CSCI331 - Project 4

Generated by Doxygen 1.9.5

Chapter 1

readme

Project 4 Instructions

Compiling Compile with g++ –std=c++17 -Wall -o ZipCode.out ∗.cpp -lstdc++fs

(-lstdc++fs links the filesystem library and must be at the end of the call to g++)

Running There are two optional arguments that will change the way the program functions.

-Z<list_of_zip_codes> where list_of_zip_codes is a comma separated list of zip codes to search the file for

-C<csv_filename> where csv_filename is the name of a csv file of the type used in this project.

The only required argument is a path to a file.

If -C is used, the program will try to convert the csv file to the lirf format and write it to the lirf path provided.
If -Z is used, the program will load the file from the path provided and search for the zip codes.
If both -C and -Z are used, the program will currently act as if only -C was used. (This might change later)

These options can appear in any order after the executable name.

Examples:

./ZipCode.out -Z24321,42444 example_file
will try to open and read a file called example_file and search for the zip codes 24321 and 42444. Any zip codes that are found in the file will have their entire record printed in a table. The zip codes that are not found will be listed after any that were found.

1.0.1 For example, using these two zip codes on the file provided, the output will look like

1.0.2 Zip Place NameState County Latitude Longitude

1.0.3 42444 Poole KY Webster 37.641 -87.6439

The following zip codes did not match any records in the file: 24321

./ZipCode.out -Cexample_csv_file.csv example_file
will try to open and read a file called example_csv_file.csv and will create or overwrite a file called example_file with the data read from the csv file.

./ZipCode.out example_file
without either optional argument, the program will simply try to read example_file as a file and do nothing.

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 CsvBuffer Class Reference

Public Member Functions

- **CsvBuffer** (const char delim=',')
  Construct a new Csv Buffer object.
- bool **read** (std::istream &instream)
  Reads one record into the buffer if there is data left in the stream.
- bool **unpack** (std::string &str)
  Reads a field and puts it into a string.
- void **init** (std::istream &instream)
  Performs the first read and extracts the headers.
- std::pair< HeaderField, std::string > **getCurFieldHeader** ()
  Gets the type and value of the current field.
- std::vector< std::pair< HeaderField, std::string > > **getHeaders** () const

4.1.1 Detailed Description

Definition at line **13** of file **CsvBuffer.h**.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 CsvBuffer()

```
CsvBuffer::CsvBuffer (
            const char delim = ',' )
```

Construct a new Csv Buffer object.

Parameters

delim	The delimiter used in the csv file

Definition at line **9** of file **CsvBuffer.cpp**.

```
00009 :  delim(delim){};
```

4.1.3 Member Function Documentation

4.1.3.1 getCurFieldHeader()

```
std::pair< HeaderField, std::string > CsvBuffer::getCurFieldHeader ( )
```

Gets the type and value of the current field.

Precondition

headers has been initialized

Postcondition

returns a pair containing the HeaderField type and the string value of the current field's header

Returns

std::pair<HeaderField, std::string>

Definition at line **88** of file **CsvBuffer.cpp**.

```
00088                                                    {
00089     return headers[fieldNum];
00090 }
```

Here is the caller graph for this function:

4.1.3.2 getHeaders()

```
std::vector< std::pair< HeaderField, std::string > > CsvBuffer::getHeaders ( ) const
```

Definition at line **127** of file **CsvBuffer.cpp**.

```
00127                                                                 {
00128     return headers;
00129 }
```

4.1.3.3 init()

```
void CsvBuffer::init (
            std::istream & instream )
```

Performs the first read and extracts the headers.

Parameters

in	*instream*	stream to be read from

Precondition

buffer is empty

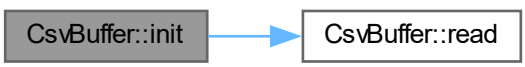
Postcondition

headers contains the values returned by readHeader
buffer contains one unprocessed record
curr points to the start of the record.
fieldNum is increased by one if the record contains more fields or is set to zero if the entire record has been read.

Definition at line **11** of file **CsvBuffer.cpp**.

```
00011                                          {
00012     read(instream);
00013     readHeader();
00014 }
```

Here is the call graph for this function:



4.1.3.4 read()

```
bool CsvBuffer::read (
            std::istream & instream )
```

Reads one record into the buffer if there is data left in the stream.

Parameters

in	*instream*	one record will be read

Precondition

instream is an open stream that contains data in a CSV format

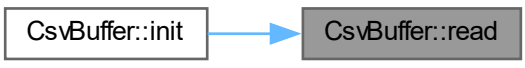
Postcondition

buffer contains data to be unpacked
instream points to next record or end of stream

Definition at line **16** of file **CsvBuffer.cpp**.

```
00016                                                {
00017     bool inQuotes = false;
00018     bool endOfFile = false;
00019
00020     curr = 0;
00021     buffer.clear();
00022
00023     char c = 0;
00024     while (!instream.eof()) {
00025         endOfFile = instream.get(c).eof();  // will be set to true if we try to read beyond the end of
      the file
00026
00027         if (c == '\r' && instream.peek() == '\n' && !inQuotes) {
00028             continue;
00029         } else if (c == '\n' && !inQuotes) {
00030             buffer.push_back(c);
00031             break;
00032         } else if (c == '"') {
00033             inQuotes = !inQuotes;
00034         }
00035
00036         buffer.push_back(c);
00037     }
00038
00039     return !endOfFile;
00040 }
```

Here is the caller graph for this function:



4.1.3.5 unpack()

```
bool CsvBuffer::unpack (
            std::string & str )
```

Reads a field and puts it into a string.

Parameters

out	*str*	the string that will hold the value of the field

Returns

true record has not had every field unpacked

false record has no more fields to unpack

Precondition

curr is pointing to the start of a field str is an empty std::string

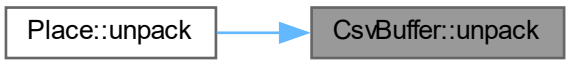
Postcondition

str contains the value of the field curr is pointing to the start of the next field or the end of the record

Definition at line **42** of file **CsvBuffer.cpp**.

```
00042                                       {
00043     auto state = CSVState::UnquotedField;  // assume field is not quoted by default
00044
00045     bool fieldHasMore = true;
00046     bool recordHasMore = true;
00047     while (fieldHasMore) {
00048         char c = buffer[curr];
00049         switch (state) {
00050             case CSVState::UnquotedField:
00051                 if (c == delim) {
00052                     fieldHasMore = false;
00053                     fieldNum++;
00054                 } else if (c == '\n') {
00055                     fieldHasMore = false;
00056                     recordHasMore = false;
00057                     fieldNum = 0;
00058                 } else if (c == '"') {
00059                     state = CSVState::QuotedField;
00060                 } else {
00061                     str.push_back(c);
00062                 }
00063                 break;
00064             case CSVState::QuotedField:
00065                 if (c == '"') {
00066                     state = CSVState::QuotedQuote;
00067                 } else {
00068                     str.push_back(c);
00069                 }
00070                 break;
00071             case CSVState::QuotedQuote:
00072                 if (c == delim) {
00073                     fieldHasMore = false;
00074                     fieldNum++;
00075                 } else if (c == '"') {
00076                     str.push_back(c);
00077                     state = CSVState::QuotedField;
00078                 } else {
00079                     state = CSVState::UnquotedField;
00080                 }
00081                 break;
00082         }
00083         curr++;
00084     }
00085     return recordHasMore;
00086 }
```

Here is the caller graph for this function:



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

- CsvBuffer.h
- CsvBuffer.cpp

4.2 FieldInfo Struct Reference

Public Attributes

- char **fieldName** [50]
- HeaderField **fieldType**

  the name of the field

Friends

- std::istream & **operator**>> (std::istream &ins, **FieldInfo** &fieldInfo)

  the HeaderField type of the field
- std::ostream & **operator**<< (std::ostream &os, **FieldInfo** &fieldInfo)

4.2.1 Detailed Description

Definition at line **62** of file **Header.h**.

4.2.2 Friends And Related Function Documentation

4.2.2.1 operator<<

```
std::ostream & operator<< (
             std::ostream & os,
              FieldInfo & fieldInfo )  [friend]
```

Definition at line **71** of file **Header.h**.

```
00071                                                                    {
00072         os.write(reinterpret_cast<char*>(&fieldInfo), sizeof(fieldInfo));
00073         return os;
00074     }
```

4.2.2.2 operator>>

```
std::istream & operator>> (
             std::istream & ins,
              FieldInfo & fieldInfo )  [friend]
```

the HeaderField type of the field

Definition at line **66** of file **Header.h**.

```
00066                                                                    {
00067         ins.read((char*)(&fieldInfo), sizeof(fieldInfo));
00068         return ins;
00069     }
```

4.2.3 Member Data Documentation

4.2.3.1 fieldName

```
char FieldInfo::fieldName[50]
```

Definition at line **63** of file **Header.h**.

4.2.3.2 fieldType

```
HeaderField FieldInfo::fieldType
```

the name of the field

Definition at line **64** of file **Header.h**.

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

- Header.h

4.3 FileInfo Struct Reference

Public Attributes

- int **lengthIndicatorSize**
- LengthIndicatorType **lengthIndicatorFormat**
  number of bytes in length indicator
- int **fieldsPerRecord**
  ASCII, BINARY, or BCD.
- int **primaryKeyPosition**
  number of fields in each record
- char **indexFileName** [100]
  the ordinal position of the primary key used to index the file

Friends

- std::istream & **operator**>> (std::istream &ins, **FileInfo** &fileInfo)
  the name of the index file to be loaded at program start
- std::ostream & **operator**<< (std::ostream &os, **FileInfo** &fileInfo)

4.3.1 Detailed Description

Definition at line **35** of file **Header.h**.

4.3.2 Friends And Related Function Documentation

4.3.2.1 operator<<

```
std::ostream & operator<< (
             std::ostream & os,
              FileInfo & fileInfo )  [friend]
```

Definition at line **49** of file **Header.h**.

```
00049                                                                    {
00050         os.write(reinterpret_cast<char*>(&fileInfo.lengthIndicatorSize),
      sizeof(fileInfo.lengthIndicatorSize));
00051         os.write(reinterpret_cast<char*>(&fileInfo.lengthIndicatorFormat),
      sizeof(fileInfo.lengthIndicatorFormat));
00052
00053         os.write(reinterpret_cast<char*>(&fileInfo.fieldsPerRecord),
      sizeof(fileInfo.fieldsPerRecord));
00054         os.write(reinterpret_cast<char*>(&fileInfo.primaryKeyPosition),
      sizeof(fileInfo.primaryKeyPosition));
00055
00056         os.write(reinterpret_cast<char*>(&fileInfo.indexFileName), sizeof(fileInfo.indexFileName));
00057
00058         return os;
00059     }
```

4.3.2.2 operator>>

```
std::istream & operator>> (
             std::istream & ins,
              FileInfo & fileInfo )  [friend]
```

the name of the index file to be loaded at program start

Definition at line **44** of file **Header.h**.

```
00044                                                                    {
00045         ins.read((char*)(&fileInfo), sizeof(fileInfo));
00046         return ins;
00047     }
```

4.3.3 Member Data Documentation

4.3.3.1 fieldsPerRecord

```
int FileInfo::fieldsPerRecord
```

ASCII, BINARY, or BCD.

Definition at line **39** of file **Header.h**.

4.3.3.2 indexFileName

```
char FileInfo::indexFileName[100]
```

the ordinal position of the primary key used to index the file

Definition at line **42** of file **Header.h**.

4.3.3.3 lengthIndicatorFormat

```
LengthIndicatorType FileInfo::lengthIndicatorFormat
```

number of bytes in length indicator

Definition at line **37** of file **Header.h**.

4.3.3.4 lengthIndicatorSize

```
int FileInfo::lengthIndicatorSize
```

Definition at line **36** of file **Header.h**.

4.3.3.5 primaryKeyPosition

```
int FileInfo::primaryKeyPosition
```

number of fields in each record

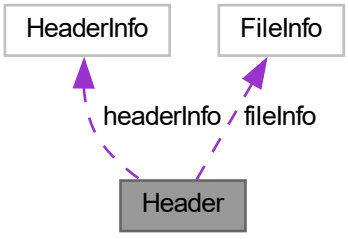
Definition at line **40** of file **Header.h**.

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

- Header.h

4.4 Header Struct Reference

Collaboration diagram for Header:



Public Attributes

- **HeaderInfo headerInfo**
- **FileInfo fileInfo**
- std::vector< **FieldInfo** > **fields**

Friends

- std::ostream & **operator**<< (std::ostream &os, **Header** &header)

4.4.1 Detailed Description

Definition at line **77** of file **Header.h**.

4.4.2 Friends And Related Function Documentation

4.4.2.1 operator<<

```
std::ostream & operator<< (
             std::ostream & os,
              Header & header )  [friend]
```

Definition at line **82** of file **Header.h**.

```
00082                                                          {
00083         os « header.headerInfo;
00084         os « header.fileInfo;
00085
00086         for (auto f :  header.fields) {
00087             os « f;
00088         }
00089         return os;
00090     }
```

4.4.3 Member Data Documentation

4.4.3.1 fields

```
std::vector< FieldInfo> Header::fields
```

Definition at line **80** of file **Header.h**.

