords.

std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord>> A map where the key is the block number and the value is a map of ZipCodeRecords within that block.

Definition at line 169 of file Buffer.cpp.

3.3.2.4 parse_csv_line()

```
ZipCodeRecord Buffer::parse_csv_line (
             const std::string & line) const
```

Parses a single line from the CSV file into a ZipCodeRecord.

Parameters

line	A string containing a single CSV line.

Returns

A parsed ZipCodeRecord object.

Parameters

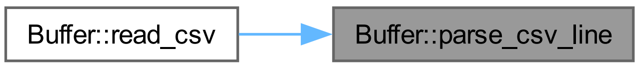
line	A string containing a single CSV line.

Returns

ZipCodeRecord The parsed ZipCodeRecord.

Definition at line 93 of file Buffer.cpp.

Here is the caller graph for this function:



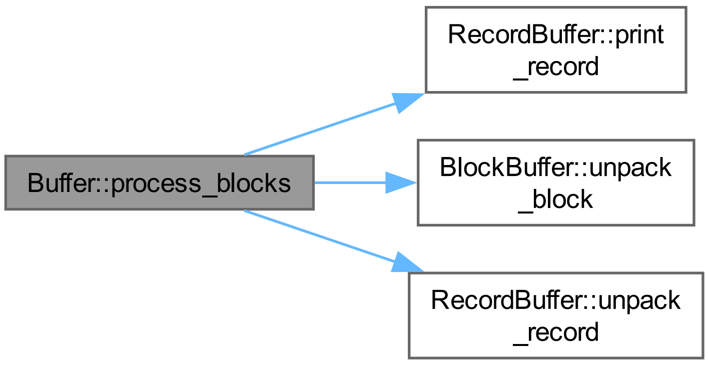
3.3.2.5 process_blocks()

```
void Buffer::process_blocks ()
```

Processes the buffer block-by-block, unpacking records and fields.

Definition at line 112 of file Buffer.cpp.

Here is the call graph for this function:


RecordBuffer::print_record
Buffer::process_blocks
BlockBuffer::unpack_block
RecordBuffer::unpack_record

3.3.2.6 read_csv()

```
bool Buffer::read_csv (
            const std::string & csv_filename,
            size_t records_per_block)
```

Reads a CSV file and stores the records in the buffer.

Parameters

csv_filename	The name of the CSV file to read.
records_per_block	The maximum number of records per block.

Returns

True if the CSV file was successfully read, false otherwise.

Parameters

csv_filename	The name of the CSV file to read.
records_per_block	The maximum number of records per block.

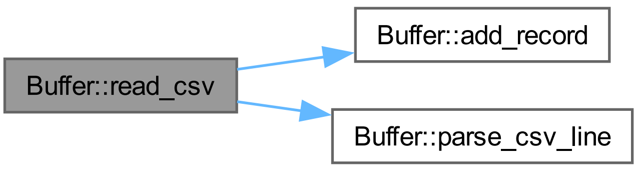
Returns

true If the CSV file was successfully read and processed.

false If the file could not be opened or read.

Definition at line 59 of file Buffer.cpp.

Here is the call graph for this function:



3.3.2.7 sort_records()

```
void Buffer::sort_records ()
```

Sorts all records in the buffer by zip code.

Definition at line 132 of file Buffer.cpp.

3.3.3 Member Data Documentation

3.3.3.1 blocks

```
std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord> > Buffer::blocks
[private]
```

Definition at line 115 of file Buffer.h.

3.3.3.2 records

```
std::vector<ZipCodeRecord> Buffer::records  [private]
```

Definition at line 118 of file Buffer.h.

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

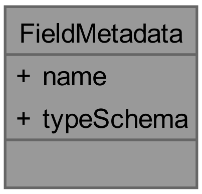
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.h
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.cpp

3.4 FieldMetadata Struct Reference

Metadata structure for field information in the header.

```
#include <HeaderRecord.h>
```

Collaboration diagram for FieldMetadata:



Public Attributes

- std::string name
  Name or ID of the field.
- std::string typeSchema
  Type and format information for the field.

3.4.1 Detailed Description

Metadata structure for field information in the header.

Definition at line 10 of file HeaderRecord.h.

3.4.2 Member Data Documentation

3.4.2.1 name

```
std::string FieldMetadata::name
```

Name or ID of the field.

Definition at line 11 of file HeaderRecord.h.

3.4.2.2 typeSchema

```
std::string FieldMetadata::typeSchema
```

Type and format information for the field.

Definition at line 12 of file HeaderRecord.h.

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

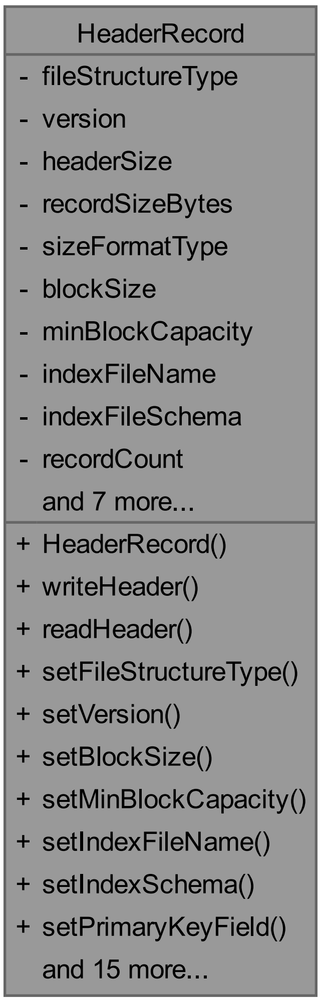
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.h

3.5 HeaderRecord Class Reference

Manages the header record for blocked sequence set files.

```
#include <HeaderRecord.h>
```

Collaboration diagram for HeaderRecord:



Public Member Functions

- HeaderRecord ()

  Default constructor for HeaderRecord.
- bool writeHeader (std::ofstream &file)

  Writes the header information to a file.
- bool readHeader (const std::string &filename)

  Reads and parses header information from a file.
- void setFileStructureType (const std::string &type)
- void setVersion (const std::string &ver)
- void setBlockSize (int size)
- void setMinBlockCapacity (double capacity)
- void setIndexFileName (const std::string &name)
- void setIndexSchema (const std::string &schema)
- void setPrimaryKeyField (int field)
- void setAvailListRBN (int rbn)
- void setActiveListRBN (int rbn)
- void setStaleFlag (bool flag)
- void addField (const std::string &name, const std::string &schema)

  Adds a new field definition to the header.
- std::string getFileStructureType () const
- std::string getVersion () const
- int getBlockSize () const
- double getMinBlockCapacity () const
- std::string getIndexFileName () const
- std::string getIndexSchema () const
- int getPrimaryKeyField () const
- int getAvailListRBN () const
- int getActiveListRBN () const
- bool getStaleFlag () const
- const std::vector< FieldMetadata > & getFields () const

Private Attributes

- std::string fileStructureType

  Type of file structure.
- std::string version

  Version of the file structure.
- int headerSize

  Size of the header record in bytes.
- int recordSizeBytes

  Number of bytes for record size integers.
- std::string sizeFormatType

  Format type for sizes (ASCII/binary)
- int blockSize

  Size of each block in bytes.
- double minBlockCapacity

  Minimum block capacity (default 50%)
- std::string indexFileName

  Name of the index file.
- std::string indexFileSchema

  Schema information for the index file.

- int recordCount

    Total number of records.
- int blockCount

    Total number of blocks.
- int fieldCount

    Number of fields per record.
- std::vector< FieldMetadata > fields

    Metadata for each field.
- int primaryKeyField

    Ordinal number of primary key field.
- int availListRBN

    RBN link to block avail-list.
- int activeListRBN

    RBN link to active sequence set list.
- bool isStale

    Stale flag for header.

3.5.1 Detailed Description

Manages the header record for blocked sequence set files.

This class handles reading and writing header records that contain metadata about the file structure, block organization, and field definitions.

Definition at line 22 of file HeaderRecord.h.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 HeaderRecord()

```
HeaderRecord::HeaderRecord ()
```

Default constructor for HeaderRecord.

Initializes all numeric members to sensible defaults

Definition at line 12 of file HeaderRecord.cpp.

3.5.3 Member Function Documentation

3.5.3.1 addField()

```
void HeaderRecord::addField (
             const std::string & name,
             const std::string & schema)
```

Adds a new field definition to the header.

Parameters

name	Name or ID of the field
schema	Type and format information for the field

Definition at line 34 of file HeaderRecord.cpp.

3.5.3.2 getActiveListRBN()

```
int HeaderRecord::getActiveListRBN () const  [inline]
```

Definition at line 62 of file HeaderRecord.h.

3.5.3.3 getAvailListRBN()

```
int HeaderRecord::getAvailListRBN () const  [inline]
```

Definition at line 61 of file HeaderRecord.h.

3.5.3.4 getBlockSize()

```
int HeaderRecord::getBlockSize () const  [inline]
```

Definition at line 56 of file HeaderRecord.h.

3.5.3.5 getFields()

```
const std::vector< FieldMetadata > & HeaderRecord::getFields () const  [inline]
```

Definition at line 64 of file HeaderRecord.h.

3.5.3.6 getFileStructureType()

```
std::string HeaderRecord::getFileStructureType () const  [inline]
```

Definition at line 54 of file HeaderRecord.h.

3.5.3.7 getIndexFileName()

```
std::string HeaderRecord::getIndexFileName () const  [inline]
```

Definition at line 58 of file HeaderRecord.h.

3.5.3.8 getIndexSchema()

```
std::string HeaderRecord::getIndexSchema () const  [inline]
```

Definition at line 59 of file HeaderRecord.h.

3.5.3.9 getMinBlockCapacity()

```
double HeaderRecord::getMinBlockCapacity () const  [inline]
```

Definition at line 57 of file HeaderRecord.h.

3.5.3.10 getPrimaryKeyField()

```
int HeaderRecord::getPrimaryKeyField () const  [inline]
```

Definition at line 60 of file HeaderRecord.h.

3.5.3.11 getStaleFlag()

```
bool HeaderRecord::getStaleFlag () const  [inline]
```

Definition at line 63 of file HeaderRecord.h.

3.5.3.12 getVersion()

```
std::string HeaderRecord::getVersion () const  [inline]
```

Definition at line 55 of file HeaderRecord.h.

3.5.3.13 readHeader()

```
bool HeaderRecord::readHeader (
             const std::string & filename)
```

Reads and parses header information from a file.

Parameters

filename	Name of the file to read from

Returns

true if successful, false otherwise

Definition at line 95 of file HeaderRecord.cpp.

3.5.3.14 setActiveListRBN()

```
void HeaderRecord::setActiveListRBN (
             int rbn)  [inline]
```

Definition at line 49 of file HeaderRecord.h.

3.5.3.15 setAvailListRBN()

```
void HeaderRecord::setAvailListRBN (
             int rbn)  [inline]
```

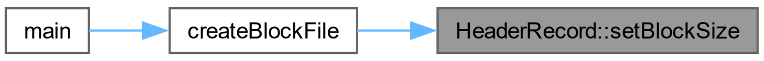
Definition at line 48 of file HeaderRecord.h.

3.5.3.16 setBlockSize()

```
void HeaderRecord::setBlockSize (
             int size)  [inline]
```

Definition at line 43 of file HeaderRecord.h.

Here is the caller graph for this function:

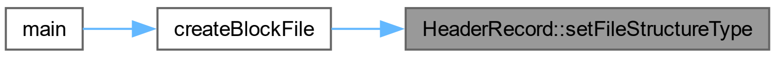


3.5.3.17 setFileStructureType()

```
void HeaderRecord::setFileStructureType (
             const std::string & type)  [inline]
```

Definition at line 41 of file HeaderRecord.h.

