CSC 331.001

Project 8 - External Hashing

Sam Jentsch

Account: CSC331119

Data files used

prog8.dat

```
8 blank 0 0.0
12345 Item06 45 14.2
12434 Item04 21 17.3
12382 Item09 62 41.37
34186 Item25 18 17.75
12165 Item16 30 7.69
16541 Item12 21 9.99
21212 Itme31 19 8.35
41742 Item14 55 12.36
```

Execution

Compile

```
[cs331119@cs Lab8]$ Lab8.sh



Mon Dec  1 20:11:27 CST 2014
```

Build

```
[cs331119@cs Lab8]$ Build
Welcome! This Builder program takes a path to a text record file and
        A. Creates a binary representation of the text record file
        B. Creates a hash table primary key list for the record file.
        C. Writes the hash table to a binary file.

        The Binary Record file created is in "BinaryDataFile.dat".
        The Primary Index file created is in "PrimaryIndex.ixp".

Please enter the path to the text data file now: ../../instr/prog8.dat

DATA FILE:
NUMBER OF RECORDS: 8
12345 Item06 45 14.2
12434 Item04 21 17.3
12382 Item09 62 41.37
34186 Item25 18 17.75
12165 Item16 30 7.69
16541 Item12 21 9.99
21212 Itme31 19 8.35
41742 Item14 55 12.36

TABLE:
INDEX: 0
        12345 -- Offset: 1
        12165 -- Offset: 5
INDEX: 1
        34186 -- Offset: 4
        16541 -- Offset: 6
INDEX: 2
        12382 -- Offset: 3
        21212 -- Offset: 7
        41742 -- Offset: 8
INDEX: 3
        EMPTY
INDEX: 4
        12434 -- Offset: 2


BinaryDataFile.dat and PrimaryKey.ixp created.
```