4.4.3.2 fileInfo

```
FileInfo Header::fileInfo
```

Definition at line **79** of file **Header.h**.

4.4.3.3 headerInfo

```
HeaderInfo Header::headerInfo
```

Definition at line **78** of file **Header.h**.

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

- Header.h

4.5 HeaderBuffer Class Reference

Public Member Functions

- void **read** (std::istream &ins)
  Read header file into buffer.
- **Header unpack** ()
  *Unpacks header fields from buffer into **Header*** (p. **??**) *object.*

4.5.1 Detailed Description

Definition at line **12** of file **HeaderBuffer.h**.

4.5.2 Member Function Documentation

4.5.2.1 read()

```
void HeaderBuffer::read (
            std::istream & ins )
```

Read header file into buffer.

Parameters

in	*ins*	stream to be read in from

Precondition

ins is pointing to an open length indicated file and ins.good() is true

Postcondition

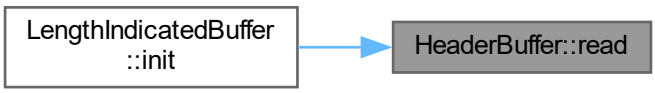
ins is positioned at the first character after the header

buffer is filled with header bytes

Definition at line **7** of file **HeaderBuffer.cpp**.

```
00007                                           {
00008     ins.seekg(0);
00009
00010     HeaderInfo hInfo;
00011     ins » hInfo;
00012
00013     buffer.resize(hInfo.headerSize);
00014     ins.seekg(0);
00015
00016     char c;
00017     for (int i = 0; i < hInfo.headerSize; i++) {
00018         ins.read(&c, 1);
00019         buffer[i] = c;
00020     }
00021 }
```

Here is the caller graph for this function:



4.5.2.2 unpack()

Header HeaderBuffer::unpack ( )

Unpacks header fields from buffer into **Header** (p. **??**) object.

Precondition

read() (p. **??**) has been called on a valid file and buffer contains a header with the **Header** (p. **??**) format

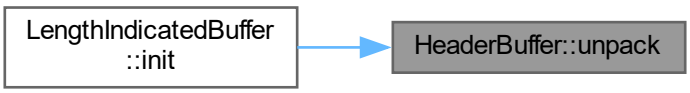
Returns

a **Header** (p. **??**) object loaded with the information in the file

Definition at line **23** of file **HeaderBuffer.cpp**.

```
00023                                {
00024     Header header;
00025     std::vector<FieldInfo> fields;
00026
00027     auto fileInfoOffset = sizeof(Header::headerInfo);
00028
00029     memcpy(&header.headerInfo, &buffer[0], sizeof(HeaderInfo));
00030     memcpy(&header.fileInfo, &buffer[fileInfoOffset], sizeof(FileInfo));
00031
00032     size_t fieldInfoOffset = fileInfoOffset + sizeof(Header::fileInfo);
00033     for (int i = 0; i < header.fileInfo.fieldsPerRecord; i++) {
00034         FieldInfo fieldInfo;
00035         memcpy(&fieldInfo, &buffer[fieldInfoOffset], sizeof(FieldInfo));
00036         fields.push_back(fieldInfo);
00037
00038         // set offset to beginning of next field info
00039         fieldInfoOffset += sizeof(FieldInfo);
00040     }
00041
00042     header.fields = fields;
00043
00044     return header;
00045 }
```

Here is the caller graph for this function:



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

- HeaderBuffer.h
- HeaderBuffer.cpp

4.6 HeaderInfo Struct Reference

Public Attributes

- char **magic** [4]
- int **version**

  4 bytes at the start indicating that the file is of the correct type
- int **headerSize**

  version number

Friends

- std::istream & **operator**>> (std::istream &ins, **HeaderInfo** &headerInfo)

  size of header in bytes, including header info
- std::ostream & **operator**<< (std::ostream &os, **HeaderInfo** &headerInfo)

4.6.1 Detailed Description

Definition at line **17** of file **Header.h**.

4.6.2 Friends And Related Function Documentation

4.6.2.1 operator<<

```
std::ostream & operator<< (
            std::ostream & os,
             HeaderInfo & headerInfo )  [friend]
```

Definition at line **27** of file **Header.h**.

```
00027                                                                   {
00028         os.write(reinterpret_cast<char*>(&headerInfo.magic), sizeof(headerInfo.magic));
00029         os.write(reinterpret_cast<char*>(&headerInfo.version), sizeof(headerInfo.version));
00030         os.write(reinterpret_cast<char*>(&headerInfo.headerSize), sizeof(headerInfo.headerSize));
00031         return os;
00032     }
```

4.6.2.2 operator>>

```
std::istream & operator>> (
            std::istream & ins,
             HeaderInfo & headerInfo )  [friend]
```

size of header in bytes, including header info

Definition at line **22** of file **Header.h**.

```
00022                                                                   {
00023         ins.read((char*)(&headerInfo), sizeof(headerInfo));
00024         return ins;
00025     }
```

4.6.3 Member Data Documentation

4.6.3.1 headerSize

```
int HeaderInfo::headerSize
```

version number

Definition at line **20** of file **Header.h**.

4.6.3.2 magic

```
char HeaderInfo::magic[4]
```

Definition at line **18** of file **Header.h**.

4.6.3.3 version

```
int HeaderInfo::version
```

4 bytes at the start indicating that the file is of the correct type

Definition at line **19** of file **Header.h**.

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

- Header.h

4.7 PrimaryKey::IndexFileHeader Struct Reference

Public Attributes

- int **version**
- int **keyCount**
- int **format**

4.7.1 Detailed Description

Definition at line **18** of file **PrimaryKey.h**.

4.7.2 Member Data Documentation

4.7.2.1 format

```
int PrimaryKey::IndexFileHeader::format
```

Definition at line **21** of file **PrimaryKey.h**.

4.7.2.2 keyCount

```
int PrimaryKey::IndexFileHeader::keyCount
```

Definition at line **20** of file **PrimaryKey.h**.

4.7.2.3 version

```
int PrimaryKey::IndexFileHeader::version
```

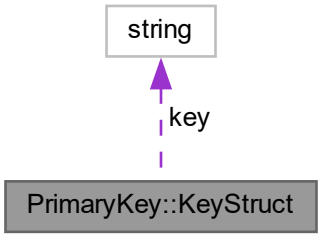
Definition at line **19** of file **PrimaryKey.h**.

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

- PrimaryKey.h

4.8 PrimaryKey::KeyStruct Struct Reference

Collaboration diagram for PrimaryKey::KeyStruct:



Public Attributes

- std::string **key**
- unsigned int **offset**

4.8.1 Detailed Description

Definition at line **23** of file **PrimaryKey.h**.

4.8.2 Member Data Documentation

4.8.2.1 key

```
std::string PrimaryKey::KeyStruct::key
```

Definition at line **24** of file **PrimaryKey.h**.

4.8.2.2 offset

```
unsigned int PrimaryKey::KeyStruct::offset
```

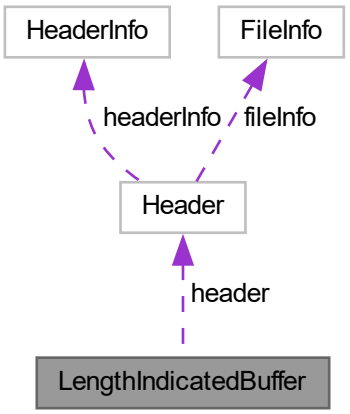
Definition at line **25** of file **PrimaryKey.h**.

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

- PrimaryKey.h

4.9 LengthIndicatedBuffer Class Reference

Collaboration diagram for LengthIndicatedBuffer:



Public Member Functions

- **LengthIndicatedBuffer** (const char delim=',')

  *Construct a new **LengthIndicatedBuffer** (p. **??**) object.*
- bool **read** (std::istream &instream)

  Read a single length indicated record into the buffer.
- bool **read** (std::istream &instream, int indexOffset)

  Seek to the specified offset in the file and read a single length indicated record into the buffer.
- bool **unpack** (std::string &str)

  Reads a field and puts it into a string.
- void **pack** (const std::string str)

  Packs a field into the buffer.
- void **write** (std::ostream &outstream)

  Writes length of the field and the data in the buffer to the stream.
- bool **init** (std::istream &instream)

  read and extract the header.
- void **writeHeader** (std::ostream &outstream)

  Seeks to the start of the stream and writes the header member to the stream.
- void **clear** ()

  Sets curr to start of buffer.
- bool **checkFileType** (std::istream &instream)

  read the first 4 bytes of the file and check against the magic number.
- std::string **getIndexFileName** ()

  get the name of the index file from the header
- **FieldInfo** **getCurFieldHeader** ()

  Gets the type and value of the current field.

Public Attributes

- **Header** **header**

4.9.1 Detailed Description

Definition at line **15** of file **LengthIndicatedBuffer.h**.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 LengthIndicatedBuffer()

```
LengthIndicatedBuffer::LengthIndicatedBuffer (
            const char delim = ',' )
```

Construct a new **LengthIndicatedBuffer** (p. **??**) object.

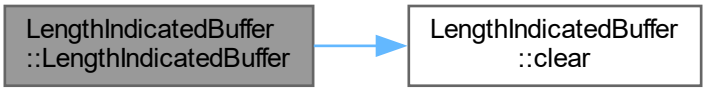
Parameters

delim	The delimiter used between the record fields

Definition at line **15** of file **LengthIndicatedBuffer.cpp**.

```
00015                                                                   :  delim(delim) {
00016     clear();
00017     memset(buffer, 0, sizeof(buffer));
00018 }
```

Here is the call graph for this function:



4.9.3 Member Function Documentation

4.9.3.1 checkFileType()

```
bool LengthIndicatedBuffer::checkFileType (
             std::istream & instream )
```

read the first 4 bytes of the file and check against the magic number.

Parameters

in	*instream*	stream to be read from

Precondition

instream is open for reading

Return values

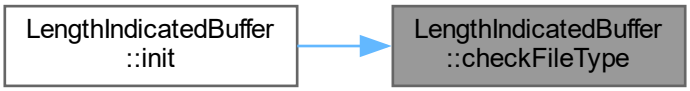
true	if file has correct magic number
false	if file does not have correct magic number

Definition at line **24** of file **LengthIndicatedBuffer.cpp**.

```
00024                                                                   {
00025     instream.seekg(0);
00026     char first4[4];
```

```
00027
00028     instream.read(first4, sizeof(first4));
00029
00030     bool good = true;
00031     for (int i = 0; i < 4; i++) {
00032         good = (first4[i] == MAGIC_HEADER_NUMBER[i]);
00033     }
00034     return good;
00035 }
```

Here is the caller graph for this function:



4.9.3.2 clear()

```
void LengthIndicatedBuffer::clear ( )
```

Sets curr to start of buffer.

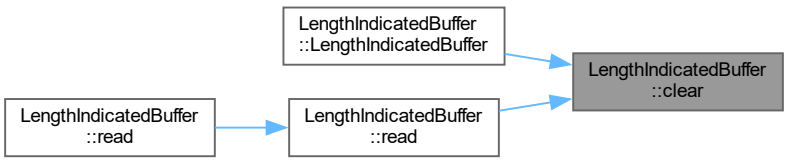
Postcondition

curr = 0

Definition at line **155** of file **LengthIndicatedBuffer.cpp**.

```
00155                                   {
00156     curr = 0;
00157 }
```

Here is the caller graph for this function:



4.9.3.3 getCurFieldHeader()

```
FieldInfo LengthIndicatedBuffer::getCurFieldHeader ( )
```

Gets the type and value of the current field.

Precondition

headers has been initialized

Postcondition

returns a **FieldInfo** (p. **??**) struct with the field name and field type

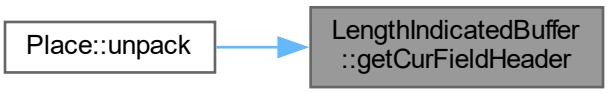
Returns

FieldInfo (p. **??**)

Definition at line **191** of file **LengthIndicatedBuffer.cpp**.

```
00191                                                   {
00192     return header.fields[fieldNum];
00193 }
```

Here is the caller graph for this function:



4.9.3.4 getIndexFileName()

```
std::string LengthIndicatedBuffer::getIndexFileName ( )
```

get the name of the index file from the header

Precondition

initialized = true

Returns

string containing index file name

Definition at line **20** of file **LengthIndicatedBuffer.cpp**.

```
00020                                                   {
00021     return header.fileInfo.indexFileName;
00022 }
```

4.9.3.5 init()

```
bool LengthIndicatedBuffer::init (
            std::istream & instream )
```

read and extract the header.