Here is the caller graph for this function:

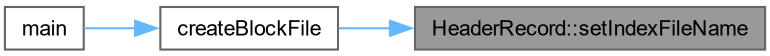


3.5.3.18 setIndexFileName()

```
void HeaderRecord::setIndexFileName (
             const std::string & name)  [inline]
```

Definition at line 45 of file HeaderRecord.h.

Here is the caller graph for this function:

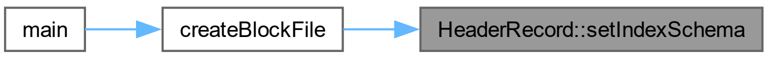


3.5.3.19 setIndexSchema()

```
void HeaderRecord::setIndexSchema (
             const std::string & schema)  [inline]
```

Definition at line 46 of file HeaderRecord.h.

Here is the caller graph for this function:

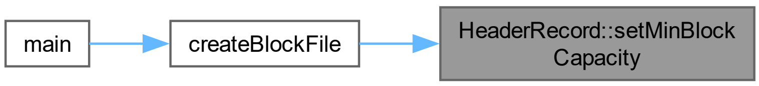


3.5.3.20 setMinBlockCapacity()

```
void HeaderRecord::setMinBlockCapacity (
             double capacity)  [inline]
```

Definition at line 44 of file HeaderRecord.h.

Here is the caller graph for this function:

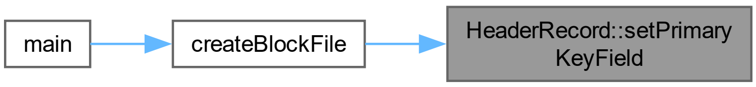


3.5.3.21 setPrimaryKeyField()

```
void HeaderRecord::setPrimaryKeyField (
             int field)  [inline]
```

Definition at line 47 of file HeaderRecord.h.

Here is the caller graph for this function:



3.5.3.22 setStaleFlag()

```
void HeaderRecord::setStaleFlag (
             bool flag)  [inline]
```

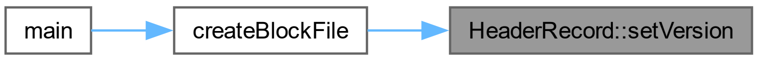
Definition at line 50 of file HeaderRecord.h.

3.5.3.23 setVersion()

```
void HeaderRecord::setVersion (
             const std::string & ver)  [inline]
```

Definition at line 42 of file HeaderRecord.h.

Here is the caller graph for this function:



3.5.3.24 writeHeader()

```
bool HeaderRecord::writeHeader (
             std::ofstream & file)
```

Writes the header information to a file.

Writes the header information to an already open file stream.

Parameters

file	ofstream of the file to write to

Returns

true if successful, false otherwise

Parameters

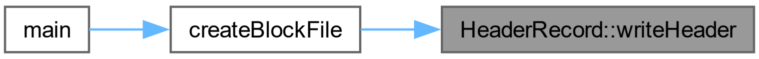
file	Reference to an open output file stream

Returns

true if successful, false otherwise

Definition at line 48 of file HeaderRecord.cpp.

Here is the caller graph for this function:



3.5.4 Member Data Documentation

3.5.4.1 activeListRBN

```
int HeaderRecord::activeListRBN  [private]
```

RBN link to active sequence set list.

Definition at line 82 of file HeaderRecord.h.

3.5.4.2 availListRBN

```
int HeaderRecord::availListRBN  [private]
```

RBN link to block avail-list.

Definition at line 81 of file HeaderRecord.h.

3.5.4.3 blockCount

```
int HeaderRecord::blockCount  [private]
```

Total number of blocks.

Definition at line 77 of file HeaderRecord.h.

3.5.4.4 blockSize

```
int HeaderRecord::blockSize  [private]
```

Size of each block in bytes.

Definition at line 72 of file HeaderRecord.h.

3.5.4.5 fieldCount

```
int HeaderRecord::fieldCount  [private]
```

Number of fields per record.

Definition at line 78 of file HeaderRecord.h.

3.5.4.6 fields

```
std::vector<FieldMetadata> HeaderRecord::fields  [private]
```

Metadata for each field.

Definition at line 79 of file HeaderRecord.h.

3.5.4.7 fileStructureType

```
std::string HeaderRecord::fileStructureType  [private]
```

Type of file structure.

Definition at line 67 of file HeaderRecord.h.

3.5.4.8 headerSize

```
int HeaderRecord::headerSize  [private]
```

Size of the header record in bytes.

Definition at line 69 of file HeaderRecord.h.

3.5.4.9 indexFileName

```
std::string HeaderRecord::indexFileName  [private]
```

Name of the index file.

Definition at line 74 of file HeaderRecord.h.

3.5.4.10 indexFileSchema

```
std::string HeaderRecord::indexFileSchema  [private]
```

Schema information for the index file.

Definition at line 75 of file HeaderRecord.h.

3.5.4.11 isStale

```
bool HeaderRecord::isStale  [private]
```

Stale flag for header.

Definition at line 83 of file HeaderRecord.h.

3.5.4.12 minBlockCapacity

```
double HeaderRecord::minBlockCapacity  [private]
```

Minimum block capacity (default 50%)

Definition at line 73 of file HeaderRecord.h.

3.5.4.13 primaryKeyField

```
int HeaderRecord::primaryKeyField  [private]
```

Ordinal number of primary key field.

Definition at line 80 of file HeaderRecord.h.

3.5.4.14 recordCount

```
int HeaderRecord::recordCount  [private]
```

Total number of records.

Definition at line 76 of file HeaderRecord.h.

3.5.4.15 recordSizeBytes

```
int HeaderRecord::recordSizeBytes [private]
```

Number of bytes for record size integers.

Definition at line 70 of file HeaderRecord.h.

3.5.4.16 sizeFormatType

```
std::string HeaderRecord::sizeFormatType [private]
```

Format type for sizes (ASCII/binary)

Definition at line 71 of file HeaderRecord.h.

3.5.4.17 version

```
std::string HeaderRecord::version [private]
```

Version of the file structure.

Definition at line 68 of file HeaderRecord.h.

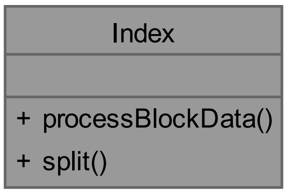
The documentation for this class was generated from the following files:

- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.h
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.cpp

3.6 Index Class Reference

```
#include <Index.h>
```

Collaboration diagram for Index:



Public Member Functions

- void processBlockData (const string &inputFileName, const string &outputFileName)

    Splits a string into tokens based on a specified delimiter.
- std::vector< std::string > split (const std::string &line, char delimiter)

    Processes block data from an input file and organizes it into an output file.

3.6.1 Detailed Description

Definition at line 8 of file Index.h.

3.6.2 Member Function Documentation

3.6.2.1 processBlockData()

```
void Index::processBlockData (
             const string & inputFileName,
             const string & outputFileName)
```

Splits a string into tokens based on a specified delimiter.

Processes block data from an input file and organizes it into an output file.

Parameters

line	The input string to be split.
delimiter	The character used as the delimiter for splitting the string.

Returns

A vector containing the tokens extracted from the input string.

This method reads data from an input file, extracts and processes relevant information, and writes the results into an output file. Each valid block and zip code pair is stored in the output file in the format "Block,Zip Code".

Parameters

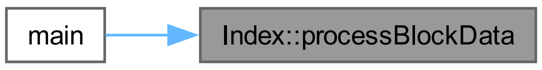
inputFileName	The name of the input file containing block data.
outputFileName	The name of the output file where processed data will be saved.

Definition at line 36 of file Index.cpp.

Here is the call graph for this function:

Index::processBlockData
Index::split

Here is the caller graph for this function:



3.6.2.2 split()

```
vector< string > Index::split (
            const std::string & line,
            char delimiter)
```

Processes block data from an input file and organizes it into an output file.

Splits a string into tokens based on a specified delimiter.

This method reads data from an input file, extracts and processes relevant information, and writes the results into an output file. Each valid block and zip code pair is stored in the output file in the format "Block,Zip Code".

Parameters

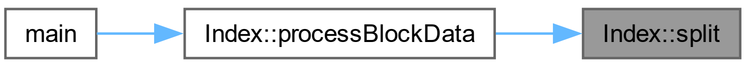
inputFileName	The name of the input file containing block data.
outputFileName	The name of the output file where processed data will be saved.
line	The input string to be split.
delimiter	The character used as the delimiter for splitting the string.

Returns

A vector containing the tokens extracted from the input string.

Definition at line 16 of file Index.cpp.

Here is the caller graph for this function:



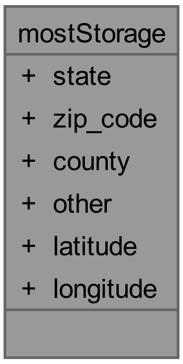
The documentation for this class was generated from the following files:

- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.h
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.cpp

3.7 mostStorage Struct Reference

Represents geographical location information for a zip code.

Collaboration diagram for mostStorage:



Public Attributes

- std::string state
- std::string zip_code
- std::string county
- std::string other
- double latitude
- double longitude

3.7.1 Detailed Description

Represents geographical location information for a zip code.

This struct stores detailed geographical data including state, zip code, latitude, and longitude coordinates

Definition at line 189 of file Block.cpp.

3.7.2 Member Data Documentation

3.7.2.1 county

```
std::string mostStorage::county
```

Definition at line 190 of file Block.cpp.

3.7.2.2 latitude

```
double mostStorage::latitude
```

Definition at line 191 of file Block.cpp.

3.7.2.3 longitude

```
double mostStorage::longitude
```

Definition at line 192 of file Block.cpp.

3.7.2.4 other

```
std::string mostStorage::other
```

Definition at line 190 of file Block.cpp.

3.7.2.5 state

```
std::string mostStorage::state
```

Definition at line 190 of file Block.cpp.

3.7.2.6 zip_code

```
std::string mostStorage::zip_code
```

Definition at line 190 of file Block.cpp.

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

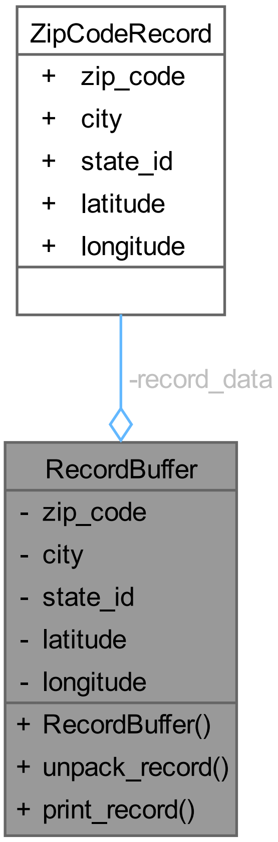
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.cpp

3.8 RecordBuffer Class Reference

A class to manage and process individual records.

```
#include <Buffer.h>
```

Collaboration diagram for RecordBuffer:



Public Member Functions

- RecordBuffer (const ZipCodeRecord &record)

  A buffer class to manage individual records.
- void unpack_record ()

  Unpacks fields from the record into individual attributes.
- void print_record () const

  Prints the contents of the record.

Private Attributes

- ZipCodeRecord record_data
- std::string zip_code
- std::string city
- std::string state_id
- double latitude
- double longitude

3.8.1 Detailed Description

A class to manage and process individual records.

Definition at line 42 of file Buffer.h.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 RecordBuffer()

```
RecordBuffer::RecordBuffer (
            const ZipCodeRecord & record)  [explicit]
```

A buffer class to manage individual records.

Definition at line 30 of file Buffer.cpp.

3.8.3 Member Function Documentation

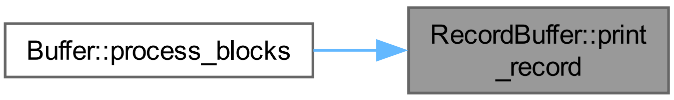
3.8.3.1 print_record()

```
void RecordBuffer::print_record () const
```

Prints the contents of the record.