Search

```
[cs331119@cs Lab8]$ Search
Welcome! Please note that the builder program must be run to create the binary files before the search can take
place.
Enter the path to the primary index file (default from builder is PrimaryIndex.ixp): PrimaryIndex.ixp

TABLE:
INDEX: 0
        12345 -- Offset: 1
        12165 -- Offset: 5
INDEX: 1
        34186 -- Offset: 4
        16541 -- Offset: 6
INDEX: 2
        12382 -- Offset: 3
        21212 -- Offset: 7
        41742 -- Offset: 8
INDEX: 3
        EMPTY
INDEX: 4
        12434 -- Offset: 2


Enter a key to search for: 12165
Found Record: 12165 Item16 30 7.69


Search again? (Y/N): N
```

```
/*
 ---------------------------------------------------------------------------
 PROGRAM NAME: Project 8, External Hashing - Builder
 PROGRAMMER:   Samuel Jentsch
 CLASS:        CSC 331.001, Fall 2014
 INSTRUCTOR:   Dr. R. Strader
 REFERENCES:   C++ How to Program
 Paul Deitel & Harvey Deitel
 Dr. Strader: assignment information sheet

 PROGRAM PURPOSE:
 a. Retrieve file path to text record file from user.
 b. Create a binary representation of this file.
 c. Construct a hash table containing the primary key list for the
    record file.
 d. Save these files for use in the Search program.

 VARIABLE DICTIONARY:
 none
 ---------------------------------------------------------------------------
 */

#include <iostream>
#include <fstream>

#include "HashTable.h"
#include "ProductManager.h"

using namespace std;

int convertStringToInt(std::string intString);

int main(int argc, const char * argv[]) {

    cout << "Welcome! This Builder program takes a path to a text record file and \n\tA. Creates a binary
representation of the text record file" <<
    "\n\tB. Creates a hash table primary key list for the record file.\n\tC. Writes the hash table to a binary
file." << endl;
    cout << endl << "\tThe Binary Record file created is in \"BinaryDataFile.dat\".";
    cout << endl << "\tThe Primary Index file created is in \"PrimaryIndex.ixp\"." << endl << endl;
    cout << "Please enter the path to the text data file now: ";

    //Get the path to and open the text data file.
    ifstream dataStream;

    string filePath;
    cin >> filePath;
    dataStream.open(filePath.c_str(), ios::in);

    while (!dataStream.is_open()) {
        //File not found.
        cout << "A file with that path was not found. Please enter a new path to the data file: ";
        cin >> filePath;
        dataStream.open(filePath.c_str(), ios::in);
    }

    cout << endl;

    //Create binary files for writing.
    ofstream binCreate("BinaryDataFile.dat", fstream::out);
    binCreate.close();
    fstream binaryDataFile;
    binaryDataFile.open("BinaryDataFile.dat", fstream::ate | fstream::binary| fstream::out | fstream::in);

    ofstream indexCreate("PrimaryIndex.ixp", fstream::out);
    binCreate.close();
    fstream indexFile;
    indexFile.open("PrimaryIndex.ixp", fstream::ate | fstream::binary| fstream::out | fstream::in);

    //Hash Table for use throughout program
    HashTable hashTable;
    ProductManager productManager(binaryDataFile);

    string dataLine;
    string word;
```

```
    //Create a binary representation of the text data file.
    int lineNumber = 0;
    DataRecord newRecord;
    while (getline(dataStream, dataLine)) {
        newRecord = productManager.getDataRecordForString(dataLine);
        if (lineNumber > 0) {
            hashTable.addRecordToTable(newRecord, lineNumber);
        }
        productManager.writeDataRecordToBinaryFile(newRecord, lineNumber);
        lineNumber++;
    }

    productManager.traverseFile();
    cout << endl;
    hashTable.printHashTable();

    //Save the table to the PrimaryIndex.ixp file
    hashTable.saveTableToFile(indexFile);
    cout << endl << endl;

    //Close the files
    dataStream.close();
    binaryDataFile.close();
    indexFile.close();

    cout << "BinaryDataFile.dat and PrimaryKey.ixp created." << endl;
}

int convertStringToInt(std::string intString) {
    //-----------------------------------------------------------------------------//
    //Convert the string passed into an integer and return the integer value.
    //Preconditions: string representation of an integer passed as parameter.
    //Postconditions: the string passed is parsed and the integer value of the string
    //is returned to the caller.
    //-----------------------------------------------------------------------------//
    int convertedInteger;

    std::istringstream converter(intString);
    if (converter >> convertedInteger) {
        return convertedInteger;
    } else {
        return -1;
    }

}
```

```
/*
 ---------------------------------------------------------------------
 PROGRAM NAME: Project 8, External Hashing - Search
 PROGRAMMER:   Samuel Jentsch
 CLASS:        CSC 331.001, Fall 2014
 INSTRUCTOR:   Dr. R. Strader
 REFERENCES:   C++ How to Program
 Paul Deitel & Harvey Deitel
 Dr. Strader: assignment information sheet

 PROGRAM PURPOSE:
 a. Retrieve file path to index record file from user.
 b. Recreate the hash table containing the primary key list from the
    file specified.
 c. Prompt the user for a key to search for.
 d. Search the hash table for this key (by hashing the key and traversing
    the chain).
 e. If found, return the record from the binary representation of the
    record file created in the Builder program.

 VARIABLE DICTIONARY:
 none
 ---------------------------------------------------------------------
 */

#include <iostream>
#include <fstream>

#include "HashTable.h"
#include "ProductManager.h"

using namespace std;

int convertStringToInt(std::string intString);

int main(int argc, const char * argv[]) {
    fstream primaryIndexFile;
    //primaryIndexFile.open("PrimaryIndex.ixp", fstream::ate | fstream::binary| fstream::out | fstream::in);

    cout << "Welcome! Please note that the builder program must be run to create the binary files before the
search can take place." << endl;
    cout << "Enter the path to the primary index file (default from builder is PrimaryIndex.ixp): ";

    string indexFilePath;
    cin >> indexFilePath;
    primaryIndexFile.open(indexFilePath.c_str(), ios::in);

    while (!primaryIndexFile.is_open()) {
        //File not found.
        cout << "A file with that path was not found. Please enter a new path to the index file: ";
        cin >> indexFilePath;
        primaryIndexFile.open(indexFilePath.c_str(), ios::in);
    }

    fstream binDataFile;
    binDataFile.open("BinaryDataFile.dat", fstream::ate | fstream::binary| fstream::out | fstream::in);

    if (!binDataFile.is_open()) {
        cout << "The binary data file for the search was not found. Please run the Builder program and try
again." << endl;
        return -1;
    }

    ProductManager pManager(binDataFile);
    //pManager.traverseFile();

    cout << endl;

    HashTable readTable;
    readTable.readHashFile(primaryIndexFile);
    readTable.printHashTable();

    string searchKey;
    bool continueSearch = true;
    while (continueSearch) {
```

```
        cout << "\n\nEnter a key to search for: ";
        cin >> searchKey;

        while (convertStringToInt(searchKey) == -1) {
            //Invalid key entered.
            cout << "That was an invalid key. Please enter an integer value: ";
            cin >> searchKey;
        }

        int RRN = readTable.searchTableForPrimaryKey(convertStringToInt(searchKey));

        DataRecord foundRecord;
        if (RRN != -1) {
            cout << "Found Record: ";
            foundRecord = pManager.getDataRecordAtOffset(RRN);
            foundRecord.printRecord();
            cout << endl;
        } else {
            cout << "Record with key " << searchKey << " not found." << endl;
        }

        string response;
        cout << endl << "Search again? (Y/N): ";
        cin >> response;

        if (toupper(response[0]) != 'Y') {
            continueSearch = false;
        }
    }

    //Close file
    primaryIndexFile.close();
    binDataFile.close();
}

int convertStringToInt(std::string intString) {
    //-------------------------------------------------------------------------------//
    //Convert the string passed into an integer and return the integer value.
    //Preconditions: string representation of an integer passed as parameter.
    //Postconditions: the string passed is parsed and the integer value of the string
    //is returned to the caller.
    //-------------------------------------------------------------------------------//
    int convertedInteger;

    std::istringstream converter(intString);
    if (converter >> convertedInteger) {
        return convertedInteger;
    } else {
        return -1;
    }

}
```