Parameters

in	*instream*	stream to be read from

Precondition

instream points to a valid length indicated file opened for reading

Postcondition

header has been loaded with the values from the stream

initialized has been set to true if header read was successful, false if it was not

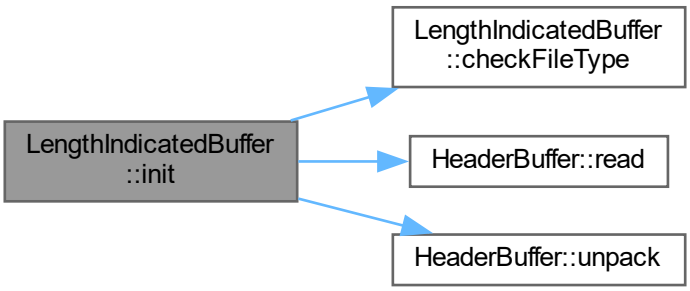
Return values

true	header read was successful
false	header read was not successful

Definition at line **37** of file **LengthIndicatedBuffer.cpp**.

```
00037                                                   {
00038     if (checkFileType(instream)) {
00039         // if file has magic number
00040         instream.seekg(0);
00041         HeaderBuffer hBuf;
00042         hBuf.read(instream);
00043         header = hBuf.unpack();
00044         initialized = true;
00045
00046     } else {
00047         // if file does not have magic number
00048         initialized = false;
00049     }
00050
00051     return initialized;
00052 }
```

Here is the call graph for this function:



4.9.3.6 pack()

```
void LengthIndicatedBuffer::pack (
            const std::string str )
```

Packs a field into the buffer.

Parameters

in	*str*	the string that will holds the value of the field

Precondition

str is of the correct type indicated by headers[fieldNum].fieldType

Postcondition

if the field is not the first, buffer has had a comma and the data from str minus the null terminator added

if the field is first, buffer has had the data from str minus the null terminator added

curr points to the first position in buffer after the newly added field

Definition at line **159** of file **LengthIndicatedBuffer.cpp**.

```
00159                                                   {
00160     // put delimiters in between each field skipping the first
00161     if (curr > 0) {
00162         buffer[curr++] = delim;
00163     }
00164
00165     // note that the size method of std::string does not include
00166     // the null terminator in the length, thus we are just copying the
00167     // values in the string to the buffer, as desired
00168     memcpy(&buffer[curr], str.c_str(), str.size());
00169
00170     // move curr pointer to the end of the field we just added
00171     curr += str.size();
00172 }
```

4.9.3.7 read() [1/2]

```
bool LengthIndicatedBuffer::read (
            std::istream & instream )
```

Read a single length indicated record into the buffer.

Parameters

in	*instream*	the stream that points to the record

Precondition

instream is an open stream pointing to the start of a length indicated record

record size $<$ sizeof(buffer)

Postcondition

instream is positioned at the start of the next record or the end of the stream

curr is set to 0 and buffer is zeroed

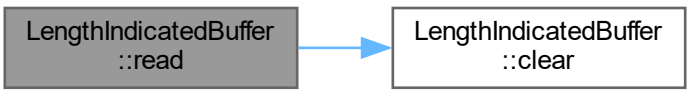
Return values

true	when instream.good() is true (indicating that we can read another record)
false	when instream.good() is false (indicating that we are probably at the end of the file)

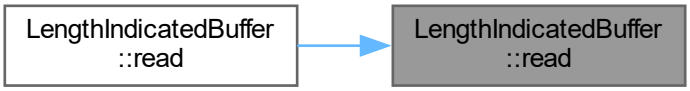
Definition at line **59** of file **LengthIndicatedBuffer.cpp**.

```
00059                                                           {
00060     clear();
00061
00062     // get length of record length indicator
00063     auto lengthIndicatorLength = header.fileInfo.lengthIndicatorSize;
00064
00065     int recordLength = 0;
00066
00067     // get actual record length
00068     switch (LengthIndicatorType(header.fileInfo.lengthIndicatorFormat)) {
00069         case LengthIndicatorType::BINARY:
00070             instream.read((char*)&recordLength, lengthIndicatorLength);
00071             break;
00072         case LengthIndicatorType::ASCII: {
00073             std::string recordLengthStr;
00074             char c;
00075             for (int i = 0; i < lengthIndicatorLength; i++) {
00076                 instream.get(c);
00077                 if (c == EOF) {
00078                     return false;
00079                 }
00080                 recordLengthStr.push_back(c);
00081             }
00082
00083             try {
00084                 recordLength = std::stoi(recordLengthStr);
00085             } catch (std::invalid_argument& err) {
00086                 if (instream.eof()) {
00087                     return false;
00088                 }
00089                 throw err;
00090             }
00091
00092             break;
00093         }
00094         case LengthIndicatorType::BCD:
00095             break;
00096     }
00097
00098     // read record
00099     instream.read(buffer, recordLength);
00100     this->recordLength = recordLength;
00101     return instream.good();
00102 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.9.3.8 read() [2/2]

```
bool LengthIndicatedBuffer::read (
            std::istream & instream,
            int indexOffset )
```

Seek to the specified offset in the file and read a single length indicated record into the buffer.

Parameters

in	*instream*	the stream that points to the record
in	*indexOffset*	the number of bytes from the start of the file to seek to before reading

Precondition

instream is an open stream pointing to the start of a length indicated record

indexOffset is the number of bytes from the start of the file to the start of a valid length indicated record

Postcondition

instream is positioned at the start of the next record or the end of the stream

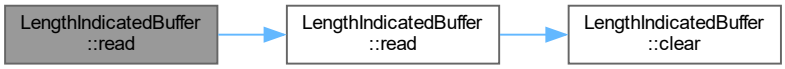
Return values

true	when instream.good() is true (indicating that we can read another record)
false	when instream.good() is false (indicating that we are probably at the end of the file)

Definition at line **54** of file **LengthIndicatedBuffer.cpp**.

```
00054                                                                {
00055     instream.seekg(indexOffset);
00056     return read(instream);
00057 }
```

Here is the call graph for this function:



4.9.3.9 unpack()

```
bool LengthIndicatedBuffer::unpack (
            std::string & str )
```

Reads a field and puts it into a string.

Parameters

out	*str*	the string that will hold the value of the field

Precondition

curr is pointing to the start of a field str is an empty std::string

Postcondition

str contains the value of the field

curr is pointing to the start of the next field or the next record

fieldNum is pointing to the type of the next field

Return values

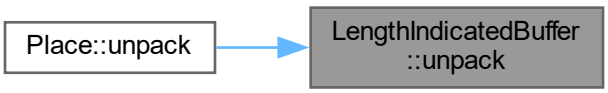
true	record has not had every field unpacked
false	record has no more fields to unpack

Definition at line **109** of file **LengthIndicatedBuffer.cpp**.

```
00109                                                   {
00110     auto state = CSVState::UnquotedField;  // assume field is not quoted by default
00111
00112     bool fieldHasMore = true;
00113     bool recordHasMore = true;
00114     while (fieldHasMore) {
00115         char c = buffer[curr];
00116         switch (state) {
00117             case CSVState::UnquotedField:
00118                 if (c == delim) {
00119                     fieldHasMore = false;
00120                     fieldNum++;
00121                 } else if (curr >= recordLength) {
00122                     fieldHasMore = false;
00123                     recordHasMore = false;
```

```
00124                     fieldNum = 0;
00125                 } else if (c == '"') {
00126                     state = CSVState::QuotedField;
00127                 } else {
00128                     str.push_back(c);
00129                 }
00130                 break;
00131             case CSVState::QuotedField:
00132                 if (c == '"') {
00133                     state = CSVState::QuotedQuote;
00134                 } else {
00135                     str.push_back(c);
00136                 }
00137                 break;
00138             case CSVState::QuotedQuote:
00139                 if (c == delim) {
00140                     fieldHasMore = false;
00141                     fieldNum++;
00142                 } else if (c == '"') {
00143                     str.push_back(c);
00144                     state = CSVState::QuotedField;
00145                 } else {
00146                     state = CSVState::UnquotedField;
00147                 }
00148                 break;
00149         }
00150         curr++;
00151     }
00152     return recordHasMore;
00153 }
```

Here is the caller graph for this function:



4.9.3.10 write()

```
void LengthIndicatedBuffer::write (
            std::ostream & outstream )
```

Writes length of the field and the data in the buffer to the stream.

Parameters

in	*str*	the string that will holds the value of the field

Precondition

str is of the correct type indicated by headers[fieldNum].fieldType

Postcondition

Definition at line **174** of file **LengthIndicatedBuffer.cpp**.

```
00174                                                                {
00175     switch (LengthIndicatorType(header.fileInfo.lengthIndicatorFormat)) {
00176         case LengthIndicatorType::ASCII: {
00177             std::ostringstream lengthStream;
00178             lengthStream « std::setfill('0') « std::setw(header.fileInfo.lengthIndicatorSize) «
      (curr);
00179             auto lengthStr = lengthStream.str();
00180             outstream.write(lengthStr.c_str(), lengthStream.str().size());
00181             break;
00182         }
00183         case LengthIndicatorType::BCD:
00184         case LengthIndicatorType::BINARY:
00185             break;
00186     }
00187
00188     outstream.write(buffer, curr);
00189 }
```

4.9.3.11 writeHeader()

```
void LengthIndicatedBuffer::writeHeader (
             std::ostream & outstream )
```

Seeks to the start of the stream and writes the header member to the stream.

Postcondition

outstream is pointing to the first byte after the header

Parameters

out	*outstream*	the stream to be written to

Definition at line **104** of file **LengthIndicatedBuffer.cpp**.

```
00104                                                                {
00105     outstream.seekp(0);
00106     outstream « header;
00107 }
```

4.9.4 Member Data Documentation

4.9.4.1 header

```
Header LengthIndicatedBuffer::header
```

Definition at line **34** of file **LengthIndicatedBuffer.h**.

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

- LengthIndicatedBuffer.h
- LengthIndicatedBuffer.cpp

4.10 LengthIndicatedFile Class Reference

Public Member Functions

- **LengthIndicatedFile** (std::string fileName)
- void **initializeBuffers** ()
- void **initializeIndex** ()
- bool **openDataFile** ()
- std::optional< **Place** > **findRecord** (std::string recordKey)
- void **generateIndex** ()
- bool **indexFileExists** ()

4.10.1 Detailed Description

Definition at line **18** of file **LengthIndicatedFile.h**.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 LengthIndicatedFile()

```
LengthIndicatedFile::LengthIndicatedFile (
            std::string fileName )
```

Definition at line **7** of file **LengthIndicatedFile.cpp**.

```
00007                                                       :  fileName(fileName) {
00008     openDataFile();
00009     initializeBuffers();
00010     dataStart = header.headerInfo.headerSize;
00011     initializeIndex();
00012 }
```

4.10.2.2 ~LengthIndicatedFile()

```
LengthIndicatedFile::~LengthIndicatedFile ( )
```

Definition at line **14** of file **LengthIndicatedFile.cpp**.

```
00014                                              {
00015     file.close();
00016 }
```

4.10.3 Member Function Documentation

4.10.3.1 findRecord()

```
std::optional< Place > LengthIndicatedFile::findRecord (
            std::string recordKey )
```

Definition at line **39** of file **LengthIndicatedFile.cpp**.

```
00039                                                                    {
00040     file.clear();
00041     auto recordFound = index.BinarySearch(recordKey);
00042
00043     if (recordFound == index.notFound) {
00044         return {};
00045     }
00046     Place p;
00047     readBuf.read(file, recordFound);
00048     p.unpack(readBuf);
00049     return p;
00050 }
```

4.10.3.2 generateIndex()

```
void LengthIndicatedFile::generateIndex ( )
```

Definition at line **52** of file **LengthIndicatedFile.cpp**.

```
00052                                          {
00053     file.clear();
00054     file.seekg(dataStart);
00055
00056     auto pos = (unsigned int)file.tellg();
00057     while (readBuf.read(file)) {
00058         Place place;
00059         place.unpack(readBuf);
00060         index.Add({place.getZipCode(), pos});
00061         pos = (unsigned int)file.tellg();
00062     }
00063     index.GenerateIndexFile(header.fileInfo.indexFileName);
00064 }
```

4.10.3.3 indexFileExists()

```
bool LengthIndicatedFile::indexFileExists ( )
```

Definition at line **66** of file **LengthIndicatedFile.cpp**.

```
00066                                        {
00067     return std::filesystem::exists(header.fileInfo.indexFileName);
00068 }
```

4.10.3.4 initializeBuffers()

```
void LengthIndicatedFile::initializeBuffers ( )
```

Definition at line **18** of file **LengthIndicatedFile.cpp**.

```
00018                                           {
00019     std::ifstream dataFile(fileName);
00020     readBuf.init(dataFile);
00021     writeBuf.init(dataFile);
00022     this->header = readBuf.header;
00023     dataFile.close();
00024 }
```

4.10.3.5 initializeIndex()

```
void LengthIndicatedFile::initializeIndex ( )
```

Definition at line **26** of file **LengthIndicatedFile.cpp**.