Definition at line 41 of file Buffer.cpp.

Here is the caller graph for this function:



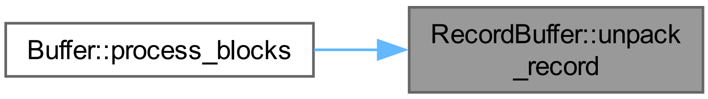
3.8.3.2 unpack_record()

```
void RecordBuffer::unpack_record ()
```

Unpacks fields from the record into individual attributes.

Definition at line 33 of file Buffer.cpp.

Here is the caller graph for this function:



3.8.4 Member Data Documentation

3.8.4.1 city

```
std::string RecordBuffer::city  [private]
```

Definition at line 59 of file Buffer.h.

3.8.4.2 latitude

```
double RecordBuffer::latitude  [private]
```

Definition at line 61 of file Buffer.h.

3.8.4.3 longitude

```
double RecordBuffer::longitude  [private]
```

Definition at line 62 of file Buffer.h.

3.8.4.4 record_data

```
ZipCodeRecord RecordBuffer::record_data  [private]
```

Definition at line 57 of file Buffer.h.

3.8.4.5 state_id

```
std::string RecordBuffer::state_id  [private]
```

Definition at line 60 of file Buffer.h.

3.8.4.6 zip_code

```
std::string RecordBuffer::zip_code  [private]
```

Definition at line 58 of file Buffer.h.

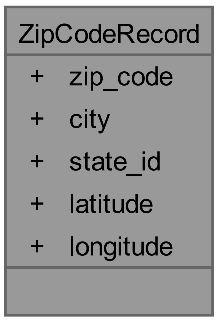
The documentation for this class was generated from the following files:

- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.h
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.cpp

3.9 ZipCodeRecord Struct Reference

```
#include <Buffer.h>
```

Collaboration diagram for ZipCodeRecord:



Public Attributes

- std::string zip_code
- std::string city
- std::string state_id
- double latitude
- double longitude

3.9.1 Detailed Description

Definition at line 11 of file Buffer.h.

3.9.2 Member Data Documentation

3.9.2.1 city

```
std::string ZipCodeRecord::city
```

Definition at line 13 of file Buffer.h.

3.9.2.2 latitude

```
double ZipCodeRecord::latitude
```

Definition at line 15 of file Buffer.h.

3.9.2.3 longitude

```
double ZipCodeRecord::longitude
```

Definition at line 16 of file Buffer.h.

3.9.2.4 state_id

```
std::string ZipCodeRecord::state_id
```

Definition at line 14 of file Buffer.h.

3.9.2.5 zip_code

```
std::string ZipCodeRecord::zip_code
```

Definition at line 12 of file Buffer.h.

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

- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.h

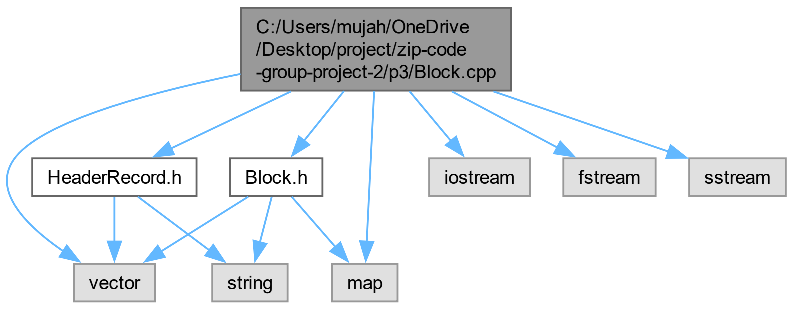
Chapter 4

File Documentation

4.1 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.cpp File Reference

```
#include "Block.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include "HeaderRecord.h"
```

Include dependency graph for Block.cpp:



Classes

- struct mostStorage

    Represents geographical location information for a zip code.

Functions

- bool createBlockFile (const std::string &inputFile, const std::string &outputFile, size_t BLOCK_SIZE)

  Creates a block file from an input CSV file.
- void parseBlockFile (const string &blockFile)

  Parses a block file and populates the global map of blocks.
- void dumpPhysicalOrder ()

  Dumps all blocks in physical order.
- void dumpLogicalOrder ()

  Dumps all blocks in logical order starting from the active list head.
- void listMost ()

  Finds and lists the extreme points (easternmost, westernmost, northernmost, southernmost) for each state.
- std::vector< std::string > splitZipLine (const std::string &str)

  Splits a string containing zip codes separated by "-z" delimiter.
- Block ∗ getBlockByRBN (int requestedRBN)

  Retrieves a block by its Relative Block Number (RBN)
- void search (const std::string &str, const std::string &indexName)

  Searches for a specific zip code in the block file and index file.
- void createBlock (int RBN, bool isAvailable, const vector< string > &records, int predecessorRBN, int successorRBN)

  Creates a new block and inserts it into the global map.

Variables

- map< int, Block > blocks

  Global map of blocks indexed by Relative Block Number (RBN).
- int listHeadRBN = -1

  Head of the active block list (RBN).
- int availHeadRBN = -1

  Head of the available block list (RBN).

4.1.1 Function Documentation

4.1.1.1 createBlock()

```
void createBlock (
             int RBN,
             bool isAvailable,
             const vector< string > & records,
             int predecessorRBN,
             int successorRBN)
```

Creates a new block and inserts it into the global map.

This function initializes a new block with the provided details and inserts it into the `blocks` map. It also updates the global head pointers for the active and available block lists as needed.

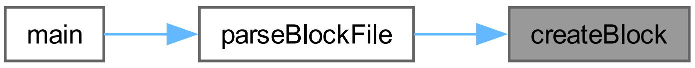
Parameters

RBN	Relative Block Number of the new block.
isAvailable	Flag indicating whether the block is available (true) or active (false).

records	List of records to store in the block.
predecessorRBN	RBN of the predecessor block in the chain.
successorRBN	RBN of the successor block in the chain.

Definition at line 470 of file Block.cpp.

Here is the caller graph for this function:



4.1.1.2 createBlockFile()

```
bool createBlockFile (
            const std::string & inputFile,
            const std::string & outputFile,
            size_t BLOCK_SIZE)
```

Creates a block file from an input CSV file.

This function reads an input CSV file, divides its data into fixed-size blocks, and writes those blocks into a new output file.

Parameters

inputFile	Path to the input CSV file.
outputFile	Path to the output block file.
BLOCK_SIZE	Maximum size of each block in bytes.

Returns

True if the file was successfully created, false otherwise.

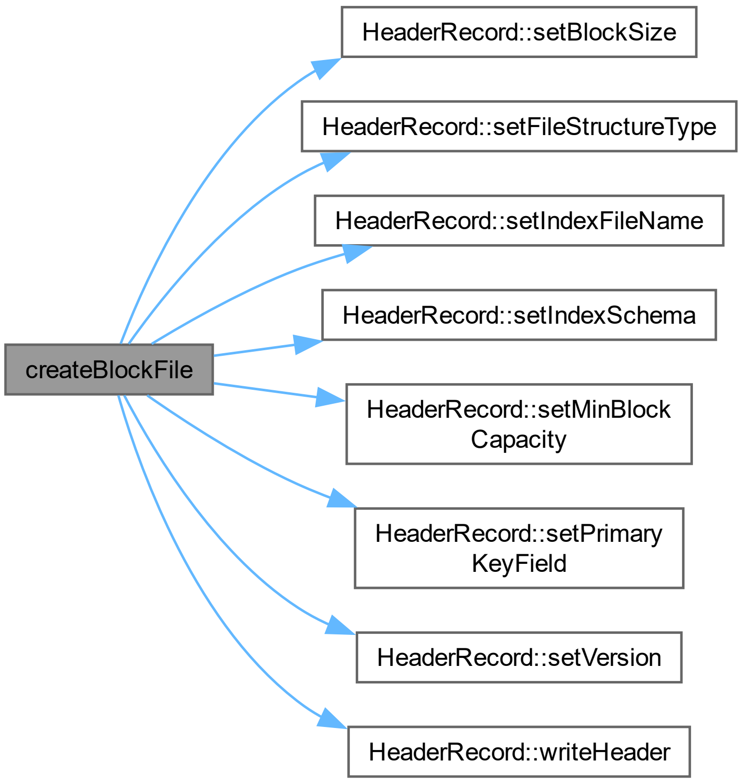
< Current block number being written

< Current size of the block in bytes

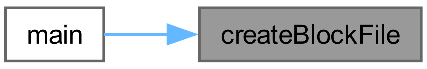
< Records for the current block

Definition at line 43 of file Block.cpp.

Here is the call graph for this function:


createBlockFile
HeaderRecord::setBlockSize
HeaderRecord::setFileStructureType
HeaderRecord::setIndexFileName
HeaderRecord::setIndexSchema
HeaderRecord::setMinBlock Capacity
HeaderRecord::setPrimary KeyField
HeaderRecord::setVersion
HeaderRecord::writeHeader

Here is the caller graph for this function:



4.1.1.3 dumpLogicalOrder()

```
void dumpLogicalOrder ()
```

Dumps all blocks in logical order starting from the active list head.

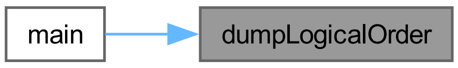
Dumps blocks in logical order starting from the active list head.

This function follows the logical chain of blocks using their successor links and prints the details of each block in sequence. < Start from the logical list head

< Move to the next block in the chain

Definition at line 170 of file Block.cpp.

Here is the caller graph for this function:



4.1.1.4 dumpPhysicalOrder()

```
void dumpPhysicalOrder ()
```

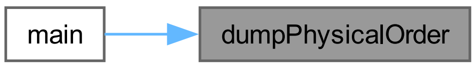
Dumps all blocks in physical order.

Dumps blocks in physical order based on their RBNs.

This function iterates through all blocks stored in the global `blocks` map and prints their details in ascending order of their RBNs.

Definition at line 153 of file Block.cpp.

Here is the caller graph for this function:



4.1.1.5 getBlockByRBN()

```
Block * getBlockByRBN (
             int requestedRBN)
```

Retrieves a block by its Relative Block Number (RBN)

This function searches the global blocks map for a block with the specified RBN. It returns a pointer to the block if found, or nullptr if the block does not exist.

Parameters

requestedRBN	The Relative Block Number of the block to retrieve

Returns

Block∗ Pointer to the block if found, nullptr otherwise

Note

Uses the global `blocks` map to perform the lookup

Warning

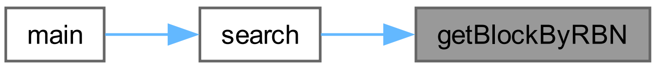
Returns nullptr if the block is not found

See also

blocks
Block

Definition at line 339 of file Block.cpp.

Here is the caller graph for this function:



4.1.1.6 listMost()

```
void listMost ()
```

Finds and lists the extreme points (easternmost, westernmost, northernmost, southernmost) for each state.

This function processes a collection of blocks containing location records, identifying the extreme geographical points for each state based on longitude and latitude coordinates.

The function performs the following steps:

- Iterates through all blocks and their records
- Extracts state, ZIP code, latitude, and longitude information
- Tracks the extreme points for each state
- Stores the results in a map of state to extreme locations
- Prints out the extreme point ZIP codes for each state

Note

Assumes records are in a specific order:

- Record 1: ZIP code
- Record 3: State
- Record 5: Latitude
- Record 6: Longitude

Precondition

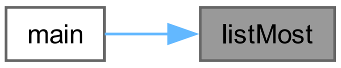
Requires a global `blocks` container with records

Postcondition

Prints extreme point information for each state

Definition at line 218 of file Block.cpp.