HashTable.h

```
/*
 ----------------------------------------------------------------------
 CLASS NAME: HashTable
 PROGRAMMER:   Samuel Jentsch
 CLASS:        CSC 331.001, Fall 2014
 INSTRUCTOR:   Dr. R. Strader
 REFERENCES:   C++ How to Program
 Paul Deitel & Harvey Deitel
 Dr. Strader: assignment information sheet

 CLASS PURPOSE:
 a. A hash table containing nodes for implementing separate chaining as
 a collision resolution scheme.
 b. The class implements methods for adding words to the hash table and
 searching for a word in the hash table.
 c. Supports writing the hashtable to a passed file pointer.
 d. Supports reading the hashtable from a passed file pointer.

 VARIABLE DICTIONARY:

 table - RecordNode*, the array of Node references used to implement the
 hash table with separate chaining.

 numberOfItems - int, used to track the number of items in the hash table
                    to right to the header record for the binary file
                    containing the hash table.
 ----------------------------------------------------------------------
 */

#ifndef ___31_Project8__HashTable__
#define ___31_Project8__HashTable__

#include <stdio.h>
#include <fstream>

#include "RecordNode.h"
#include "DataRecord.h"

#define kHashSize 5

class HashTable {
private:
    RecordNode* table[kHashSize];
    int numberOfItems;
public:
    HashTable();

    /***Add***/
    void addRecordToTable(DataRecord newRecord, int lineNumber);

    /***Search***/
    int searchTableForPrimaryKey(int key);

    /***Hashing***/
    int hashRecord(DataRecord record);
    int hashRecordKey(int recordKey);

    /***Print***/
    void printHashTable();

    /***File Operations***/
    void saveTableToFile(std::fstream &fileOut);
    void readHashFile(std::fstream &fileIn);

    /***Create New Record Nodes for addition to hash table***/
    RecordNode* getNewNode(const DataRecord &record, const int &lineNumber);
    RecordNode* getNewNode(const int &key, const int &lineNumber);
};

#endif /* defined(___31_Project8__HashTable__) */
```