```
00026                                          {
00027     if (indexFileExists()) {
00028         index.ReadIndexFile(header.fileInfo.indexFileName);
00029     } else {
00030         generateIndex();
00031     }
00032 }
```

4.10.3.6 openDataFile()

```
bool LengthIndicatedFile::openDataFile ( )
```

Definition at line **34** of file **LengthIndicatedFile.cpp**.

```
00034                                       {
00035     file.open(fileName, std::ios::binary | std::ios::in | std::ios::out);
00036     return file.good();
00037 }
```

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

- LengthIndicatedFile.h
- LengthIndicatedFile.cpp

4.11 Place Class Reference

Public Member Functions

- **Place** (const **Place** &loc)

  Copy constructor.
- std::string **getZipCode** () const

  Returns the Zip Code.
- std::string **getState** () const

  Returns the 2 digit State Id.
- std::string **getName** () const

  *Returns the **Place** (p. **??**) Name.*
- std::string **getCounty** () const

  Returns the County.
- double **getLat** () const

  Returns the latitude.
- double **getLongi** () const

  Returns the longitude.
- void **unpack** (**CsvBuffer** &buffer)

  Reads a record from the buffer and unpacks the fields into the class members.
- void **unpack** (**LengthIndicatedBuffer** &buffer)

  Reads a record from the buffer and unpacks the fields into the class members.
- void **pack** (**LengthIndicatedBuffer** &buffer)
- void **operator=** (const **Place** &loc)

  Assignment operator overload.
- size_t **getSize** ()

  get size of object in bytes
- void **print** ()

4.11.1 Detailed Description

Definition at line **15** of file **Place.h**.

4.11.2 Constructor & Destructor Documentation

4.11.2.1 Place() [1/2]

```
Place::Place ( )
```

Definition at line **11** of file **Place.cpp**.

```
00011             {
00012     zipcode = "";
00013     state = "";
00014     name = "";
00015     latitude = 0;
00016     county = "";
00017     longitude = 0;
00018 };
```

4.11.2.2 Place() [2/2]

```
Place::Place (
            const  Place & loc )
```

Copy constructor.

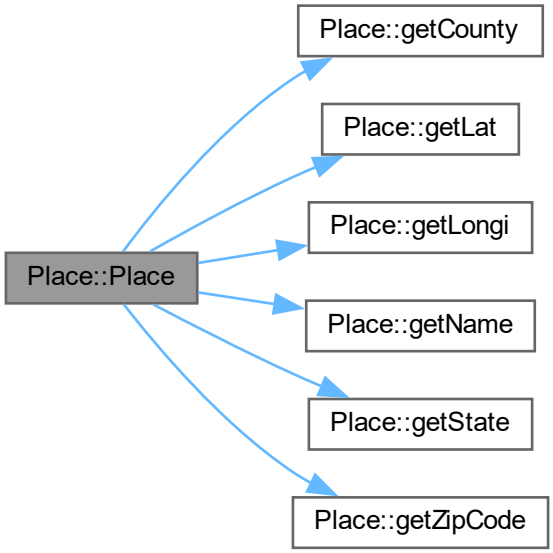
Parameters

loc	

Definition at line **21** of file **Place.cpp**.

```
00021                              {
00022     zipcode = loc.getZipCode();
00023     name = loc.getName();
00024     state = loc.getState();
00025     county = loc.getCounty();
00026     latitude = loc.getLat();
00027     longitude = loc.getLongi();
00028 };
```

Here is the call graph for this function:



4.11.3 Member Function Documentation

4.11.3.1 getCounty()

```
string Place::getCounty ( ) const
```

Returns the County.

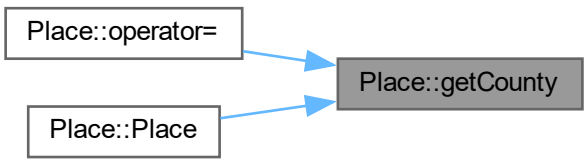
Returns

std::string

Definition at line **43** of file **Place.cpp**.

```
00043 { return county; }    // county value
```

Here is the caller graph for this function:



4.11.3.2 getLat()

```
double Place::getLat ( ) const
```

Returns the latitude.

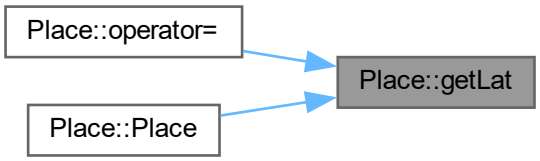
Returns

double

Definition at line **44** of file **Place.cpp**.

```
00044 { return latitude; }      // Latitude value
```

Here is the caller graph for this function:



4.11.3.3 getLongi()

```
double Place::getLongi ( ) const
```

Returns the longitude.

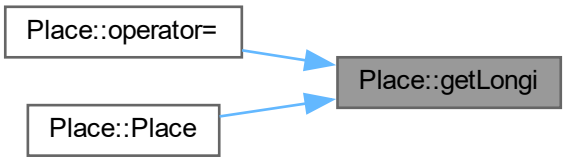
Returns

double

Definition at line **45** of file **Place.cpp**.

```
00045 { return longitude; }  // longitude value
```

Here is the caller graph for this function:



4.11.3.4 getName()

```
string Place::getName ( ) const
```

Returns the **Place** (p. **??**) Name.

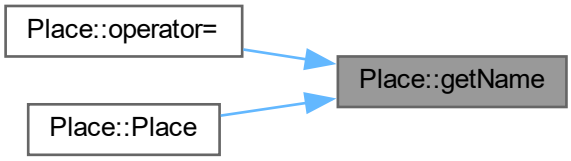
Returns

std::string

Definition at line **41** of file **Place.cpp**.

```
00041 { return name; }        // name value
```

Here is the caller graph for this function:



4.11.3.5 getSize()

```
size_t Place::getSize ( )
```

get size of object in bytes

Returns

size_t size of object

Definition at line **158** of file **Place.cpp**.

```
00158                         {
00159     size_t size = 0;
00160     size += name.size();
00161     size += zipcode.size();
00162     size += state.size();
00163     size += county.size();
00164     size += sizeof(latitude);
00165     size += sizeof(longitude);
00166     return size;
00167 }
```

4.11.3.6 getState()

```
string Place::getState ( ) const
```

Returns the 2 digit State Id.

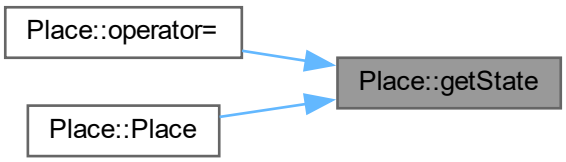
Returns

std::string

Definition at line **42** of file **Place.cpp**.

```
00042 { return state; }      // State value
```

Here is the caller graph for this function:



4.11.3.7 getZipCode()

```
string Place::getZipCode ( ) const
```

Returns the Zip Code.

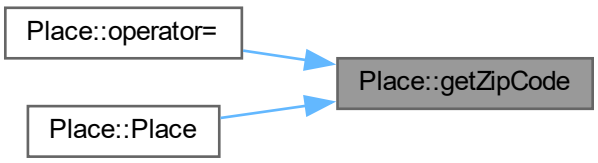
Returns

std::string

Definition at line **40** of file **Place.cpp**.

```
00040 { return zipcode; }  // zipcode value
```

Here is the caller graph for this function:



4.11.3.8 operator=()

```
void Place::operator= (
            const  Place & loc )
```

Assignment operator overload.

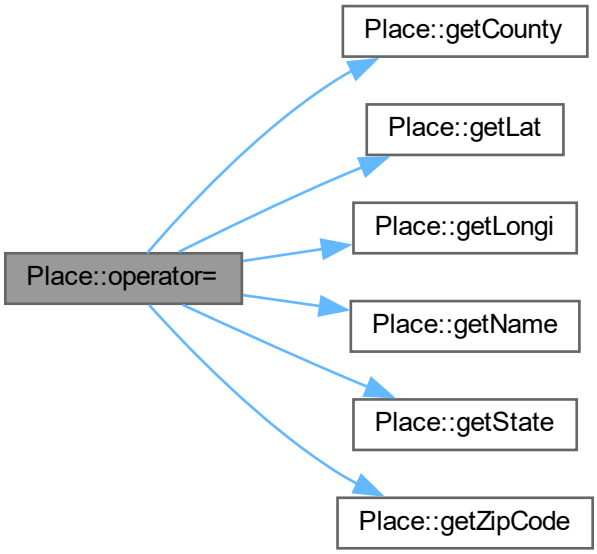
Parameters

loc	The place object that this one's parameters will match

Definition at line **31** of file **Place.cpp**.

```
00031                                   {
00032     this->zipcode = loc.getZipCode();
00033     this->name = loc.getName();
00034     this->state = loc.getState();
00035     this->county = loc.getCounty();
00036     this->latitude = loc.getLat();
00037     this->longitude = loc.getLongi();
00038 }
```

Here is the call graph for this function:



4.11.3.9 pack()

```
void Place::pack (
            LengthIndicatedBuffer & buffer )
```

Definition at line **119** of file **Place.cpp**.

```
00119                                                {
00120     buffer.clear();
00121     std::stringstream lat_strStream;
00122     std::stringstream long_strStream;
00123 
00124     lat_strStream « std::setprecision(10) « latitude;
00125     long_strStream « std::setprecision(10) « longitude;
00126 
00127     for (auto f :  buffer.header.fields) {
00128         switch (HeaderField(f.fieldType)) {
00129             case HeaderField::ZipCode:
00130                 buffer.pack(zipcode);
00131                 break;
00132             case HeaderField::PlaceName:
00133                 buffer.pack(name);
00134                 break;
00135             case HeaderField::State:
00136                 buffer.pack(state);
00137                 break;
00138             case HeaderField::County:
00139                 buffer.pack(county);
00140                 break;
00141             case HeaderField::Latitude:
00142                 buffer.pack(lat_strStream.str());
00143                 break;
00144             case HeaderField::Longitude:
00145                 buffer.pack(long_strStream.str());
00146                 break;
00147             default:
00148                 break;
00149         }
00150     }
00151 }
```

4.11.3.10 print()

```
void Place::print ( )
```

Definition at line **153** of file **Place.cpp**.

```
00153                  {
00154     std::cout « getZipCode() « " " « getName() « " " « getState() « " "
00155               « getCounty() « " " « getLat() « " " « getLongi() « " " « std::endl;
00156 }
```

4.11.3.11 unpack() [1/2]

```
void Place::unpack (
               CsvBuffer & buffer )
```

Reads a record from the buffer and unpacks the fields into the class members.

Parameters

in,out	buffer	The buffer to be read from

Precondition

buffer has a record that contains zipcode, place name, state id, county, latitude, and longitude fields

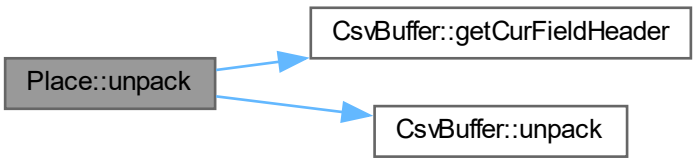
Postcondition

the member variables have been set to the values mentioned above, if the column with that name was found

Definition at line **48** of file **Place.cpp**.

```
00048                                   {
00049     std::string skip;
00050     string lat_str, long_str;
00051
00052     bool moreFields = true;
00053     while (moreFields) {
00054         auto curField = buffer.getCurFieldHeader();
00055         switch (HeaderField(curField.first)) {
00056             case HeaderField::ZipCode:
00057                 moreFields = buffer.unpack(zipcode);
00058                 break;
00059             case HeaderField::PlaceName:
00060                 moreFields = buffer.unpack(name);
00061                 break;
00062             case HeaderField::State:
00063                 moreFields = buffer.unpack(state);
00064                 break;
00065             case HeaderField::County:
00066                 moreFields = buffer.unpack(county);
00067                 break;
00068             case HeaderField::Latitude:
00069                 moreFields = buffer.unpack(lat_str);
00070                 break;
00071             case HeaderField::Longitude:
00072                 moreFields = buffer.unpack(long_str);
00073                 break;
00074             default:
00075                 moreFields = buffer.unpack(skip);
00076                 break;
00077         }
00078     }
00079     std::stringstream(lat_str) » std::setprecision(10) » latitude;    // convert to float
00080     std::stringstream(long_str) » std::setprecision(10) » longitude;  // convert to float
00081 }
```

Here is the call graph for this function:



4.11.3.12 unpack() [2/2]

```
void Place::unpack (
            LengthIndicatedBuffer & buffer )
```

Reads a record from the buffer and unpacks the fields into the class members.

Parameters

in,out	*buffer*	The buffer to be read from

Precondition

buffer has a record that contains zipcode, place name, state id, county, latitude, and longitude fields

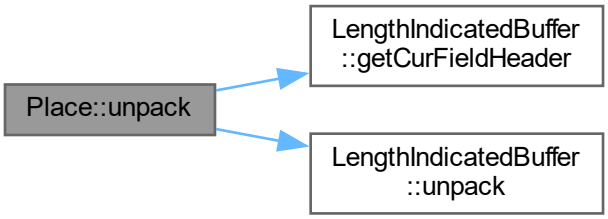
Postcondition

the member variables have been set to the values mentioned above, if the column with that name was found

Definition at line **83** of file **Place.cpp**.