Here is the caller graph for this function:



4.1.1.7 parseBlockFile()

```
void parseBlockFile (
             const string & blockFile)
```

Parses a block file and populates the global map of blocks.

This function reads a block file, splits its content into blocks, and populates the `blocks` map with their respective details.

Parameters

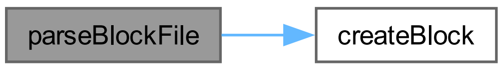
blockFile	Path to the block file to parse.

< Extracted RBN of the block

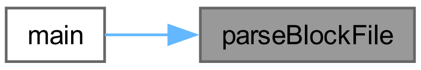
< Records in the block

Definition at line 120 of file Block.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.8 search()

```
void search (
             const std::string & str,
             const std::string & indexName)
```

Searches for a specific zip code in the block file and index file.

This function performs the following steps:

1. Opens the index file and block file
2. Searches for the given zip code in the index file
3. If found, retrieves the corresponding block
4. Parses the block records to extract and display matching record details

Parameters

str	The zip code to search for
indexName	The name of the index file containing zip code to RBN mappings

Precondition

Requires a valid index file and block file to be present

Postcondition

Prints the details of the matching record or a "not found" message

Note

Uses mostStorage struct to store and display record information

Assumes a specific record structure within each block

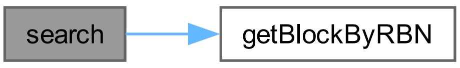
See also

mostStorage

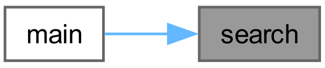
Block

Definition at line 375 of file Block.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.9 splitZipLine()

```
std::vector< std::string > splitZipLine (
             const std::string & str)
```

Splits a string containing zip codes separated by "-z" delimiter.

Parameters

str	Input string containing zip codes

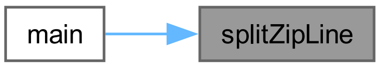
Returns

vector<string> Vector containing individual zip codes

Processes a string containing multiple zip codes separated by "-z" delimiter, handles special cases like strings starting with "-z" and empty segments

Definition at line 295 of file Block.cpp.

Here is the caller graph for this function:



4.1.2 Variable Documentation

4.1.2.1 availHeadRBN

```
int availHeadRBN = -1
```

Head of the available block list (RBN).

Stores the RBN of the first block in the available (free) list.

Definition at line 30 of file Block.cpp.

4.1.2.2 blocks

```
map<int, Block> blocks
```

Global map of blocks indexed by Relative Block Number (RBN).

This map stores all blocks, where the key is the RBN, and the value is the block object.

Definition at line 16 of file Block.cpp.

4.1.2.3 listHeadRBN

```
int listHeadRBN = -1
```

Head of the active block list (RBN).

Stores the RBN of the first block in the active (logical) sequence.

Definition at line 23 of file Block.cpp.

4.2 Block.cpp

Go to the documentation of this file.

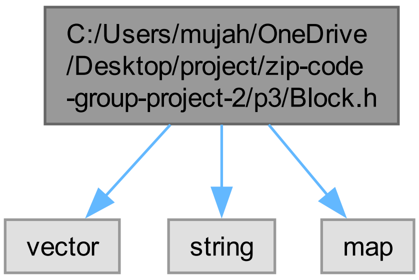
```
#include "Block.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include "HeaderRecord.h"

using namespace std;

map<int, Block> blocks;

int listHeadRBN = -1;

int availHeadRBN = -1;

bool createBlockFile(const std::string& inputFile, const std::string& outputFile, size_t BLOCK_SIZE) {
    ifstream inFile(inputFile);
    ofstream outFile(outputFile);
    if (!inFile.is_open() || !outFile.is_open()) {
        cerr « "Error: Could not open input or output file: " « inputFile « " | " « outputFile « endl;
        return false;
    }

    HeaderRecord header;

    // Set basic header information
    header.setFileStructureType("blocked_sequence_set");
    header.setVersion("1.0");
    header.setBlockSize(512);  // Default block size
    header.setMinBlockCapacity(0.5);  // 50% minimum capacity
    header.setIndexFileName("headerTest.idx");
    header.setIndexSchema("key:string,rbn:int");

    // Set primary key field (zip_code is field 0)
    header.setPrimaryKeyField(0);

    // First write the header
    if (!header.writeHeader(outFile)) {
        std::cerr « "Failed to write header to output file" « std::endl;
        return false;
    }

    size_t blockNumber = 1;
    size_t currentBlockSize = 0;
    vector<string> blockRecords;

    string line;
    getline(inFile, line); // Skip header
    while (getline(inFile, line)) {
        size_t lineSize = line.size() + 1; // Include newline character
        if (currentBlockSize + lineSize > BLOCK_SIZE) {
            // Write the current block to the output file
            outFile « blockNumber « ":";
            for (size_t i = 0; i < blockRecords.size(); i++) {
                outFile « blockRecords[i];
                if (i < blockRecords.size() - 1) outFile « ",";
            }
            outFile « "\n";

            blockRecords.clear();
            currentBlockSize = 0;
            blockNumber++;
        }

        blockRecords.push_back(line);
        currentBlockSize += lineSize;
    }

    // Write the last block if there are remaining records
    if (!blockRecords.empty()) {
        outFile « blockNumber « ":";
        for (size_t i = 0; i < blockRecords.size(); i++) {
            outFile « blockRecords[i];
            if (i < blockRecords.size() - 1) outFile « ",";
        }
        outFile « "\n";
    }

    inFile.close();
    outFile.close();

    return true;
}

void parseBlockFile(const string& blockFile) {
    ifstream inFile(blockFile);
    if (!inFile.is_open()) {
        cerr « "Error: Could not open block file: " « blockFile « endl;
        return;
    }

    string line;
    while (getline(inFile, line)) {
        size_t colonPos = line.find(':');
        int RBN = stoi(line.substr(0, colonPos));
        string recordsPart = line.substr(colonPos + 1);

        vector<string> records;
        stringstream recordStream(recordsPart);
        string record;
        while (getline(recordStream, record, ',')) {
            records.push_back(record);
        }

        // Create a block using the parsed data
        createBlock(RBN, false, records, -1, -1);
    }

    inFile.close();
}

void dumpPhysicalOrder() {
    cout « "Dumping Blocks by Physical Order:\n";
    for (const auto& [RBN, block] : blocks) {
        cout « "RBN: " « RBN « " ";
        for (const string& record : block.records) {
            cout « record « " ";
        }
        cout « "\n";
    }
}

void dumpLogicalOrder() {
    cout « "Dumping Blocks by Logical Order:\n";
    int currentRBN = listHeadRBN;
    while (currentRBN != -1) {
        const Block& block = blocks[currentRBN];
        cout « "RBN: " « currentRBN « " ";
        for (const string& record : block.records) {
            cout « record « " " ;
        }
        cout « "\n";
        currentRBN = block.successorRBN;
    }
}
struct mostStorage {
    std::string state, zip_code, county, other;
    double latitude;
    double longitude;
};
void listMost() {
    int currentRBN = listHeadRBN;
    int recordPart = 0;
    int testnum = 0;
    mostStorage current, easternmost, westernmost, northernmost, southernmost;
    std::map<string, std::vector<mostStorage» sorted_directions;


    for (const auto& [RBN, block] : blocks) {
          bool initialized = false;

            for (const string& record : block.records) {
                    recordPart++;
                    if(recordPart == 1){
                    current.zip_code = record;
                    }

                    if(recordPart == 3){
                    current.state = record;
                    }

                if(recordPart == 5){
                    current.latitude = std::stod(record);
                }
                if(recordPart == 6){
                    current.longitude = std::stod(record);
                    if (!initialized) {
                    easternmost = current;
                    westernmost = current;
                    northernmost = current;
                    southernmost = current;
                    initialized = true;
                }
                if ( current.longitude < easternmost.longitude ) {
                    easternmost = current;
                }
                if ( current.longitude > westernmost.longitude ) {
                    westernmost = current;
                }
                if ( current.latitude > northernmost.latitude ) {
                    northernmost = current;
                }
                if ( current.latitude < southernmost.latitude ) {
                    southernmost = current;
                }
                recordPart=0;
                sorted_directions[ current.state ] = { easternmost, westernmost, northernmost, southernmost };
                }




             }
                            }





    cout «"State: "« "Easternmost: " « "westernmost: "« "northernnmost: "« "southernnmost: " «endl;
    for (const auto& [state, locations] : sorted_directions) {
        if (locations.size() == 4) {  // Ensure we have all 4 directional records
                cout « state « ","
                « locations[0].zip_code « ","  // Easternmost
                 « locations[1].zip_code « ","  // Westernmost
                 « locations[2].zip_code « ","  // Northernmost
                 « locations[3].zip_code « "\n";  // Southernmost
        }
    }
}
std::vector<std::string> splitZipLine(const std::string& str) {
    std::vector<std::string> result;
    size_t start = 0;
    size_t end = str.find("-z", start);

    // Skip the first empty part if string starts with "-z"
    if (start == end) {
        start += 2;  // length of "-z"
        end = str.find("-z", start);
    }

    while (end != std::string::npos) {
        // Add the part between current position and next "-z"
        if (end - start > 0) {
            result.push_back(str.substr(start, end - start));
        }
        start = end + 2;  // Skip over "-z"
        end = str.find("-z", start);
    }

    // Add the last part if there's anything left
    if (start < str.length()) {
        result.push_back(str.substr(start));
    }

    return result;
}


Block* getBlockByRBN(int requestedRBN) {
    // Check if the block exists in the global blocks map
    auto it = blocks.find(requestedRBN);

    if (it != blocks.end()) {
        // Block found, return a pointer to the block
        return &(it->second);
    } else {
        // Block not found
        std::cerr « "Block with RBN " « requestedRBN « " not found." « std::endl;
        return nullptr;
    }
}

void search(const std::string& str, const std::string& indexName){
    mostStorage current;
    bool notfound = true;
    std::string correct_line;
     std::ifstream file2(indexName); // Open the file add name later
    if (!file2.is_open()) {
        std::cerr « "Error opening file: index.txt " « std::endl;
        return;
    }
     std::ifstream file3("block.txt"); // Open the file add name later
    if (!file3.is_open()) {
        std::cerr « "Error opening file: block.txt " « std::endl;
        return;
    }
    std::string strcopy = str;

    std::string rbn, zipcode, line;
    int recordPart = 0;
    //int i = 5;
    getline( file2, line );
    line = "";
    while ((file2 » zipcode » rbn) && !file2.eof()) {  // reads word by word
        if(zipcode == str){
            int block = std::stoi(rbn);
            cout « "Zipcode:  " « zipcode « " is at "« block «endl;
            Block* myBlock = getBlockByRBN(block);
            for (const string& record : myBlock->records) {
                    recordPart++;
                    if(recordPart == 1){
                    current.zip_code = record;

                    }
                    if(recordPart == 2){
                    current.other = record;

                    }
                    if(recordPart == 3){
                    current.state = record;
                    }
                    if(recordPart == 4){
                    current.county = record;

                    }
                if(recordPart == 5){
                    current.latitude = std::stod(record);
                }
                if(recordPart == 6){
                    current.longitude = std::stod(record);

                    if(current.zip_code == zipcode){
                        cout « current.zip_code « " " «current.other « " "«current.state « "
"«current.county
                    « " "«current.latitude « " " «current.longitude « " "« endl;
                    notfound = false;
                    break;
                    }
                //  current.longitude = std::stod(record);
                /*  if (!initialized) {
                    easternmost = current;
                    westernmost = current;
                    northernmost = current;
                    southernmost = current;
                    initialized = true;*/


                recordPart=0;
            }
                }
            }


        //  break;
            }



    if(notfound){
            cout« str « " was not found in the file."«endl;
            }
            file2.close();
}


void createBlock(int RBN, bool isAvailable, const vector<string>& records, int predecessorRBN, int
successorRBN) {
    Block block;
    block.RBN = RBN;
    block.isAvailable = isAvailable;
    block.records = records;
    block.predecessorRBN = predecessorRBN;
    block.successorRBN = successorRBN;

    blocks[RBN] = block;

    // Update the global head pointers
    if (!isAvailable && listHeadRBN == -1) {
        listHeadRBN = RBN;
    }
    if (isAvailable && availHeadRBN == -1) {
        availHeadRBN = RBN;
    }
}
```

4.3 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.h File Reference

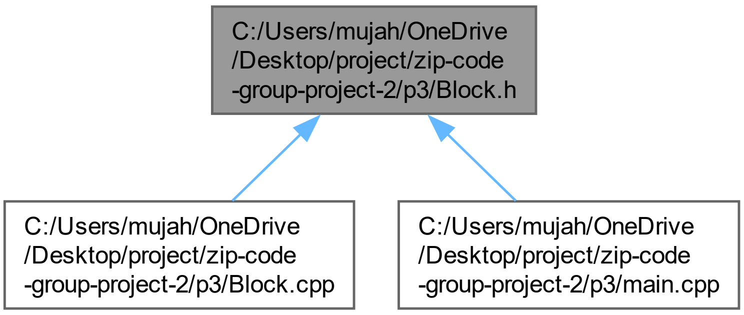
Declaration of the Block structure and related global variables and functions for managing a blocked sequence set.

```
#include <vector>
#include <string>
#include <map>
```

Include dependency graph for Block.h:



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


C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/Block.h
C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/Block.cpp
C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/main.cpp

Classes

- struct Block

  Represents a single block in the blocked sequence set.