HashTable.cpp

```
#include "HashTable.h"

HashTable::HashTable() {
    for (int i = 0; i < kHashSize; i++) {
        table[i] = NULL;
    }

    numberOfItems = 0;
}

void HashTable::addRecordToTable(DataRecord newRecord, int lineNumber) {
    //-----------------------------------------------------------------------------------------------//
    //Add a record with a given primary key to the hash table. A new node
    //is created with the word and line number and added to the hashtable.
    //Preconditions: new DataRecord and lineNumber (offset) passed as parameters.
    //Postconditions: A RecordNode containing the primary key of the record passed and the offset
    //indicated by line number is created and added to the table.
    //-----------------------------------------------------------------------------------------------//
    int indexForRecord = hashRecord(newRecord);

    RecordNode *chain = table[indexForRecord];

    RecordNode *newNode = getNewNode(newRecord, lineNumber);

    if (table[indexForRecord] == NULL) {
        //Set table index to new node.
        table[indexForRecord] = newNode;
    } else {
        while (chain->getNext() != NULL) {
            chain = chain->getNext();
        }

        chain->setNext(newNode);
    }

    numberOfItems++;
}

int HashTable::hashRecord(DataRecord record) {
    //-----------------------------------------------------------------------------------------------//
    //Hash primary key of record passed using location = (key value) mod 5
    //Preconditions: Record to hash passed as parameter.
    //Postconditions: key is hashed and result is returned to caller.
    //-----------------------------------------------------------------------------------------------//
    return record.getKey() % 5;

}

int HashTable::hashRecordKey(int recordKey) {
    //-----------------------------------------------------------------------------------------------//
    //Hash primary key passed using location = (key value) mod 5
    //Preconditions: key to hash passed as parameter.
    //Postconditions: key is hashed and result is returned to caller.
    //-----------------------------------------------------------------------------------------------//
    return recordKey % 5;
}

int HashTable::searchTableForPrimaryKey(int key) {
    //-----------------------------------------------------------------------------------------------//
    //Search the hash table for the key passed as a parameter. The key is hashed and seached for
    //in separate chain. -1 is returned if not found, or the RRN for the record is returned if it
    //was found in the hash table.
    //Preconditions: Key to search for passed as parameter.
    //Postconditions: If key is found the RRN for the record is returned. Otherwise, -1 is returned
    //to the caller.
    //-----------------------------------------------------------------------------------------------//

    int recordRRN = -1;

    int hashIndex = hashRecordKey(key);
    //std::cout << "hashed to: " << hashIndex << std::endl;

    RecordNode *chain = table[hashIndex];
```

```
    if (table[hashIndex] != NULL) {

        while (chain != NULL && recordRRN == -1) {
           //std::cout << "Checking: " << chain->getKey() << std::endl;
            if (chain->getKey() == key) {
                recordRRN = chain->getRelativeOffset();
            }
            chain = chain->getNext();
        }

    }
    return recordRRN;
}

void HashTable::printHashTable() {
    //-------------------------------------------------------------------------------------------//
    //Display the contents of the hashtable to the console. Include the line numbers that each
    //Node's word is a part of.
    //Preconditions: hashtable table as class variable.
    //Postconditions: Hashtable contents are displayed to the console.
    //-------------------------------------------------------------------------------------------//
    std::cout << "TABLE: " << std::endl;
    for (int i = 0; i < kHashSize; i++) {
        RecordNode *chain = table[i];
        std::cout << "INDEX: " << i << std::endl;
        if (chain == NULL) {
            std::cout << "\tEMPTY" << std::endl;
        } else {
            while (chain != NULL) {
                std::cout << "\t" << chain->getKey();
                std::cout << " -- Offset: " << chain->getRelativeOffset() << std::endl;
                chain = chain->getNext();
            }
        }
    }

}

void HashTable::saveTableToFile(std::fstream &fileOut) {
    //-------------------------------------------------------------------------------//
    //Save the key list to file reference passed. The file is written so that each index
    //of the hash table ends with a RecordNode with the primary key set to -1. If an
    //index of the hash table is empty, an "empty" record (key set to -1) is written to
    //the file. Otherwise, each RecordNode contained in that index is written to the file,
    //followed by an empty record signifying the end of that index.
    //Preconditions: fileOut reference to file passed as parameter.
    //Postconditions: primary key list saved to file in format described above.
    //-------------------------------------------------------------------------------//
    fileOut.seekp(0);
    RecordNode headerRecord;
    headerRecord.setKey(numberOfItems);

    //Write out header record.
    fileOut.write(reinterpret_cast<char*>(&headerRecord), sizeof(RecordNode));

    RecordNode blankNode;
    blankNode.setKey(-1);
    RecordNode duplicate;
    for (int i = 0; i < kHashSize; i++) {
        //Write out for each hash index
        RecordNode *chain = table[i];

        while (chain != NULL) {
            duplicate.setKey(chain->getKey());
            duplicate.setRelativeOffset(chain->getRelativeOffset());
            //std::cout << "Writing chain with key: " << duplicate.getKey() << std::endl;
            fileOut.write(reinterpret_cast<char*>(&duplicate), sizeof(RecordNode));
            chain = chain->getNext();
        }
        fileOut.write(reinterpret_cast<char*>(&blankNode), sizeof(RecordNode));
    }
}

void HashTable::readHashFile(std::fstream &fileIn) {
    //-------------------------------------------------------------------------------------------//
    //Read the hash file into the HashTable containing the primary key list from the reference to
```

```
    //the file passed. The file is read using the following format:
    //  A record containing a -1 as the primary key is an empty index of the hash table.
    //  If a record containing a number not equal to -1 as its primary key is read, it is added to
    //      the hash table at the current index. Each subsequent non -1 record is added to that
    //      index until another -1 is encountered.
    //Preconditions: fileIn reference to file passed as parameter.
    //Postconditions: Contents of the HashTable representing the primary key list
    //are read from the file.
    //-------------------------------------------------------------------------------------------//

    fileIn.seekp(0);
    RecordNode headerRecord;

    //Read in header record.
    fileIn.read(reinterpret_cast<char*>(&headerRecord), sizeof(RecordNode));

    RecordNode readNode;
    for (int i = 0; i < kHashSize; i++) {
        fileIn.read(reinterpret_cast<char*>(&readNode), sizeof(RecordNode));

        while (readNode.getKey() != -1) {
            //std::cout << "INDEX: " << i << " KEY: " << readNode.getKey() << std::endl;
            RecordNode *chain = table[i];

            RecordNode *newNode = getNewNode(readNode.getKey(), readNode.getRelativeOffset());

            if (table[i] == NULL) {
                //Set table index to new node.
                table[i] = newNode;
            } else {
                while (chain->getNext() != NULL) {
                    chain = chain->getNext();
                }

                chain->setNext(newNode);
            }

            fileIn.read(reinterpret_cast<char*>(&readNode), sizeof(RecordNode));
        }
    }
}

RecordNode* HashTable::getNewNode(const DataRecord &record, const int &lineNumber) {
    //-----------------------------------------------------------------//
    //Initialize node and return to caller.
    //Preconditions: DataRecord and LineNumber (offset) passed as parameters
    //Postconditions: Node initialized and returned to caller
    //-----------------------------------------------------------------//
    return new RecordNode(record.getKey(), lineNumber);
}

RecordNode* HashTable::getNewNode(const int &key, const int &lineNumber) {
    //-----------------------------------------------------------------//
    //Initialize node and return to caller.
    //Preconditions: key and lineNumber (offset) passed as parameters
    //Postconditions: Node initialized and returned to caller
    //-----------------------------------------------------------------//
    return new RecordNode(key, lineNumber);
}
```