```
00083                                                {
00084     std::string skip;
00085     string lat_str, long_str;
00086
00087     bool hasMore = true;
00088     while (hasMore) {
00089         auto curField = buffer.getCurFieldHeader();
00090         switch (HeaderField(curField.fieldType)) {
00091             case HeaderField::ZipCode:
00092                 hasMore = buffer.unpack(zipcode);
00093                 break;
00094             case HeaderField::PlaceName:
00095                 hasMore = buffer.unpack(name);
00096                 break;
00097             case HeaderField::State:
00098                 hasMore = buffer.unpack(state);
00099                 break;
00100             case HeaderField::County:
00101                 hasMore = buffer.unpack(county);
00102                 break;
00103             case HeaderField::Latitude:
00104                 hasMore = buffer.unpack(lat_str);
00105                 break;
00106             case HeaderField::Longitude:
00107                 hasMore = buffer.unpack(long_str);
00108                 break;
00109             default:
00110                 hasMore = buffer.unpack(skip);
00111                 break;
00112         }
00113     }
00114
00115     std::stringstream(lat_str) » std::setprecision(10) » latitude;    // convert to float
00116     std::stringstream(long_str) » std::setprecision(10) » longitude;  // convert to float
00117 }
```

Here is the call graph for this function:



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

- Place.h
- Place.cpp

4.12 PrimaryKey Class Reference

Classes

- struct **IndexFileHeader**
- struct **KeyStruct**

Public Types

- enum **IndexFileFormat** { **ASCII** , **BCD** , **BINARY** }

Public Member Functions

- void **GenerateIndexFile** (std::string fileName)

  Generates an index file from the internal index.
- bool **ReadIndexFile** (std::string fileName)

  Reads an index file and stores the values in the internal index. Also records if the index's primary keys are in ascending order, to facilitate binary search.
- void **Add** (**KeyStruct** keyStruct)

  Adds a key, value pair to the index.
- int **Find** (std::string key)

  Performs a linear search on the internal index to try to find the given key.
- int **BinarySearch** (std::string str)

  Performs a binary search on the internal index to try to find the given key.

Static Public Attributes

- static const int **notFound** = -1

4.12.1 Detailed Description

Definition at line **9** of file **PrimaryKey.h**.

4.12.2 Member Enumeration Documentation

4.12.2.1 IndexFileFormat

```
enum PrimaryKey::IndexFileFormat
```

Definition at line **13** of file **PrimaryKey.h**.

```
00013                           {
00014         ASCII,
00015         BCD,
00016         BINARY
00017     };
```

4.12.3 Member Function Documentation

4.12.3.1 Add()

```
void PrimaryKey::Add (
              KeyStruct keyStruct )
```

Adds a key, value pair to the index.

Parameters

keyStruct	the struct containing the key, value pair to be added

Definition at line **73** of file **PrimaryKey.cpp**.

```
00073                                       {
00074     vKey.push_back(keyStruct);
00075 }
```

4.12.3.2 BinarySearch()

```
int PrimaryKey::BinarySearch (
             std::string str )
```

Performs a binary search on the internal index to try to find the given key.

Precondition

list does not have to be sorted since the first thing the function checks is if it is, then sorts it if it is not

Postcondition

internal list is sorted in ascending order

Parameters

key	the zip code to search for

Returns

int if >= 0, the return value is the offset from the start of the file to the start of the record

int if = -1, the key could not be found in the record

Definition at line **85** of file **PrimaryKey.cpp**.

```
00085                                       {  // Binary search for string type
00086     if (!isSorted) {
00087         std::sort(vKey.begin(), vKey.end(), CompareStr);
```

```
00088         isSorted = true;
00089     }
00090
00091     int left = 0;
00092     int right = vKey.size() - 1;
00093     int middle;
00094
00095     while (left <= right) {
00096         middle = left + ((right - left) / 2);
00097
00098         std::string& k = vKey[middle].key;
00099         if (key == k) {
00100             return vKey[middle].offset;
00101         }
00102
00103         if (key.length() > k.length() || (key > k && key.length() == k.length())) {
00104             left = middle + 1;
00105         } else {
00106             right = middle - 1;
00107         }
00108     }
00109
00110     return -1;
00111 }
```

4.12.3.3 Find()

```
int PrimaryKey::Find (
            std::string key )
```

Performs a linear search on the internal index to try to find the given key.

Parameters

key	the zip code to search for

Returns

int if >= 0, the return value is the offset from the start of the file to the start of the record

int if = -1, the key could not be found in the record

Definition at line **77** of file **PrimaryKey.cpp**.

```
00077                                   {
00078     for (auto& e :  vKey)
00079         if (key == e.key)
00080             return e.offset;
00081
00082     return -1;
00083 }
```

4.12.3.4 GenerateIndexFile()

```
void PrimaryKey::GenerateIndexFile (
            std::string fileName )
```

Generates an index file from the internal index.

Parameters

fileName	the name of the index file that will be created

Definition at line **12** of file **PrimaryKey.cpp**.

```
00012                                                   {
00013     std::ofstream ofile(fileName, std::ios::binary);
00014
00015     header.version = 1;
00016     header.keyCount = vKey.size();
00017     header.format = IndexFileFormat::ASCII;
00018     ofile « header.version « " " « header.keyCount « " " « header.format « std::endl;
00019
00020     for (size_t i = 0; i < vKey.size(); i++) {
00021         ofile « vKey[i].key « " " « vKey[i].offset « '\n';
00022     }
00023
00024     ofile.close();
00025 }
```

4.12.3.5 ReadIndexFile()

```
bool PrimaryKey::ReadIndexFile (
            std::string fileName )
```

Reads an index file and stores the values in the internal index. Also records if the index's primary keys are in ascending order, to facilitate binary search.

Parameters

fileName	the name of the index file to be read

Returns

true file was read successfully

false file was unable to be read successfully

Definition at line **27** of file **PrimaryKey.cpp**.

```
00027                                                   {
00028     std::ifstream iFile(fileName, std::ios::binary);
00029
00030     if (!iFile.is_open())
00031         return false;
00032
00033     vKey.clear();
00034
00035     std::string str;
00036     unsigned int offset;
00037
00038     std::string line;
00039     PrimaryKey::KeyStruct prevZip = {"0", 0};  // dummy value for first comparison
00040
00041     isSorted = true;
00042     bool initHeader = false;
00043     while (std::getline(iFile, line)) {
00044         std::stringstream ss(line);
00045         if (!initHeader) {
00046             ss » header.version;
00047             ss » header.keyCount;
00048             ss » header.format;
00049             initHeader = true;
00050             continue;
00051         }
00052         ss » str;
```

```
00053         ss » offset;
00054         PrimaryKey::KeyStruct currZip = {str, offset};
00055         vKey.push_back(currZip);
00056 
00057         // here we check each record to see if the zip codes are in ascending order in the file
00058         // to determine if we need to sort before binary searching
00059         if (isSorted && !CompareStr(prevZip, currZip)) {
00060             isSorted = false;
00061             // if the index is not sorted, the if statement will short circut upon checking isSorted
00062             // and thus we don't need to keep reassigning prevZip, hence the continue
00063             continue;
00064         } else {
00065             prevZip = currZip;
00066         }
00067     }
00068 
00069     iFile.close();
00070     return true;
00071 }
```

4.12.4 Member Data Documentation

4.12.4.1 notFound

```
const int PrimaryKey::notFound = -1  [static]
```

Definition at line **12** of file **PrimaryKey.h**.

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

- PrimaryKey.h
- PrimaryKey.cpp

Chapter 5

File Documentation

5.1 CsvBuffer.cpp

```
00001
00004 #include "CsvBuffer.h"
00005 #include <iostream>
00006 #include <regex>
00007 #include "enums.h"
00008
00009 CsvBuffer::CsvBuffer(const char delim) :  delim(delim){};
00010
00011 void CsvBuffer::init(std::istream& instream) {
00012     read(instream);
00013     readHeader();
00014 }
00015
00016 bool CsvBuffer::read(std::istream& instream) {
00017     bool inQuotes = false;
00018     bool endOfFile = false;
00019
00020     curr = 0;
00021     buffer.clear();
00022
00023     char c = 0;
00024     while (!instream.eof()) {
00025         endOfFile = instream.get(c).eof();  // will be set to true if we try to read beyond the end of
      the file
00026
00027         if (c == '\r' && instream.peek() == '\n' && !inQuotes) {
00028             continue;
00029         } else if (c == '\n' && !inQuotes) {
00030             buffer.push_back(c);
00031             break;
00032         } else if (c == '"') {
00033             inQuotes = !inQuotes;
00034         }
00035
00036         buffer.push_back(c);
00037     }
00038
00039     return !endOfFile;
00040 }
00041
00042 bool CsvBuffer::unpack(std::string& str) {
00043     auto state = CSVState::UnquotedField;  // assume field is not quoted by default
00044
00045     bool fieldHasMore = true;
00046     bool recordHasMore = true;
00047     while (fieldHasMore) {
00048         char c = buffer[curr];
00049         switch (state) {
00050             case CSVState::UnquotedField:
00051                 if (c == delim) {
00052                     fieldHasMore = false;
00053                     fieldNum++;
00054                 } else if (c == '\n') {
00055                     fieldHasMore = false;
00056                     recordHasMore = false;
00057                     fieldNum = 0;
00058                 } else if (c == '"') {
00059                     state = CSVState::QuotedField;
```

```
00060                 } else {
00061                     str.push_back(c);
00062                 }
00063                 break;
00064             case CSVState::QuotedField:
00065                 if (c == '"') {
00066                     state = CSVState::QuotedQuote;
00067                 } else {
00068                     str.push_back(c);
00069                 }
00070                 break;
00071             case CSVState::QuotedQuote:
00072                 if (c == delim) {
00073                     fieldHasMore = false;
00074                     fieldNum++;
00075                 } else if (c == '"') {
00076                     str.push_back(c);
00077                     state = CSVState::QuotedField;
00078                 } else {
00079                     state = CSVState::UnquotedField;
00080                 }
00081                 break;
00082         }
00083         curr++;
00084     }
00085     return recordHasMore;
00086 }
00087
00088 std::pair<HeaderField, std::string> CsvBuffer::getCurFieldHeader() {
00089     return headers[fieldNum];
00090 }
00091
00092 HeaderField getFieldType(std::string headerValue) {
00093     std::regex zipCodePat("Zip\\s*Code");
00094     std::regex placeNamePat("Place\\s*Name");
00095     std::regex statePat("State");
00096     std::regex countyPat("County");
00097     std::regex latitudePat("Lat");
00098     std::regex longitudePat("Long");
00099
00100     if (std::regex_search(headerValue, zipCodePat)) {
00101         return HeaderField::ZipCode;
00102     } else if (std::regex_search(headerValue, placeNamePat)) {
00103         return HeaderField::PlaceName;
00104     } else if (std::regex_search(headerValue, statePat)) {
00105         return HeaderField::State;
00106     } else if (std::regex_search(headerValue, countyPat)) {
00107         return HeaderField::County;
00108     } else if (std::regex_search(headerValue, latitudePat)) {
00109         return HeaderField::Latitude;
00110     } else if (std::regex_search(headerValue, longitudePat)) {
00111         return HeaderField::Longitude;
00112     } else {
00113         return HeaderField::Unknown;
00114     }
00115 }
00116
00117 void CsvBuffer::readHeader() {
00118     bool more = true;
00119     while (more) {
00120         std::string temp;
00121         more = unpack(temp);
00122         headers.push_back({getFieldType(temp), temp});
00123     }
00124     numFields = headers.size();
00125 }
00126
00127 std::vector<std::pair<HeaderField, std::string» CsvBuffer::getHeaders()const {
00128     return headers;
00129 }
```

5.2 CsvBuffer.h

```
00001
00004 #ifndef CSVBUFFER_H
00005 #define CSVBUFFER_H
00006
00007 #include <istream>
00008 #include <string>
00009 #include <vector>
00010
00011 #include "enums.h"
00012
```

```
00013 class CsvBuffer {
00014    private:
00015     const char delim;
00016
00017     std::string buffer;
00018
00020     size_t curr;
00021
00023     // size_t recordCount;
00025     size_t fieldNum;
00027     size_t numFields;
00028
00029     // first part holds the header type for use when unpacking,
00030     // second part holds the actual header value
00031     std::vector<std::pair<HeaderField, std::string» headers;
00032
00042     void readHeader();
00043
00044    public:
00051     CsvBuffer(const char delim = ',');
00052
00063     bool read(std::istream& instream);
00064
00079     bool unpack(std::string& str);
00080
00092     void init(std::istream& instream);
00093
00103     std::pair<HeaderField, std::string> getCurFieldHeader();
00104
00105     std::vector<std::pair<HeaderField, std::string» getHeaders() const;
00106 };
00107
00108 #endif
```

5.3 enums.h