Functions

- void dumpPhysicalOrder ()

  Dumps blocks in physical order based on their RBNs.
- void dumpLogicalOrder ()

  Dumps blocks in logical order starting from the active list head.
- void createBlock (int RBN, bool isAvailable, const std::vector< std::string > &records, int predecessorRBN, int successorRBN)

  Creates a new block and inserts it into the global map.
- void parseBlockFile (const std::string &blockFile)

  Parses a block file and populates the global map of blocks.
- bool createBlockFile (const std::string &inputFile, const std::string &outputFile, size_t BLOCK_SIZE=512)

  Creates a block file from an input CSV file.
- void listMost ()

  Finds and lists the extreme points (easternmost, westernmost, northernmost, southernmost) for each state.
- void search (const std::string &str, const std::string &indexName)

  Searches for a specific zip code in the block file and index file.

Variables

- std::map< int, Block > blocks

  Global map of blocks indexed by Relative Block Number (RBN).
- int listHeadRBN

  Head of the active block list (RBN).
- int availHeadRBN

  Head of the available block list (RBN).

4.3.1 Detailed Description

Declaration of the Block structure and related global variables and functions for managing a blocked sequence set.

This file defines the structure of a block and declares global variables and functions used to manage a sequence of blocks for a blocked file system. It supports operations such as dumping blocks in physical or logical order.

Date

11/21/2024

Definition in file Block.h.

4.3.2 Function Documentation

4.3.2.1 createBlock()

```
void createBlock (
            int RBN,
            bool isAvailable,
            const std::vector< std::string > & records,
            int predecessorRBN,
            int successorRBN)
```

Creates a new block and inserts it into the global map.

Parameters

RBN	Relative Block Number of the new block.
isAvailable	Flag indicating whether the block is available (true) or active (false).
records	List of records to store in the block.
predecessorRBN	RBN of the predecessor block in the chain.
successorRBN	RBN of the successor block in the chain.

This function initializes a new block with the provided parameters and adds it to the global map.

4.3.2.2 createBlockFile()

```
bool createBlockFile (
            const std::string & inputFile,
            const std::string & outputFile,
            size_t BLOCK_SIZE)
```

Creates a block file from an input CSV file.

This function reads an input CSV file, divides the data into blocks of a specified size, and writes the blocks to an output file.

Parameters

inputFile	Path to the input CSV file.
outputFile	Path to the output block file.
BLOCK_SIZE	Maximum size of each block in bytes (default is 512).

Returns

True if successful, false otherwise.

This function reads an input CSV file, divides its data into fixed-size blocks, and writes those blocks into a new output file.

Parameters

inputFile	Path to the input CSV file.
outputFile	Path to the output block file.
BLOCK_SIZE	Maximum size of each block in bytes.

Returns

True if the file was successfully created, false otherwise.

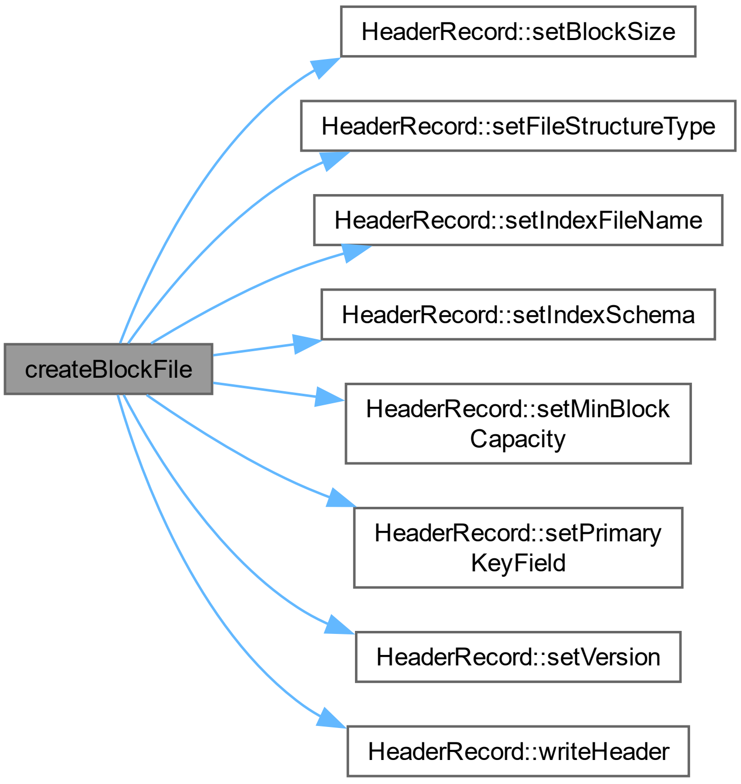
< Current block number being written

< Current size of the block in bytes

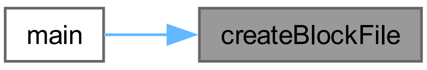
< Records for the current block

Definition at line 43 of file Block.cpp.

Here is the call graph for this function:


HeaderRecord::setBlockSize
HeaderRecord::setFileStructureType
HeaderRecord::setIndexFileName
HeaderRecord::setIndexSchema
createBlockFile
HeaderRecord::setMinBlock Capacity
HeaderRecord::setPrimary KeyField
HeaderRecord::setVersion
HeaderRecord::writeHeader

Here is the caller graph for this function:



4.3.2.3 dumpLogicalOrder()

```
void dumpLogicalOrder ()
```

Dumps blocks in logical order starting from the active list head.

This function follows the logical chain of blocks using their successor links and prints details of each block.

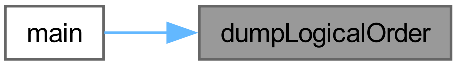
Dumps blocks in logical order starting from the active list head.

This function follows the logical chain of blocks using their successor links and prints the details of each block in sequence. < Start from the logical list head

< Move to the next block in the chain

Definition at line 170 of file Block.cpp.

Here is the caller graph for this function:



4.3.2.4 dumpPhysicalOrder()

```
void dumpPhysicalOrder ()
```

Dumps blocks in physical order based on their RBNs.

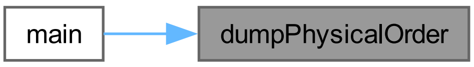
This function iterates over all blocks in ascending order of their RBNs and prints their details. Available blocks are explicitly marked.

Dumps blocks in physical order based on their RBNs.

This function iterates through all blocks stored in the global `blocks` map and prints their details in ascending order of their RBNs.

Definition at line 153 of file Block.cpp.

Here is the caller graph for this function:



4.3.2.5 listMost()

```
void listMost ()
```

Finds and lists the extreme points (easternmost, westernmost, northernmost, southernmost) for each state.

This function processes a collection of blocks containing location records, identifying the extreme geographical points for each state based on longitude and latitude coordinates.

The function performs the following steps:

- Iterates through all blocks and their records
- Extracts state, ZIP code, latitude, and longitude information
- Tracks the extreme points for each state
- Stores the results in a map of state to extreme locations
- Prints out the extreme point ZIP codes for each state

Note

Assumes records are in a specific order:

- Record 1: ZIP code
- Record 3: State
- Record 5: Latitude
- Record 6: Longitude

Precondition

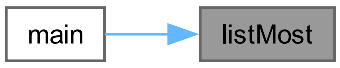
Requires a global `blocks` container with records

Postcondition

Prints extreme point information for each state

Definition at line 218 of file Block.cpp.

Here is the caller graph for this function:



4.3.2.6 parseBlockFile()

```
void parseBlockFile (
            const std::string & blockFile)
```

Parses a block file and populates the global map of blocks.

This function reads a block file, extracts block information, and populates the global `blocks` map.

Parameters

blockFile	Path to the block file to parse.

4.3.2.7 search()

```
void search (
            const std::string & str,
            const std::string & indexName)
```

Searches for a specific zip code in the block file and index file.

This function performs the following steps:

1. Opens the index file and block file
2. Searches for the given zip code in the index file
3. If found, retrieves the corresponding block
4. Parses the block records to extract and display matching record details

Parameters

str	The zip code to search for
indexName	The name of the index file containing zip code to RBN mappings

Precondition

Requires a valid index file and block file to be present

Postcondition

Prints the details of the matching record or a "not found" message

Note

Uses mostStorage struct to store and display record information

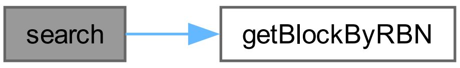
Assumes a specific record structure within each block

See also

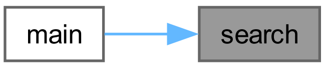
mostStorage
Block

Definition at line 375 of file Block.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



4.3.3 Variable Documentation

4.3.3.1 availHeadRBN

```
int availHeadRBN  [extern]
```

Head of the available block list (RBN).

Stores the RBN of the first block in the available (free) list.

Definition at line 30 of file Block.cpp.

4.3.3.2 blocks

```
std::map<int, Block> blocks  [extern]
```

Global map of blocks indexed by Relative Block Number (RBN).

This map stores all blocks, with the RBN as the key and the corresponding block as the value.

This map stores all blocks, where the key is the RBN, and the value is the block object.

Definition at line 16 of file Block.cpp.

4.3.3.3 listHeadRBN

```
int listHeadRBN  [extern]
```

Head of the active block list (RBN).

Stores the RBN of the first block in the logical (active) sequence.

Stores the RBN of the first block in the active (logical) sequence.

Definition at line 23 of file Block.cpp.

4.4 Block.h

Go to the documentation of this file.