RecordNode.h

```
/*
 --------------------------------------------------------------------------
 CLASS NAME: RecordNode
 PROGRAMMER:   Samuel Jentsch
 CLASS:        CSC 331.001, Fall 2014
 INSTRUCTOR:   Dr. R. Strader
 REFERENCES:   C++ How to Program
 Paul Deitel & Harvey Deitel
 Dr. Strader: assignment information sheet

 CLASS PURPOSE:
 a. A node containing a key, relative offset, and a reference to the next
    node in the list.
 b. Represents a primary key entry in the primary key list.

 VARIABLE DICTIONARY:

 key - int, contains the key for the record.

 relativeOffset - int, contains the relative offset for the record
                  referenced by the index record

 next - Node, reference to next Node in the list.
 --------------------------------------------------------------------------
 */

#ifndef ___31_Project8__RecordNode__
#define ___31_Project8__RecordNode__

#include <stdio.h>

class RecordNode {
private:
    int key;
    int relativeOffset;

    RecordNode *next;
public:
    RecordNode();
    RecordNode(int key, int RRN);

    void setKey(int key);
    int getKey() const;

    //***relativeOffset***//
    void setRelativeOffset(int relativeOffset);
    int getRelativeOffset() const;

    void setNext(RecordNode *next);
    RecordNode* getNext() const;
};

#endif /* defined(___31_Project8__RecordNode__) */
```