```
00001
00004 #ifndef ENUMS_H
00005 #define ENUMS_H
00006
00007 // used for tiny csv parsing state machine
00008 enum class CSVState {
00009     QuotedField,
00010     UnquotedField,
00011     QuotedQuote
00012 };
00013
00015 enum class HeaderField :  int {
00016     ZipCode,
00017     PlaceName,
00018     State,
00019     County,
00020     Latitude,
00021     Longitude,
00022     Unknown
00023 };
00024
00025 enum class LengthIndicatorType :  int {
00026     ASCII,
00027     BCD,
00028     BINARY
00029 };
00030
00031 #endif
```

5.4 Header.h

```
00001
00004 #ifndef HEADER_H
00005 #define HEADER_H
00006
00007 #include <cstring>
00008 #include <iostream>
00009 #include <string>
00010 #include <vector>
00011
00012 #include "enums.h"
00013
00014
```

```
00015 // needed to remove automatic alignment of struct members
00016 #pragma pack(push, 1)
00017 struct HeaderInfo {
00018     char magic[4];
00019     int version;
00020     int headerSize;
00021 
00022     friend std::istream& operator»(std::istream& ins, HeaderInfo& headerInfo) {
00023         ins.read((char*)(&headerInfo), sizeof(headerInfo));
00024         return ins;
00025     }
00026 
00027     friend std::ostream& operator«(std::ostream& os, HeaderInfo& headerInfo) {
00028         os.write(reinterpret_cast<char*>(&headerInfo.magic), sizeof(headerInfo.magic));
00029         os.write(reinterpret_cast<char*>(&headerInfo.version), sizeof(headerInfo.version));
00030         os.write(reinterpret_cast<char*>(&headerInfo.headerSize), sizeof(headerInfo.headerSize));
00031         return os;
00032     }
00033 };
00034 
00035 struct FileInfo {
00036     int lengthIndicatorSize;
00037     LengthIndicatorType lengthIndicatorFormat;
00038 
00039     int fieldsPerRecord;
00040     int primaryKeyPosition;
00041 
00042     char indexFileName[100];
00043 
00044     friend std::istream& operator»(std::istream& ins, FileInfo& fileInfo) {
00045         ins.read((char*)(&fileInfo), sizeof(fileInfo));
00046         return ins;
00047     }
00048 
00049     friend std::ostream& operator«(std::ostream& os, FileInfo& fileInfo) {
00050         os.write(reinterpret_cast<char*>(&fileInfo.lengthIndicatorSize),
      sizeof(fileInfo.lengthIndicatorSize));
00051         os.write(reinterpret_cast<char*>(&fileInfo.lengthIndicatorFormat),
      sizeof(fileInfo.lengthIndicatorFormat));
00052 
00053         os.write(reinterpret_cast<char*>(&fileInfo.fieldsPerRecord),
      sizeof(fileInfo.fieldsPerRecord));
00054         os.write(reinterpret_cast<char*>(&fileInfo.primaryKeyPosition),
      sizeof(fileInfo.primaryKeyPosition));
00055 
00056         os.write(reinterpret_cast<char*>(&fileInfo.indexFileName), sizeof(fileInfo.indexFileName));
00057 
00058         return os;
00059     }
00060 };
00061 
00062 struct FieldInfo {
00063     char fieldName[50];
00064     HeaderField fieldType;
00065 
00066     friend std::istream& operator»(std::istream& ins, FieldInfo& fieldInfo) {
00067         ins.read((char*)(&fieldInfo), sizeof(fieldInfo));
00068         return ins;
00069     }
00070 
00071     friend std::ostream& operator«(std::ostream& os, FieldInfo& fieldInfo) {
00072         os.write(reinterpret_cast<char*>(&fieldInfo), sizeof(fieldInfo));
00073         return os;
00074     }
00075 };
00076 
00077 struct Header {
00078     HeaderInfo headerInfo;
00079     FileInfo fileInfo;
00080     std::vector<FieldInfo> fields;
00081 
00082     friend std::ostream& operator«(std::ostream& os, Header& header) {
00083         os « header.headerInfo;
00084         os « header.fileInfo;
00085 
00086         for (auto f :  header.fields) {
00087             os « f;
00088         }
00089         return os;
00090     }
00091 };
00092 
00093 #pragma pack(pop)
00094 
00095 #endif
```

5.5 HeaderBuffer.cpp

```
00001
00004 #include "HeaderBuffer.h"
00005 #include <iostream>
00006
00007 void HeaderBuffer::read(std::istream& ins) {
00008     ins.seekg(0);
00009
00010     HeaderInfo hInfo;
00011     ins » hInfo;
00012
00013     buffer.resize(hInfo.headerSize);
00014     ins.seekg(0);
00015
00016     char c;
00017     for (int i = 0; i < hInfo.headerSize; i++) {
00018         ins.read(&c, 1);
00019         buffer[i] = c;
00020     }
00021 }
00022
00023 Header HeaderBuffer::unpack() {
00024     Header header;
00025     std::vector<FieldInfo> fields;
00026
00027     auto fileInfoOffset = sizeof(Header::headerInfo);
00028
00029     memcpy(&header.headerInfo, &buffer[0], sizeof(HeaderInfo));
00030     memcpy(&header.fileInfo, &buffer[fileInfoOffset], sizeof(FileInfo));
00031
00032     size_t fieldInfoOffset = fileInfoOffset + sizeof(Header::fileInfo);
00033     for (int i = 0; i < header.fileInfo.fieldsPerRecord; i++) {
00034         FieldInfo fieldInfo;
00035         memcpy(&fieldInfo, &buffer[fieldInfoOffset], sizeof(FieldInfo));
00036         fields.push_back(fieldInfo);
00037
00038         // set offset to beginning of next field info
00039         fieldInfoOffset += sizeof(FieldInfo);
00040     }
00041
00042     header.fields = fields;
00043
00044     return header;
00045 }
```

5.6 HeaderBuffer.h

```
00001
00004 #ifndef HEADER_BUFFER_H
00005 #define HEADER_BUFFER_H
00006
00007 #include <iostream>
00008 #include <vector>
00009
00010 #include "Header.h"
00011
00012 class HeaderBuffer {
00013    private:
00014     std::vector<unsigned char> buffer;
00015
00016    public:
00027     void read(std::istream& ins);
00028
00037     Header unpack();
00038 };
00039
00040 #endif
```

5.7 LengthIndicatedBuffer.cpp

```
00001
00004 #include "LengthIndicatedBuffer.h"
00005 #include <iomanip>
00006 #include <iostream>
00007 #include <regex>
00008 #include <sstream>
00009 #include <vector>
00010
```

```
00011 #include "HeaderBuffer.h"
00012
00013 const char MAGIC_HEADER_NUMBER[4] = {'Z', 'C', '0', '2'};
00014
00015 LengthIndicatedBuffer::LengthIndicatedBuffer(const char delim) :  delim(delim) {
00016     clear();
00017     memset(buffer, 0, sizeof(buffer));
00018 }
00019
00020 std::string LengthIndicatedBuffer::getIndexFileName() {
00021     return header.fileInfo.indexFileName;
00022 }
00023
00024 bool LengthIndicatedBuffer::checkFileType(std::istream& instream) {
00025     instream.seekg(0);
00026     char first4[4];
00027
00028     instream.read(first4, sizeof(first4));
00029
00030     bool good = true;
00031     for (int i = 0; i < 4; i++) {
00032         good = (first4[i] == MAGIC_HEADER_NUMBER[i]);
00033     }
00034     return good;
00035 }
00036
00037 bool LengthIndicatedBuffer::init(std::istream& instream) {
00038     if (checkFileType(instream)) {
00039         // if file has magic number
00040         instream.seekg(0);
00041         HeaderBuffer hBuf;
00042         hBuf.read(instream);
00043         header = hBuf.unpack();
00044         initialized = true;
00045
00046     } else {
00047         // if file does not have magic number
00048         initialized = false;
00049     }
00050
00051     return initialized;
00052 }
00053
00054 bool LengthIndicatedBuffer::read(std::istream& instream, int indexOffset) {
00055     instream.seekg(indexOffset);
00056     return read(instream);
00057 }
00058
00059 bool LengthIndicatedBuffer::read(std::istream& instream) {
00060     clear();
00061
00062     // get length of record length indicator
00063     auto lengthIndicatorLength = header.fileInfo.lengthIndicatorSize;
00064
00065     int recordLength = 0;
00066
00067     // get actual record length
00068     switch (LengthIndicatorType(header.fileInfo.lengthIndicatorFormat)) {
00069         case LengthIndicatorType::BINARY:
00070             instream.read((char*)&recordLength, lengthIndicatorLength);
00071             break;
00072         case LengthIndicatorType::ASCII: {
00073             std::string recordLengthStr;
00074             char c;
00075             for (int i = 0; i < lengthIndicatorLength; i++) {
00076                 instream.get(c);
00077                 if (c == EOF) {
00078                     return false;
00079                 }
00080                 recordLengthStr.push_back(c);
00081             }
00082
00083             try {
00084                 recordLength = std::stoi(recordLengthStr);
00085             } catch (std::invalid_argument& err) {
00086                 if (instream.eof()) {
00087                     return false;
00088                 }
00089                 throw err;
00090             }
00091
00092             break;
00093         }
00094         case LengthIndicatorType::BCD:
00095             break;
00096     }
00097
```

```
00098     // read record
00099     instream.read(buffer, recordLength);
00100     this->recordLength = recordLength;
00101     return instream.good();
00102 }
00103
00104 void LengthIndicatedBuffer::writeHeader(std::ostream& outstream) {
00105     outstream.seekp(0);
00106     outstream « header;
00107 }
00108
00109 bool LengthIndicatedBuffer::unpack(std::string& str) {
00110     auto state = CSVState::UnquotedField;  // assume field is not quoted by default
00111
00112     bool fieldHasMore = true;
00113     bool recordHasMore = true;
00114     while (fieldHasMore) {
00115         char c = buffer[curr];
00116         switch (state) {
00117             case CSVState::UnquotedField:
00118                 if (c == delim) {
00119                     fieldHasMore = false;
00120                     fieldNum++;
00121                 } else if (curr >= recordLength) {
00122                     fieldHasMore = false;
00123                     recordHasMore = false;
00124                     fieldNum = 0;
00125                 } else if (c == '"') {
00126                     state = CSVState::QuotedField;
00127                 } else {
00128                     str.push_back(c);
00129                 }
00130                 break;
00131             case CSVState::QuotedField:
00132                 if (c == '"') {
00133                     state = CSVState::QuotedQuote;
00134                 } else {
00135                     str.push_back(c);
00136                 }
00137                 break;
00138             case CSVState::QuotedQuote:
00139                 if (c == delim) {
00140                     fieldHasMore = false;
00141                     fieldNum++;
00142                 } else if (c == '"') {
00143                     str.push_back(c);
00144                     state = CSVState::QuotedField;
00145                 } else {
00146                     state = CSVState::UnquotedField;
00147                 }
00148                 break;
00149         }
00150         curr++;
00151     }
00152     return recordHasMore;
00153 }
00154
00155 void LengthIndicatedBuffer::clear() {
00156     curr = 0;
00157 }
00158
00159 void LengthIndicatedBuffer::pack(const std::string str) {
00160     // put delimiters in between each field skipping the first
00161     if (curr > 0) {
00162         buffer[curr++] = delim;
00163     }
00164
00165     // note that the size method of std::string does not include
00166     // the null terminator in the length, thus we are just copying the
00167     // values in the string to the buffer, as desired
00168     memcpy(&buffer[curr], str.c_str(), str.size());
00169
00170     // move curr pointer to the end of the field we just added
00171     curr += str.size();
00172 }
00173
00174 void LengthIndicatedBuffer::write(std::ostream& outstream) {
00175     switch (LengthIndicatorType(header.fileInfo.lengthIndicatorFormat)) {
00176         case LengthIndicatorType::ASCII: {
00177             std::ostringstream lengthStream;
00178             lengthStream « std::setfill('0') « std::setw(header.fileInfo.lengthIndicatorSize) «
    (curr);
00179             auto lengthStr = lengthStream.str();
00180             outstream.write(lengthStr.c_str(), lengthStream.str().size());
00181             break;
00182         }
00183         case LengthIndicatorType::BCD:
```

```
00184         case LengthIndicatorType::BINARY:
00185             break;
00186     }
00187
00188     outstream.write(buffer, curr);
00189 }
00190
00191 FieldInfo LengthIndicatedBuffer::getCurFieldHeader() {
00192     return header.fields[fieldNum];
00193 }
```

5.8 LengthIndicatedBuffer.h