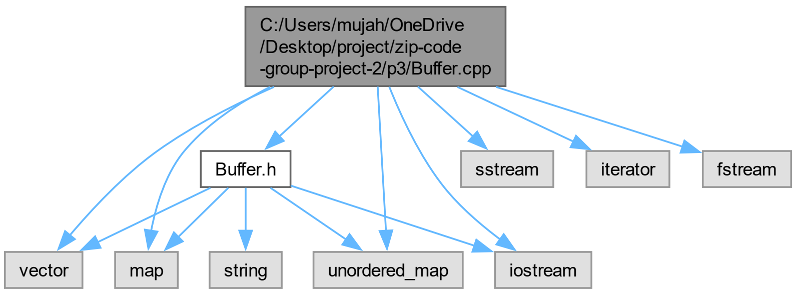
```
#ifndef BLOCK_H
#define BLOCK_H

#include <vector>
#include <string>
#include <map>

struct Block {
    int RBN;
    bool isAvailable;
    std::vector<std::string> records;
    int predecessorRBN;
    int successorRBN;
};

extern std::map<int, Block> blocks;

extern int listHeadRBN;

extern int availHeadRBN;

void dumpPhysicalOrder();

void dumpLogicalOrder();

void createBlock(int RBN, bool isAvailable, const std::vector<std::string>& records, int
predecessorRBN, int successorRBN);

void parseBlockFile(const std::string& blockFile);

bool createBlockFile(const std::string& inputFile, const std::string& outputFile, size_t BLOCK_SIZE =
512);



void listMost();

void search(const std::string& str, const std::string& indexName);

#endif // BLOCK_H
```

4.5 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.cpp File Reference

```
#include "Buffer.h"
#include <iostream>
#include <sstream>
#include <iterator>
#include <fstream>
#include <map>
```

```
#include <vector>
#include <unordered_map>
```

Include dependency graph for Buffer.cpp:


C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.cpp
Buffer.h
sstream
iterator
fstream
vector
map
string
unordered_map
iostream

4.6 Buffer.cpp

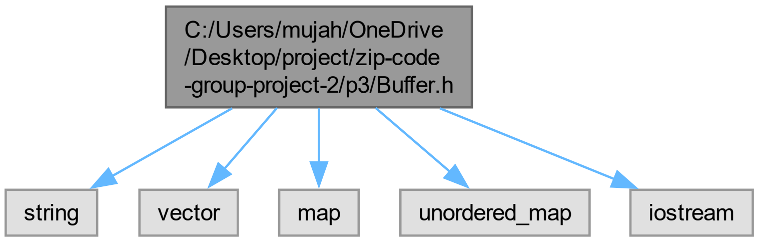
Go to the documentation of this file.

```
#include "Buffer.h"
#include <iostream>
#include <sstream>
#include <iterator>
#include <fstream>
#include <map>
#include <vector>
#include <unordered_map>

    // BlockBuffer class method definitions
BlockBuffer::BlockBuffer(const std::unordered_map<std::string, ZipCodeRecord>& block)
    : block_data(block) {}

std::vector<ZipCodeRecord> BlockBuffer::unpack_block() const {
    std::vector<ZipCodeRecord> records;
    for (const auto& entry : block_data) {
        records.push_back(entry.second);
    }
    return records;
}


// RecordBuffer class method definitions
RecordBuffer::RecordBuffer(const ZipCodeRecord& record)
    : record_data(record) {}

void RecordBuffer::unpack_record() {
    zip_code = record_data.zip_code;
    city = record_data.city;
    state_id = record_data.state_id;
    latitude = record_data.latitude;
    longitude = record_data.longitude;
}

void RecordBuffer::print_record() const {
    std::cout « "ZipCode: " « zip_code
              « ", City: " « city
              « ", State: " « state_id
              « ", Latitude: " « latitude
              « ", Longitude: " « longitude
              « std::endl;
}


bool Buffer::read_csv(const std::string& csv_filename, size_t records_per_block) {
    std::ifstream file(csv_filename);
    if (!file.is_open()) {
        std::cerr « "Failed to open file: " « csv_filename « std::endl;
        return false;
    }
    std::string line;
    std::getline(file, line); // Skip the header line

    size_t block_number = 0;
    size_t record_count = 0;

    while (std::getline(file, line)) {
        ZipCodeRecord record = parse_csv_line(line);
        add_record(block_number, record);

        if (++record_count >= records_per_block) {
            block_number++;
            record_count = 0;
        }
    }

    file.close();
    std::cout « "CSV loaded into the buffer successfully." « std::endl;
    return true;
}

ZipCodeRecord Buffer::parse_csv_line(const std::string& line) const {
    std::stringstream ss(line);
    std::string token;
    ZipCodeRecord record;

    std::getline(ss, record.zip_code, ',');
    std::getline(ss, record.city, ',');
    std::getline(ss, record.state_id, ',');
    std::getline(ss, token, ',');
    record.latitude = std::stod(token);
    std::getline(ss, token, ',');
    record.longitude = std::stod(token);

    return record;
}

void Buffer::process_blocks() {
    for (const auto& block_entry : blocks) {
        size_t block_number = block_entry.first;
        const auto& block = block_entry.second;

        BlockBuffer block_buffer(block);
        std::vector<ZipCodeRecord> records = block_buffer.unpack_block();

        std::cout « "Processing Block " « block_number « std::endl;
        for (const auto& record : records) {
            RecordBuffer record_buffer(record);
            record_buffer.unpack_record();
            record_buffer.print_record();
        }
    }
}

void Buffer::sort_records() {
    std::map<std::string, ZipCodeRecord> sorted_records;

    for (const auto& record : records) {
        sorted_records[record.zip_code] = record;
    }

    std::cout « "Records sorted by Zip Code:" « std::endl;
    for (const auto& entry : sorted_records) {
        const auto& record = entry.second;
        std::cout « "ZipCode: " « record.zip_code
                  « ", City: " « record.city
                  « ", State: " « record.state_id
                  « ", Latitude: " « record.latitude
                  « ", Longitude: " « record.longitude
                  « std::endl;
    }
}

void Buffer::add_record(size_t block_number, const ZipCodeRecord& record) {
    blocks[block_number][record.zip_code] = record;
    records.push_back(record);
}

std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord» Buffer::get_blocks() const {
    return blocks;
}

void Buffer::dump_blocks() const {
    for (const auto& block : blocks) {
        std::cout « "Block " « block.first « " contains the following ZipCodeRecords:" « std::endl;
        for (const auto& record_pair : block.second) {
            std::cout « "ZipCode: " « record_pair.second.zip_code
                      « ", City: " « record_pair.second.city
                      « ", State: " « record_pair.second.state_id
                      « ", Latitude: " « record_pair.second.latitude
                      « ", Longitude: " « record_pair.second.longitude
                      « std::endl;
        }
    }
}
```

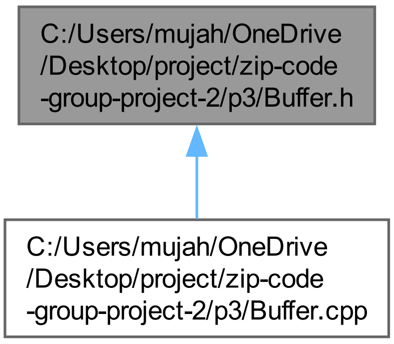
4.7 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.h File Reference

```
#include <string>
#include <vector>
#include <map>
#include <unordered_map>
#include <iostream>
```

Include dependency graph for Buffer.h:



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


C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/Buffer.h
C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/Buffer.cpp

Classes

- struct ZipCodeRecord
- class BlockBuffer

    A class to manage and process blocks of data.
- class RecordBuffer

    A class to manage and process individual records.
- class Buffer

    A buffer class to manage ZipCodeRecords and process blocks of data.

4.8 Buffer.h

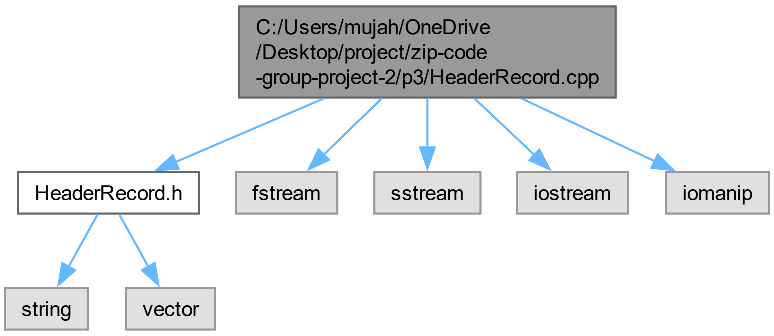
Go to the documentation of this file.

```
#ifndef BUFFER_H
#define BUFFER_H

#include <string>
#include <vector>
#include <map>
#include <unordered_map>
#include <iostream>

// Define a struct to represent a zip code record.
struct ZipCodeRecord {
    std::string zip_code;
    std::string city;
    std::string state_id;
    double latitude;
    double longitude;
};

// Forward declaration of the Buffer class
class Buffer;

class BlockBuffer {
public:
    explicit BlockBuffer(const std::unordered_map<std::string, ZipCodeRecord>& block);

    std::vector<ZipCodeRecord> unpack_block() const;

private:
    std::unordered_map<std::string, ZipCodeRecord> block_data;
};

class RecordBuffer {
public:
    explicit RecordBuffer(const ZipCodeRecord& record);

    void unpack_record();

    void print_record() const;

private:
    ZipCodeRecord record_data;
    std::string zip_code;
    std::string city;
    std::string state_id;
    double latitude;
    double longitude;
};

class Buffer {
public:
    bool read_csv(const std::string& csv_filename, size_t records_per_block);

    ZipCodeRecord parse_csv_line(const std::string& line) const;

    void process_blocks();

    void sort_records();

    void add_record(size_t block_number, const ZipCodeRecord& record);

    std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord» get_blocks() const;

    void dump_blocks() const;

private:
    // Map where the key is the block number, and the value is a map of ZipCodeRecords in the block.
    std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord» blocks;

    // A flat list of all ZipCodeRecords, used for sorting and other operations.
    std::vector<ZipCodeRecord> records;
};

#endif // BUFFER_H
```

4.9 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.cpp File Reference

```
#include "HeaderRecord.h"
#include <fstream>
#include <sstream>
#include <iostream>
#include <iomanip>
```

Include dependency graph for HeaderRecord.cpp:



4.10 HeaderRecord.cpp

Go to the documentation of this file.