```
00001
00004 #ifndef LIBUFFER_H
00005 #define LIBUFFER_H
00006
00007 #include <array>
00008 #include <istream>
00009 #include <string>
00010 #include <vector>
00011
00012 #include "Header.h"
00013 #include "enums.h"
00014
00015 class LengthIndicatedBuffer {
00016    private:
00017     const char delim;
00018     int recordLength;
00019     bool initialized = false;
00020
00021
00022     char buffer[1000];
00023
00025     int curr;
00026
00028     size_t fieldNum = 0;
00030     size_t numFields;
00031
00032
00033    public:
00034     Header header;
00035
00036
00043     LengthIndicatedBuffer(const char delim = ',');
00044
00058     bool read(std::istream& instream);
00059
00074     bool read(std::istream& instream, int indexOffset);
00075
00091     bool unpack(std::string& str);
00092
00105     void pack(const std::string str);
00106
00117     void write(std::ostream& outstream);
00118
00132     bool init(std::istream& instream);
00133
00141     void writeHeader(std::ostream& outstream);
00142
00149     void clear();
00150
00162     bool checkFileType(std::istream& instream);
00163
00172     std::string getIndexFileName();
00173
00183     FieldInfo getCurFieldHeader();
00184 };
00185
00186 #endif
```

5.9 LengthIndicatedFile.cpp

```
00001
00004 #include "LengthIndicatedFile.h"
00005
00006
00007 LengthIndicatedFile::LengthIndicatedFile(std::string fileName) :  fileName(fileName) {
00008     openDataFile();
```

```
00009     initializeBuffers();
00010     dataStart = header.headerInfo.headerSize;
00011     initializeIndex();
00012 }
00013
00014 LengthIndicatedFile::~LengthIndicatedFile() {
00015     file.close();
00016 }
00017
00018 void LengthIndicatedFile::initializeBuffers() {
00019     std::ifstream dataFile(fileName);
00020     readBuf.init(dataFile);
00021     writeBuf.init(dataFile);
00022     this->header = readBuf.header;
00023     dataFile.close();
00024 }
00025
00026 void LengthIndicatedFile::initializeIndex() {
00027     if (indexFileExists()) {
00028         index.ReadIndexFile(header.fileInfo.indexFileName);
00029     } else {
00030         generateIndex();
00031     }
00032 }
00033
00034 bool LengthIndicatedFile::openDataFile() {
00035     file.open(fileName, std::ios::binary | std::ios::in | std::ios::out);
00036     return file.good();
00037 }
00038
00039 std::optional<Place> LengthIndicatedFile::findRecord(std::string recordKey) {
00040     file.clear();
00041     auto recordFound = index.BinarySearch(recordKey);
00042
00043     if (recordFound == index.notFound) {
00044         return {};
00045     }
00046     Place p;
00047     readBuf.read(file, recordFound);
00048     p.unpack(readBuf);
00049     return p;
00050 }
00051
00052 void LengthIndicatedFile::generateIndex() {
00053     file.clear();
00054     file.seekg(dataStart);
00055
00056     auto pos = (unsigned int)file.tellg();
00057     while (readBuf.read(file)) {
00058         Place place;
00059         place.unpack(readBuf);
00060         index.Add({place.getZipCode(), pos});
00061         pos = (unsigned int)file.tellg();
00062     }
00063     index.GenerateIndexFile(header.fileInfo.indexFileName);
00064 }
00065
00066 bool LengthIndicatedFile::indexFileExists() {
00067     return std::filesystem::exists(header.fileInfo.indexFileName);
00068 }
```

5.10 LengthIndicatedFile.h

```
00001
00004 #ifndef LENGTHINDICATEDFILE_H
00005 #define LENGTHINDICATEDFILE_H
00006
00007 #include <filesystem>
00008 #include <fstream>
00009 #include <iostream>
00010 #include <optional>
00011
00012 #include "Header.h"
00013 #include "LengthIndicatedBuffer.h"
00014 #include "Place.h"
00015 #include "PrimaryKey.h"
00016 #include "enums.h"
00017
00018 class LengthIndicatedFile {
00019    private:
00020     Header header;
00021     PrimaryKey index;
00022
```

```
00023     LengthIndicatedBuffer readBuf;
00024     LengthIndicatedBuffer writeBuf;
00025
00026     std::string fileName;
00027     std::fstream file;
00028
00029     int dataStart;
00030
00031    public:
00032     LengthIndicatedFile(std::string fileName);
00033     ~LengthIndicatedFile();
00034     void initializeBuffers();
00035     void initializeIndex();
00036     bool openDataFile();
00037     std::optional<Place> findRecord(std::string recordKey);
00038     void generateIndex();
00039     bool indexFileExists();
00040 };
00041
00042 #endif
```

5.11 main.cpp

```
00001
00004 #include <filesystem>
00005 #include <fstream>
00006 #include <iomanip>
00007 #include <iostream>
00008 #include <string>
00009
00010 #include "CsvBuffer.h"
00011 #include "HeaderBuffer.h"
00012 #include "LengthIndicatedBuffer.h"
00013 #include "LengthIndicatedFile.h"
00014 #include "Place.h"
00015 #include "PrimaryKey.h"
00016
00027 std::string addingSpace(std::string str, char c) {
00028     std::string s1 = "";
00029     for (size_t i = 0; i < str.length(); i++) {
00030         if (str[i] != c)
00031             s1 = s1 + str[i];
00032         else
00033             s1 = s1 + "\t" + str[i] + "\t";
00034     }
00035     return s1;
00036 }
00037
00038 void transferRecords(std::istream& csvFile, std::iostream& lirfFile) {
00039     LengthIndicatedBuffer lBuf;
00040     CsvBuffer cBuf;
00041
00042     csvFile.seekg(0);
00043     lirfFile.seekg(0);
00044
00045     lBuf.init(lirfFile);
00046     cBuf.init(csvFile);
00047
00048     auto startPos = lBuf.header.headerInfo.headerSize;
00049
00050     // seek past the header, should be the start of the first record
00051     lirfFile.seekp(startPos);
00052
00053     while (cBuf.read(csvFile)) {
00054         Place p;
00055         p.unpack(cBuf);
00056
00057         p.pack(lBuf);
00058         lBuf.write(lirfFile);
00059     }
00060 }
00061
00062 void convertFileType(std::istream& csvFile, std::ostream& lirfFile, std::string lirfFileName) {
00063     CsvBuffer csvBuf;
00064     csvBuf.init(csvFile);
00065
00066     auto csvHeaders = csvBuf.getHeaders();
00067
00068     std::string indexFileName = lirfFileName.substr(0, 96) + ".idx";
00069
00070     std::vector<FieldInfo> fields;
00071
00072     for (auto h :  csvHeaders) {
```

```
00073         auto type = h.first;
00074         auto name = h.second;
00075 
00076         FieldInfo field;
00077 
00078         // clear fieldName array
00079         memset(field.fieldName, 0, sizeof(field.fieldName));
00080         name.copy(field.fieldName, sizeof(field.fieldName));
00081 
00082         field.fieldType = type;
00083         fields.push_back(field);
00084     }
00085 
00086     Header header = {
00087         {
00088             {'Z', 'C', '0', '2'},  // magic number
00089             1,                     // version number
00090             0                      // length of header (will be set later)
00091         },
00092         {
00093             2,                            // length indicator length
00094             LengthIndicatorType::ASCII,   // length indicator type
00095             (int)csvHeaders.size(),       // number of fields
00096             0,                            // primary key position
00097             ""                            // name of the index file (will be set later)
00098         },
00099         {}};
00100 
00101     for (auto f :  fields) {
00102         header.fields.push_back(f);
00103     }
00104 
00105     // clear entire index file name array
00106     memset(header.fileInfo.indexFileName, 0, 100);
00107 
00108     auto numFields = fields.size();
00109     auto headerSize = sizeof(header.headerInfo) + sizeof(header.fileInfo) + sizeof(FieldInfo) *
      numFields;
00110 
00111     header.fileInfo.fieldsPerRecord = numFields;
00112     header.headerInfo.headerSize = headerSize;
00113 
00114     indexFileName.copy(header.fileInfo.indexFileName, 100);
00115 
00116     lirfFile « header;
00117 }
00118 
00119 std::vector<std::string> parseZipArg(std::string zipList) {
00120     std::vector<std::string> zips;
00121     if (!zipList.size()) {  // no zip codes given
00122         return zips;
00123     }
00124 
00125     size_t commaPos;
00126     int offset = 0;
00127     while ((commaPos = zipList.find(',', offset)) != std::string::npos) {
00128         zips.push_back(zipList.substr(offset, commaPos - offset));
00129         offset = commaPos + 1;
00130     }
00131     zips.push_back(zipList.substr(offset));
00132     return zips;
00133 }
00134 
00135 bool parseArgs(int argc, char const* argv[], std::vector<std::string>& zipList, std::string&
      csvFileName, std::string& lirfFileName) {
00136     const std::string zipFlag = "-Z";
00137     const std::string csvFlag = "-C";
00138 
00139     for (int i = 1; i < argc; i++) {
00140         auto arg = std::string(argv[i]);
00141         if (arg.size() > 2) {  // check for flag at beginning of arg
00142             auto first2 = arg.substr(0, 2);
00143 
00144             if (first2 == zipFlag) {
00145                 zipList = parseZipArg(arg.substr(2));  // send arg string minus flag characters
00146             } else if (first2 == csvFlag) {
00147                 csvFileName = arg.substr(2);
00148             } else {  // if no flag, treat it as a length indicated file name
00149                 lirfFileName = arg;
00150             }
00151         }
00152     }
00153 
00154     return true;
00155 }
00156 
00157 void printFoundZips(std::vector<Place>& found) {
```

```
00158     size_t zip_w = 0;
00159     size_t name_w = 0;
00160     size_t state_w = 0;
00161     size_t county_w = 0;
00162     size_t lat_w = 12;
00163     size_t long_w = 12;
00164
00165     // calculate widths so that the width of each column is slightly larger than the maximum length
      field
00166     for (auto place :  found) {
00167         if (place.getZipCode().size() > zip_w) {
00168             zip_w = place.getZipCode().size() + 5;
00169         }
00170         if (place.getName().size() > name_w) {
00171             name_w = place.getName().size() + 5;
00172         }
00173         if (place.getState().size() > state_w) {
00174             state_w = place.getState().size() + 5;
00175         }
00176         if (place.getCounty().size() > county_w) {
00177             county_w = place.getCounty().size() + 6;
00178         }
00179     }
00180
00181     size_t total = zip_w + name_w + state_w + county_w + lat_w + long_w;
00182
00183     std::cout « std::setfill('-') « std::setw(total) « "-" « std::endl;
00184
00185     std::cout « std::setfill(' ') « std::setw(zip_w) « std::left
00186               « "Zip" « std::setw(name_w)
00187               « "Place Name" « std::setw(state_w)
00188               « "State" « std::setw(county_w)
00189               « "County" « std::setw(lat_w)
00190               « "Latitude" « std::setw(long_w)
00191               « "Longitude"
00192               « std::endl;
00193
00194     std::cout « std::setfill('-') « std::setw(total) « "-" « std::endl;
00195     // print the zipcodes that were found
00196     for (auto place :  found) {
00197         std::cout « std::setprecision(10)
00198                   « std::setfill(' ') « std::setw(zip_w) « std::left
00199                   « place.getZipCode() « std::setw(name_w)
00200                   « place.getName() « std::setw(state_w)
00201                   « place.getState() « std::setw(county_w)
00202                   « place.getCounty() « std::setw(lat_w)
00203                   « place.getLat() « std::setw(long_w)
00204                   « place.getLongi()
00205                   « std::endl;
00206     }
00207
00208     std::cout « std::setfill('-') « std::setw(total) « "-" « std::endl;
00209 }
00210
00211 void printNotFoundZips(std::vector<std::string>& notFound) {
00212     std::cout « "\n\nThe following zip codes did not match any records in the file:" « std::endl;
00213     for (auto zip :  notFound) {
00214         std::cout « std::setprecision(10)
00215                   « std::setfill(' ') « std::setw(5) « std::left
00216                   « zip « std::endl;
00217     }
00218 }
00219
00229 int main(int argc, char const* argv[]) {
00230     // check to see if there is a command line argument
00231     if (argc < 2) {
00232         std::cerr « "No input file given" « std::endl;
00233         exit(1);
00234     }
00235
00236     std::vector<std::string> zipList;
00237     std::string csvFileName;
00238     std::string lirfFileName;
00239
00240     parseArgs(argc, argv, zipList, csvFileName, lirfFileName);
00241
00242     // LengthIndicatedBuffer works the same way as the CsvBuffer,
00243     // before doing anything, pass an open ifstream pointing to the length indicated file to the init
      method,
00244     // then passing the same ifstream to the read method will read one record into the buffer
00245     // passing a place object to the unpack method will then unpack the buffer into the place object
00246     // read returns true if there are more records in the file, false if there are no more records
00247
00248     // if we are given a csv file and a lirf file
00249     // we will convert the csv file to the lirf format
00250     if (csvFileName.size() && lirfFileName.size()) {
00251         if (!std::filesystem::exists(csvFileName)) {
```

```
00252             std::cerr « "Input CSV file does not exist" « std::endl;
00253             return 1;
00254         }
00255
00256         std::ifstream csvFile(csvFileName, std::ios::binary | std::ios::in | std::ios::out);
00257         std::fstream lirfFile(lirfFileName, std::ios::binary | std::ios::in | std::ios::out |
      std::ios::trunc);
00258
00259         convertFileType(csvFile, lirfFile, lirfFileName);
00260         transferRecords(csvFile, lirfFile);
00261
00262         csvFile.close();
00263         lirfFile.close();
00264
00265         return 0;
00266     }
00267
00268     if (zipList.size() && lirfFileName.size()) {
00269         LengthIndicatedFile lirfFile(lirfFileName);
00270
00271         std::vector<Place> foundPlaces;
00272         std::vector<std::string> notFoundPlaces;
00273         for (auto zip :  zipList) {
00274             auto obj = lirfFile.findRecord(zip);
00275             if (obj) {
00276                 foundPlaces.push_back(*obj);
00277             } else {
00278                 notFoundPlaces.push_back(zip);
00279             }
00280         }
00281
00282         if (foundPlaces.size() > 0) {
00283             printFoundZips(foundPlaces);
00284             if (notFoundPlaces.size() > 0) {
00285                 printNotFoundZips(notFoundPlaces);
00286             }
00287         } else {
00288             std::cout « "No Zip Codes found" « std::endl;
00289         }
00290
00291         return 0;
00292     }
00293
00294     if (lirfFileName.size()) {
00295         // check if index exists and generate it if it doesn't
00296         LengthIndicatedFile lirfFile(lirfFileName);
00297
00298         return 0;
00299     }
00300 }
```

5.12 Place.cpp

```
00001
00004 #include "Place.h"
00005 #include <fstream>
00006 #include <iomanip>
00007 #include <sstream>
00008
00009 using std::string;
00010
00011 Place::Place() {
00012     zipcode = "";
00013     state = "";
00014     name = "";
00015     latitude = 0;
00016     county = "";
00017     longitude = 0;
00018 };
00019
00020 // copy constructor
00021 Place::Place(const Place& loc) {
00022     zipcode = loc.getZipCode();
00023     name = loc.getName();
00024     state = loc.getState();
00025     county = loc.getCounty();
00026     latitude = loc.getLat();
00027     longitude = loc.getLongi();
00028 };
00029
00030 // overload the assignment operator
00031 void Place ::operator=(const Place& loc) {
00032     this->zipcode = loc.getZipCode();
```

```
00033     this->name = loc.getName();
00034     this->state = loc.getState();
00035     this->county = loc.getCounty();
00036     this->latitude = loc.getLat();
00037     this->longitude = loc.getLongi();
00038 }
00039
00040 string Place::getZipCode()const { return zipcode; }  // zipcode value
00041 string Place::getName()const { return name; }        // name value
00042 string Place::getState()const { return state; }      // State value
00043 string Place::getCounty()const { return county; }    // county value
00044 double Place::getLat()const { return latitude; }     // Latitude value
00045 double Place::getLongi()const { return longitude; }  // longitude value
00046
00047 // passing to place object by unpacking from buffer
00048 void Place::unpack(CsvBuffer& buffer) {
00049     std::string skip;
00050     string lat_str, long_str;
00051
00052     bool moreFields = true;
00053     while (moreFields) {
00054         auto curField = buffer.getCurFieldHeader();
00055         switch (HeaderField(curField.first)) {
00056             case HeaderField::ZipCode:
00057                 moreFields = buffer.unpack(zipcode);
00058                 break;
00059             case HeaderField::PlaceName:
00060                 moreFields = buffer.unpack(name);
00061                 break;
00062             case HeaderField::State:
00063                 moreFields = buffer.unpack(state);
00064                 break;
00065             case HeaderField::County:
00066                 moreFields = buffer.unpack(county);
00067                 break;
00068             case HeaderField::Latitude:
00069                 moreFields = buffer.unpack(lat_str);
00070                 break;
00071             case HeaderField::Longitude:
00072                 moreFields = buffer.unpack(long_str);
00073                 break;
00074             default:
00075                 moreFields = buffer.unpack(skip);
00076                 break;
00077         }
00078     }
00079     std::stringstream(lat_str) » std::setprecision(10) » latitude;    // convert to float
00080     std::stringstream(long_str) » std::setprecision(10) » longitude;  // convert to float
00081 }
00082
00083 void Place::unpack(LengthIndicatedBuffer& buffer) {
00084     std::string skip;
00085     string lat_str, long_str;
00086
00087     bool hasMore = true;
00088     while (hasMore) {
00089         auto curField = buffer.getCurFieldHeader();
00090         switch (HeaderField(curField.fieldType)) {
00091             case HeaderField::ZipCode:
00092                 hasMore = buffer.unpack(zipcode);
00093                 break;
00094             case HeaderField::PlaceName:
00095                 hasMore = buffer.unpack(name);
00096                 break;
00097             case HeaderField::State:
00098                 hasMore = buffer.unpack(state);
00099                 break;
00100             case HeaderField::County:
00101                 hasMore = buffer.unpack(county);
00102                 break;
00103             case HeaderField::Latitude:
00104                 hasMore = buffer.unpack(lat_str);
00105                 break;
00106             case HeaderField::Longitude:
00107                 hasMore = buffer.unpack(long_str);
00108                 break;
00109             default:
00110                 hasMore = buffer.unpack(skip);
00111                 break;
00112         }
00113     }
00114
00115     std::stringstream(lat_str) » std::setprecision(10) » latitude;    // convert to float
00116     std::stringstream(long_str) » std::setprecision(10) » longitude;  // convert to float
00117 }
00118
00119 void Place::pack(LengthIndicatedBuffer& buffer) {
```

```
00120     buffer.clear();
00121     std::stringstream lat_strStream;
00122     std::stringstream long_strStream;
00123 
00124     lat_strStream « std::setprecision(10) « latitude;
00125     long_strStream « std::setprecision(10) « longitude;
00126 
00127     for (auto f :  buffer.header.fields) {
00128         switch (HeaderField(f.fieldType)) {
00129             case HeaderField::ZipCode:
00130                 buffer.pack(zipcode);
00131                 break;
00132             case HeaderField::PlaceName:
00133                 buffer.pack(name);
00134                 break;
00135             case HeaderField::State:
00136                 buffer.pack(state);
00137                 break;
00138             case HeaderField::County:
00139                 buffer.pack(county);
00140                 break;
00141             case HeaderField::Latitude:
00142                 buffer.pack(lat_strStream.str());
00143                 break;
00144             case HeaderField::Longitude:
00145                 buffer.pack(long_strStream.str());
00146                 break;
00147             default:
00148                 break;
00149         }
00150     }
00151 }
00152 
00153 void Place::print() {
00154     std::cout « getZipCode() « " " « getName() « " " « getState() « " "
00155               « getCounty() « " " « getLat() « " " « getLongi() « " " « std::endl;
00156 }
00157 
00158 size_t Place::getSize() {
00159     size_t size = 0;
00160     size += name.size();
00161     size += zipcode.size();
00162     size += state.size();
00163     size += county.size();
00164     size += sizeof(latitude);
00165     size += sizeof(longitude);
00166     return size;
00167 }
```

5.13 Place.h

```
00001 
00004 #ifndef PLACE_H
00005 #define PLACE_H
00006 
00007 #include <string>
00008 
00009 #include "CsvBuffer.h"
00010 #include "LengthIndicatedBuffer.h"
00011 
00012 /* This file contains the record details like zipcode, State ID, longitude and latitude.
00013 We have the unpack and read functions that
00014 puts the data from CSV file into the buffer and reads the data from CSV file.  */
00015 class Place {
00016    public:
00017     Place();
00018 
00024     Place(const Place& loc);
00025 
00031     std::string getZipCode() const;
00032 
00038     std::string getState() const;
00039 
00045     std::string getName() const;
00046 
00052     std::string getCounty() const;
00053 
00059     double getLat() const;
00060 
00066     double getLongi() const;
00067 
00077     void unpack(CsvBuffer& buffer);
00078 
```

```
00088     void unpack(LengthIndicatedBuffer& buffer);
00089
00090     void pack(LengthIndicatedBuffer& buffer);
00091
00097     void operator=(const Place& loc);
00098
00104     size_t getSize();
00105
00106     void print();
00107
00108    private:
00109     std::string zipcode;
00110     std::string name;
00111     std::string state;
00112     std::string county;
00113     double latitude;
00114     double longitude;
00115 };
00116
00117 #endif
```

5.14 PrimaryKey.cpp

```
00001
00004 #include "PrimaryKey.h"
00005 #include <algorithm>
00006 #include <sstream>
00007
00008 bool CompareStr(PrimaryKey::KeyStruct& s1, PrimaryKey::KeyStruct& s2) {
00009     return s1.key.length() < s2.key.length() || (s1.key.length() == s2.key.length() && s1.key <
      s2.key);
00010 }
00011
00012 void PrimaryKey::GenerateIndexFile(std::string fileName) {
00013     std::ofstream ofile(fileName, std::ios::binary);
00014
00015     header.version = 1;
00016     header.keyCount = vKey.size();
00017     header.format = IndexFileFormat::ASCII;
00018     ofile « header.version « " " « header.keyCount « " " « header.format « std::endl;
00019
00020     for (size_t i = 0; i < vKey.size(); i++) {
00021         ofile « vKey[i].key « " " « vKey[i].offset « '\n';
00022     }
00023
00024     ofile.close();
00025 }
00026
00027 bool PrimaryKey::ReadIndexFile(std::string fileName) {
00028     std::ifstream iFile(fileName, std::ios::binary);
00029
00030     if (!iFile.is_open())
00031         return false;
00032
00033     vKey.clear();
00034
00035     std::string str;
00036     unsigned int offset;
00037
00038     std::string line;
00039     PrimaryKey::KeyStruct prevZip = {"0", 0};  // dummy value for first comparison
00040
00041     isSorted = true;
00042     bool initHeader = false;
00043     while (std::getline(iFile, line)) {
00044         std::stringstream ss(line);
00045         if (!initHeader) {
00046             ss » header.version;
00047             ss » header.keyCount;
00048             ss » header.format;
00049             initHeader = true;
00050             continue;
00051         }
00052         ss » str;
00053         ss » offset;
00054         PrimaryKey::KeyStruct currZip = {str, offset};
00055         vKey.push_back(currZip);
00056
00057         // here we check each record to see if the zip codes are in ascending order in the file
00058         // to determine if we need to sort before binary searching
00059         if (isSorted && !CompareStr(prevZip, currZip)) {
00060             isSorted = false;
00061             // if the index is not sorted, the if statement will short circut upon checking isSorted
```

```
00062             // and thus we don't need to keep reassigning prevZip, hence the continue
00063             continue;
00064         } else {
00065             prevZip = currZip;
00066         }
00067     }
00068
00069     iFile.close();
00070     return true;
00071 }
00072
00073 void PrimaryKey::Add(KeyStruct keyStruct) {
00074     vKey.push_back(keyStruct);
00075 }
00076
00077 int PrimaryKey::Find(std::string key) {
00078     for (auto& e :  vKey)
00079         if (key == e.key)
00080             return e.offset;
00081
00082     return -1;
00083 }
00084
00085 int PrimaryKey::BinarySearch(std::string key) {  // Binary search for string type
00086     if (!isSorted) {
00087         std::sort(vKey.begin(), vKey.end(), CompareStr);
00088         isSorted = true;
00089     }
00090
00091     int left = 0;
00092     int right = vKey.size() - 1;
00093     int middle;
00094
00095     while (left <= right) {
00096         middle = left + ((right - left) / 2);
00097
00098         std::string& k = vKey[middle].key;
00099         if (key == k) {
00100             return vKey[middle].offset;
00101         }
00102
00103         if (key.length() > k.length() || (key > k && key.length() == k.length())) {
00104             left = middle + 1;
00105         } else {
00106             right = middle - 1;
00107         }
00108     }
00109
00110     return -1;
00111 }
```

5.15 PrimaryKey.h

```
00001
00004 #pragma once
00005 #include <fstream>
00006
00007 #include "Header.h"
00008
00009 class PrimaryKey {
00010
00011    public:
00012     static const int notFound = -1;
00013     enum IndexFileFormat {
00014         ASCII,
00015         BCD,
00016         BINARY
00017     };
00018      struct IndexFileHeader {
00019         int version;
00020         int keyCount;
00021         int format;
00022     };
00023     struct KeyStruct {
00024         std::string key;
00025         unsigned int offset;
00026     };
00032     void GenerateIndexFile(std::string fileName);
00033
00042     bool ReadIndexFile(std::string fileName);
00043
00049     void Add(KeyStruct keyStruct);
00050
```

```
00058     int Find(std::string key);
00059
00070     int BinarySearch(std::string str);
00071
00072    private:
00073     std::vector<KeyStruct> vKey;
00074     bool isSorted = false;
00075     IndexFileHeader header;
00076 };
```