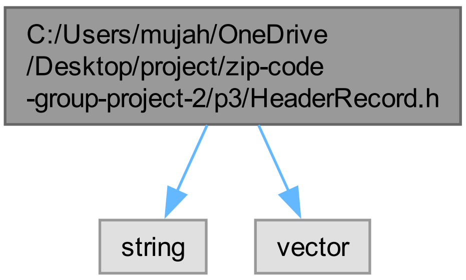
```
#include "HeaderRecord.h"
#include <fstream>
#include <sstream>
#include <iostream>
#include <iomanip>

HeaderRecord::HeaderRecord()
    : headerSize(0)
    , recordSizeBytes(-1)
    , blockSize(512)  // Default block size of 512 bytes
    , minBlockCapacity(0.5)  // Default 50% minimum capacity
    , recordCount(40933) // Record count of input data
    , blockCount(3679)
    , fieldCount(6) // Default 6 as all used zipcode data has 6 peramiters
    , primaryKeyField(0)
    , availListRBN(-1)
    , activeListRBN(-1)
    , isStale(false) {
    fileStructureType = "blocked_sequence_set";
    version = "1.0";
    sizeFormatType = "ASCII";
```

```
}

void HeaderRecord::addField(const std::string& name, const std::string& schema) {
    FieldMetadata field;
    field.name = name;
    field.typeSchema = schema;
    fields.push_back(field);
    fieldCount = fields.size();
}

bool HeaderRecord::writeHeader(std::ofstream& file) {
    if (!file.is_open()) {
        std::cerr « "Error: File stream is not open" « std::endl;
        return false;
    }

    // Lambda to write length-indicated field
    auto writeField = [&file](const std::string& value) {
        std::string lengthStr = std::to_string(value.length());
        if (lengthStr.length() < 2) lengthStr = "0" + lengthStr;
        file « lengthStr « value « ",";
    };

    // Write main header fields
    writeField(fileStructureType);
    writeField(version);
    writeField(std::to_string(headerSize));
    writeField(std::to_string(recordSizeBytes));
    writeField(sizeFormatType);
    writeField(std::to_string(blockSize));
    writeField(std::to_string(static_cast<int>(minBlockCapacity * 100)));
    writeField(indexFileName);
    writeField(indexFileSchema);
    writeField(std::to_string(recordCount));
    writeField(std::to_string(blockCount));
    writeField(std::to_string(fieldCount));
    writeField(std::to_string(primaryKeyField));
    writeField(std::to_string(availListRBN));
    writeField(std::to_string(activeListRBN));
    file « (isStale ? "1" : "0") « "\n";

    // Write field metadata
    for (const auto& field : fields) {
        writeField(field.name);
        writeField(field.typeSchema);
        file « "\n";
    }

    return true;
}

bool HeaderRecord::readHeader(const std::string& filename) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr « "Error: Unable to open file for reading: " « filename « std::endl;
        return false;
    }

    std::string line;
    if (std::getline(file, line)) {
        std::stringstream ss(line);

        // Lambda to read length-indicated field
        auto readField = [](std::stringstream& ss) -> std::string {
            std::string lenStr;
            lenStr.resize(2);
            if (!ss.read(&lenStr[0], 2)) return "";

            if (!std::isdigit(lenStr[0]) || !std::isdigit(lenStr[1])) {
                throw std::runtime_error("Invalid length indicator");
            }

            int length = std::stoi(lenStr);
            std::string value;
            value.resize(length);
            if (!ss.read(&value[0], length)) return "";

            if (ss.peek() == ',') ss.ignore();
            return value;
        };

        try {
            // Read main header fields
            fileStructureType = readField(ss);
            version = readField(ss);
            headerSize = std::stoi(readField(ss));
            recordSizeBytes = std::stoi(readField(ss));
            sizeFormatType = readField(ss);
            blockSize = std::stoi(readField(ss));
            minBlockCapacity = std::stoi(readField(ss)) / 100.0;
            indexFileName = readField(ss);
            indexFileSchema = readField(ss);
            recordCount = std::stoi(readField(ss));
            blockCount = std::stoi(readField(ss));
            fieldCount = std::stoi(readField(ss));
            primaryKeyField = std::stoi(readField(ss));
            availListRBN = std::stoi(readField(ss));
            activeListRBN = std::stoi(readField(ss));

            std::string staleStr;
            ss » staleStr;
            isStale = (staleStr == "1");

            // Read field metadata
            fields.clear();
            for (int i = 0; i < fieldCount && std::getline(file, line); i++) {
                std::stringstream fieldSS(line);
                FieldMetadata metadata;
                metadata.name = readField(fieldSS);
                metadata.typeSchema = readField(fieldSS);
                fields.push_back(metadata);
            }
        }
        catch (const std::exception& e) {
            std::cerr « "Error parsing header: " « e.what() « std::endl;
            return false;
        }
    }

    file.close();
    return true;
}
```

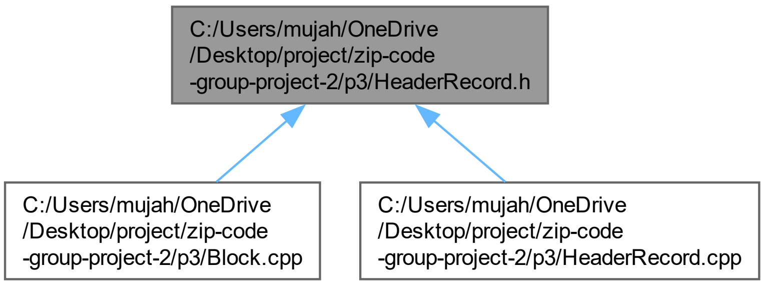
4.11 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.h File Reference

```
#include <string>
#include <vector>
```

Include dependency graph for HeaderRecord.h:


C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.h
string
vector

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


C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/HeaderRecord.h
C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/Block.cpp
C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/HeaderRecord.cpp

Classes

- struct FieldMetadata

  Metadata structure for field information in the header.
- class HeaderRecord

  Manages the header record for blocked sequence set files.

4.12 HeaderRecord.h

Go to the documentation of this file.

```
#ifndef HEADER_RECORD_H
#define HEADER_RECORD_H

#include <string>
#include <vector>

struct FieldMetadata {
    std::string name;
    std::string typeSchema;
};

class HeaderRecord {
public:
    HeaderRecord();

    bool writeHeader(std::ofstream& file);

    bool readHeader(const std::string& filename);

    // Setters
    void setFileStructureType(const std::string& type) { fileStructureType = type; }
    void setVersion(const std::string& ver) { version = ver; }
    void setBlockSize(int size) { blockSize = size; }
    void setMinBlockCapacity(double capacity) { minBlockCapacity = capacity; }
    void setIndexFileName(const std::string& name) { indexFileName = name; }
    void setIndexSchema(const std::string& schema) { indexFileSchema = schema; }
    void setPrimaryKeyField(int field) { primaryKeyField = field; }
    void setAvailListRBN(int rbn) { availListRBN = rbn; }
    void setActiveListRBN(int rbn) { activeListRBN = rbn; }
    void setStaleFlag(bool flag) { isStale = flag; }
    void addField(const std::string& name, const std::string& schema);

    // Getters
    std::string getFileStructureType() const { return fileStructureType; }
    std::string getVersion() const { return version; }
    int getBlockSize() const { return blockSize; }
    double getMinBlockCapacity() const { return minBlockCapacity; }
    std::string getIndexFileName() const { return indexFileName; }
    std::string getIndexSchema() const { return indexFileSchema; }
    int getPrimaryKeyField() const { return primaryKeyField; }
    int getAvailListRBN() const { return availListRBN; }
    int getActiveListRBN() const { return activeListRBN; }
    bool getStaleFlag() const { return isStale; }
    const std::vector<FieldMetadata>& getFields() const { return fields; }

private:
    std::string fileStructureType;
    std::string version;
    int headerSize;
    int recordSizeBytes;
    std::string sizeFormatType;
    int blockSize;
    double minBlockCapacity;
    std::string indexFileName;
    std::string indexFileSchema;
    int recordCount;
    int blockCount;
    int fieldCount;
    std::vector<FieldMetadata> fields;
    int primaryKeyField;
    int availListRBN;
    int activeListRBN;
    bool isStale;
};

#endif // HEADER_RECORD_H
```

4.13 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderTest.cpp File Reference

4.14 HeaderTest.cpp

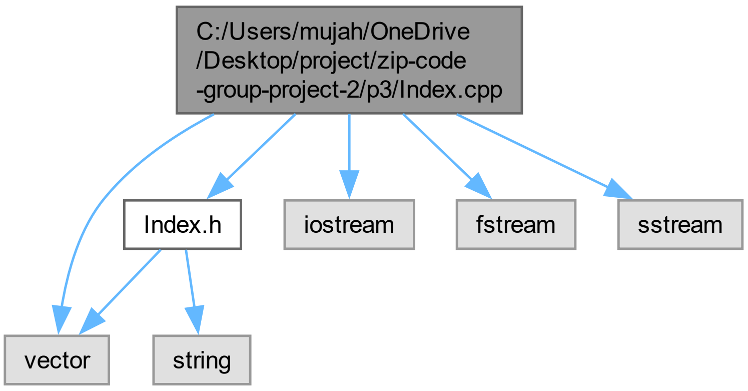
Go to the documentation of this file.

```
// #include "HeaderRecord.h"
// #include <iostream>
// #include <fstream>
// #include <string>
// #include <vector>

// /**
//  * @brief Creates a sample CSV file with zip code data
//  * @param filename Name of the file to create
//  * @return true if successful, false otherwise
//  */
// bool createSampleCSV(const std::string& filename) {
//     std::ofstream file(filename);
//     if (!file.is_open()) {
//         std::cerr « "Error: Unable to create sample CSV file: " « filename « std::endl;
//         return false;
//     }

//     // Write header row
//     file « "zip_code,city,state,latitude,longitude\n";

//     // Write some sample data
//     std::vector<std::string> sampleData = {
//         "12345,Springfield,IL,39.7817,-89.6501",
//         "23456,Riverside,CA,33.9533,-117.3961",
//         "34567,Lakewood,OH,41.4819,-81.7984",
//         "45678,Maplewood,MN,44.9530,-93.0275",
//         "56789,Oakland,CA,37.8044,-122.2711"
//     };

//     for (const auto& record : sampleData) {
//         file « record « "\n";
//     }

//     file.close();
//     return true;
// }

// /**
//  * @brief Creates a new file with header and copies CSV data
//  * @param csvFile Original CSV file
//  * @param outputFile Output file with header
//  * @param header HeaderRecord object
//  * @return true if successful, false otherwise
//  */
// bool createFileWithHeader(const std::string& csvFile, const std::string& outputFile, HeaderRecord&
header) {
//     // First write the header
//     if (!header.writeHeader(outputFile)) {
//         std::cerr « "Failed to write header to output file" « std::endl;
//         return false;
//     }
//     std::ofstream file(outputFile);
//     if (file.is_open()) {
//         if (!header.writeHeader(file)) {
//             // Handle error
//         }
//     // Continue using the file stream for other operations
//     file.close();

//     // Now append the CSV data
//     std::ifstream inFile(csvFile);
//     std::ofstream outFile(outputFile, std::ios::app);  // Open in append mode

//     if (!inFile.is_open() || !outFile.is_open()) {
//         std::cerr « "Error opening files for copying data" « std::endl;
//         return false;
//     }

//     std::string line;
//     getline(inFile, line);  // Skip the CSV header

//     // Copy the rest of the file
//     while (getline(inFile, line)) {
//         outFile « line « "\n";
//     }

//     inFile.close();
//     outFile.close();
//     return true;
// }

// int main() {
//     const std::string csvFilename = "headerTest.csv";
//     const std::string outputFilename = "headerTest_with_header.dat";

//     // Create sample CSV file
//     std::cout « "Creating sample CSV file..." « std::endl;
//     if (!createSampleCSV(csvFilename)) {
//         std::cerr « "Failed to create sample CSV file" « std::endl;
//         return 1;
//     }

//     // Create and configure header
//     std::cout « "Configuring header record..." « std::endl;
//     HeaderRecord header;

//     // Set basic header information
//     header.setFileStructureType("blocked_sequence_set");
//     header.setVersion("1.0");
//     header.setBlockSize(512);  // Default block size
//     header.setMinBlockCapacity(0.5);  // 50% minimum capacity
//     header.setIndexFileName("headerTest.idx");
//     header.setIndexSchema("key:string,rbn:int");

//     // Add field definitions
//     // header.addField("zip_code", "string(5)");
//     // header.addField("city", "string(64)");
//     // header.addField("state", "string(2)");
//     // header.addField("latitude", "decimal(8,4)");
//     // header.addField("longitude", "decimal(8,4)");

//     // Set primary key field (zip_code is field 0)
//     header.setPrimaryKeyField(0);

//     // Create the output file with header
//     std::cout « "Creating output file with header..." « std::endl;
//     if (!createFileWithHeader(csvFilename, outputFilename, header)) {
//         std::cerr « "Failed to create output file with header" « std::endl;
//         return 1;
//     }

//     // Verify by reading back the header
//     std::cout « "\nVerifying header by reading it back..." « std::endl;
//     HeaderRecord readHeader;
//     if (readHeader.readHeader(outputFilename)) {
//         std::cout « "Header verification successful!\n" « std::endl;
//         std::cout « "File structure type: " « readHeader.getFileStructureType() « std::endl;
//         std::cout « "Version: " « readHeader.getVersion() « std::endl;
//         std::cout « "Block size: " « readHeader.getBlockSize() « std::endl;
//         std::cout « "Index file: " « readHeader.getIndexFileName() « std::endl;
//         std::cout « "Primary key field: " « readHeader.getPrimaryKeyField() « std::endl;
//         std::cout « "Number of fields: " « readHeader.getFields().size() « std::endl;

//         std::cout « "\nField definitions:" « std::endl;
//         const auto& fields = readHeader.getFields();
//         for (size_t i = 0; i < fields.size(); i++) {
//             std::cout « i « ": " « fields[i].name « " (" « fields[i].typeSchema « ")" « std::endl;
//         }
//     } else {
//         std::cerr « "Failed to read back header" « std::endl;
//         return 1;
//     }

//     std::cout « "\nTest completed successfully!" « std::endl;
//     return 0;
// }
```

4.15 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.cpp File Reference

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

Include dependency graph for Index.cpp:



4.16 Index.cpp

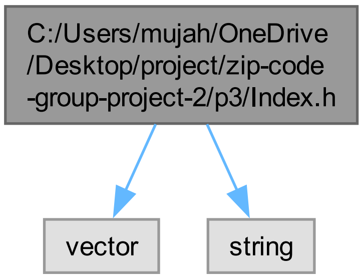
Go to the documentation of this file.

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

using namespace std;

vector<string> Index::split( const string& line, char delimiter ) {
  vector<string> tokens;
  string token;
  stringstream ss( line );
  while ( getline( ss, token, delimiter ) ) {
    tokens.push_back( token );
  }
  return tokens;
}

void Index::processBlockData( const string& inputFileName, const string& outputFileName ) {
  ifstream inputFile( inputFileName );
  if ( !inputFile.is_open() ) {
    cerr « "Error: Could not open " « inputFileName « endl;
    return;
  }

  ofstream outputFile( outputFileName );
  if ( !outputFile.is_open() ) {
    cerr « "Error: Could not open " « outputFileName « endl;
    return;
  }

  outputFile « "Block,Zip Code\n";
  string line;
  getline( inputFile, line );
  line = "";
  while ( getline( inputFile, line ) ) {
    if ( line.empty() ) continue;

    // Check if the line contains a colon; if not, skip it
    size_t colonPos = line.find( ':' );
    if ( colonPos == string::npos ) {
      // If no colon, perform additional validation
      vector<string> fields = split( line, ',' );
      if ( fields.size() < 2 || !isdigit( fields[ 0 ][ 0 ] ) ) {
        // Skip malformed lines
        continue;
      }
    }
    else {
      // Process lines with valid block:data format
      string block = line.substr( 0, colonPos ); // Block number
      string data = line.substr( colonPos + 1 ); // Rest of the data

      // Split data into fields
      vector<string> fields = split( data, ',' );

      // Extract zip codes (skip 5 fields for each)
      for ( size_t i = 0; i < fields.size(); i += 6 ) {
        if ( !fields[ i ].empty() && isdigit( fields[ i ][ 0 ] ) ) {
          string zipCode = fields[ i ];
          outputFile « zipCode « "," « block « "\n";
        }
      }
    }
  }

  inputFile.close();
  outputFile.close();

  cout « "Data successfully organized and saved to '" « outputFileName « "'.\n";
}

```

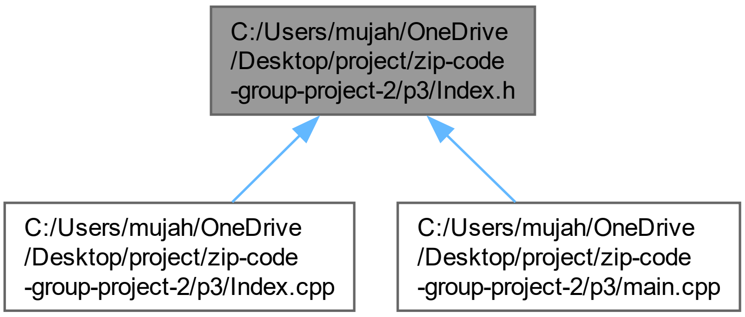
4.17 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.h File Reference

```
#include <vector>
#include <string>
```

Include dependency graph for Index.h:



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


C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/Index.h
C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/Index.cpp
C:/Users/mujah/OneDrive
/Desktop/project/zip-code
-group-project-2/p3/main.cpp

Classes

- class Index

4.18 Index.h

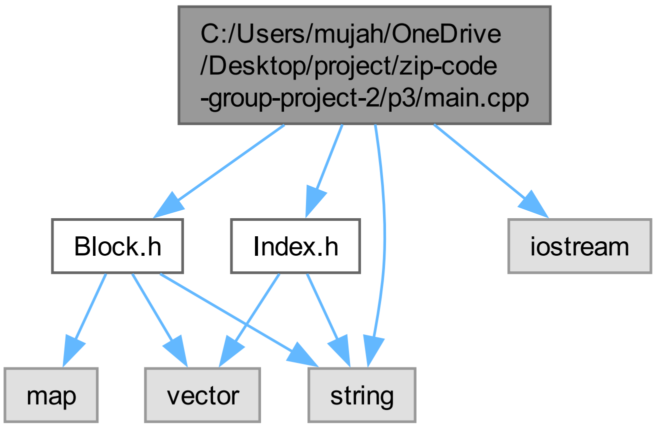
Go to the documentation of this file.

```
#ifndef INDEX_H
#define INDEX_H
#include <vector>
#include <string>

using namespace std;

class Index {
public:
  void processBlockData( const string& inputFileName, const string& outputFileName );
  std::vector<std::string> split( const std::string& line, char delimiter );
};

#endif
```

4.19 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/main.cpp File Reference

```
#include "Block.h"
#include "Index.h"
#include <iostream>
#include <string>
```

Include dependency graph for main.cpp:


C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/main.cpp
Block.h
Index.h
iostream
map
vector
string

Functions

- int main ()

  Main function to interactively manage blocks.

4.19.1 Function Documentation

4.19.1.1 main()

```
int main ()
```

Main function to interactively manage blocks.

This function provides an interactive menu-driven interface for managing and querying blocks. It performs the following steps:

1. Creates a block file from an input CSV file.
2. Parses the block file to populate the global `blocks` map.
3. Enters an infinite loop providing the user with the following options:
   - Dump all blocks in physical order.

- Dump all blocks in logical order.
- Query a specific block by its RBN.
- Exit the program.

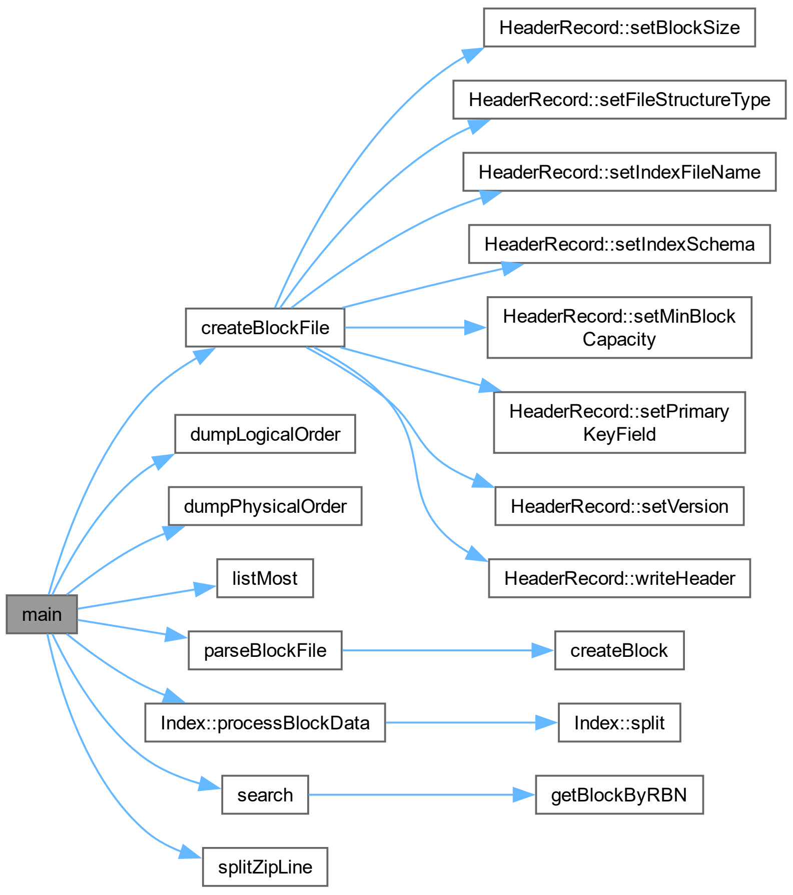
The user can query the details of a specific block by entering its RBN, including availability, records, and predecessor/successor RBNs.

Returns

int Exit code. Returns 0 if successful.

Definition at line 27 of file main.cpp.

Here is the call graph for this function:


HeaderRecord::setBlockSize
HeaderRecord::setFileStructureType
HeaderRecord::setIndexFileName
HeaderRecord::setIndexSchema
createBlockFile
HeaderRecord::setMinBlock Capacity
dumpLogicalOrder
HeaderRecord::setPrimary KeyField
dumpPhysicalOrder
HeaderRecord::setVersion
listMost
HeaderRecord::writeHeader
main
parseBlockFile
createBlock
Index::processBlockData
Index::split
search
getBlockByRBN
splitZipLine

4.20 main.cpp

Go to the documentation of this file.

```
#include "Block.h"
#include "Index.h"
```

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

using namespace std;

int main() {
    string inputFile = "us_postal_codes.csv";
    string outputFile = "block.txt";

    // Step 1: Create the block file from the input CSV
    if (createBlockFile(inputFile, outputFile)) {
        cout « "Block file created successfully.\n";
    } else {
        cerr « "Failed to create block file.\n";
        return 1;
    }

    Index index;
    index.processBlockData( outputFile, "index.idx" );
    // Step 2: Parse the block file to populate the global blocks map
    parseBlockFile(outputFile);

    // Step 3: Enter an infinite loop to provide a user menu
    while (true) {
        cout « "\n===== Block Management Menu =====\n";
        cout « "1. Dump Blocks in Physical Order\n";
        cout « "2. Dump Blocks in Logical Order\n";
        cout « "3. Query a Block by RBN\n";
        cout « "4. Get the most of each state.\n";
        cout « "5. Search for several zip codes.\n";
        cout « "6. Exit\n";

        cout « "Enter your choice: ";

        int choice;
        cin » choice;

        switch (choice) {
            case 1:
                cout « "\n----- Physical Order Dump -----\n";
                dumpPhysicalOrder();
                break;

            case 2:
                cout « "\n----- Logical Order Dump -----\n";
                dumpLogicalOrder();
                break;

            case 3: {
                cout « "\nEnter the RBN of the block you want to query: ";
                int RBN;
                cin » RBN;

                if (blocks.find(RBN) != blocks.end()) {
                    const Block& block = blocks[RBN];
                    cout « "\nDetails of Block RBN " « RBN « ":\n";
                    cout « "Available: " « (block.isAvailable ? "Yes" : "No") « "\n";
                    cout « "Records: ";
                    for (const string& record : block.records) {
                        cout « record « " ";
                    }
                    cout « "\nPredecessor RBN: " « RBN-1 « "\n";
                    cout « "Successor RBN: " « RBN+1 « "\n";
                } else {
                    cout « "\nError: Block with RBN " « RBN « " not found.\n";
                }
                break;
            }

            case 4: {
                listMost();
                cout « "\n----- State Most Data -----\n";
                break;
            }


            case 5: {
                cout « "Please enter the zip codes you want!" « endl;
        std::string text;
        cin » text;
        auto result = splitZipLine(text);
        for (const auto& str : result) {
        search(str, "index.idx");
    }
        break;
            }

            case 6:{
                cout « "Exiting the program. Goodbye!\n";
                return 0;
            }

            default:
                cout « "Invalid choice. Please try again.\n";
                break;
        }
    }
}
```

Index