RecordNode.cpp

```
#include "RecordNode.h"

RecordNode::RecordNode()
: next(NULL)
{
    setKey(0);
    setRelativeOffset(0);
}

RecordNode::RecordNode(int key, int RRN)
: next(NULL)
{
    setKey(key);
    setRelativeOffset(RRN);
}

#pragma mark - Getters/Setters
void RecordNode::setKey(int key) {
    //-------------------------------------------------------------------------------//
    //Preconditions: key parameter passed as integer.
    //Postconditions: the key attribute is set to the parameter passed.
    //-------------------------------------------------------------------------------//
    RecordNode::key = key;
}

int RecordNode::getKey() const {
    //-------------------------------------------------------------------------------//
    //Postconditions: key is returned to the calling method.
    //-------------------------------------------------------------------------------//
    return key;
}

void RecordNode::setRelativeOffset(int relativeOffset) {
    //-------------------------------------------------------------------------------//
    //Preconditions: relativeOffset parameter passed as integer.
    //Postconditions: the relativeOffset attribute is set to the parameter passed.
    //-------------------------------------------------------------------------------//
    RecordNode::relativeOffset = relativeOffset;
}

int RecordNode::getRelativeOffset() const {
    //-------------------------------------------------------------------------------//
    //Postconditions: RRN is returned to the calling method.
    //-------------------------------------------------------------------------------//
    return relativeOffset;
}


void RecordNode::setNext(RecordNode *next) {
    //-------------------------------------------------//
    //Preconditions: RecordNode next passed as parameter.
    //Postconditions: Next RecordNode is set to parameter passed.
    //-------------------------------------------------//
    RecordNode::next = next;
}

RecordNode* RecordNode::getNext() const {
    //-------------------------------------------------//
    //Preconditions: none
    //Postconditions: next is returned to caller.
    //-------------------------------------------------//
    return next;
}
```

ProductManager.h

```
/*
 --------------------------------------------------------------------------------
 CLASS NAME: ProductManager
 PROGRAMMER:   Samuel Jentsch
 CLASS:        CSC 331.001, Fall 2014
 INSTRUCTOR:   Dr. R. Strader
 REFERENCES:   C++ How to Program
 Paul Deitel & Harvey Deitel
 Dr. Strader: assignment information sheet

 CLASS PURPOSE:
 Product Manager is responsible for all actions associated with the binary data
 file, including:
 - Creating the binary representation of the text data file.
 - Inserting data records.
 - Deleting data records.
 - Retrieving records at a specified index.
 - Converting a sting representing a Data Record into an instance of the Data
 Record class for use.
 - Traversing the data record file for display to the console.
 - Maintaining the header of the data record file for the number of items.

 VARIABLE DICTIONARY:

 &binaryRecordFile, fstream - a reference to the binary data record file that will
 be searched, inserted into, retrieved from, etc.
 headerNumber - int, a global count of the number of binary records in the data
 record file.
 --------------------------------------------------------------------------------
 */

#ifndef ___31_Project8__ProductManager__
#define ___31_Project8__ProductManager__

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

class ProductManager {
private:
    std::fstream &binaryRecordFile;
    int headerNumber;
public:
    ProductManager(std::fstream &binaryRecordFile);

    //***Data Record Binary File Reader***//
    DataRecord getDataRecordAtOffset(int offset);
    int getNumberOfRecords();

    //***Data Record Binary File Writer***//
    void writeDataRecordToBinaryFile(DataRecord newRecord, int line);
    void updateHeader(int numberOfRecords);

    //***Add***//
    int addDataRecordToBinaryFile(DataRecord record);

    //***Display***//
    void traverseFile();

    /***Get Record from File***/
    DataRecord getRecordWithKey(int key);

    //***Conversion Methods***//
    DataRecord getDataRecordForString(std::string record);
    int convertStringToInt(std::string intString);
    double convertStringToDouble(std::string doubleString);
    std::vector<std::string> split(const std::string &s, char delim, std::vector<std::string> &elems);
};
#endif /* defined(___31_Project8__ProductManager__) */
```

ProductManager.cpp

```
#include "ProductManager.h"

#include "ProductManager.h"

#pragma mark - Constructor
ProductManager::ProductManager(std::fstream &binaryRecordFile)
:binaryRecordFile(binaryRecordFile)
{

}


#pragma mark - Read From File
DataRecord ProductManager::getDataRecordAtOffset(int offset) {
    //------------------------------------------------------------------------------//
    //Read in a data record from the data file at the offset passed. Return this record
    //to the caller as an instance of the DataRecord class.
    //Preconditions: an offset to read a record from binaryRecordFile class variable.
    //Postconditions: an instance of DataRecord is instantiated using the data at the
    //record passed and returned to the caller.
    //------------------------------------------------------------------------------//
    binaryRecordFile.seekg(0);
    binaryRecordFile.seekg(offset * sizeof(DataRecord));

    DataRecord record;
    binaryRecordFile.read(reinterpret_cast<char*>(&record), sizeof(DataRecord));

    return record;
}

void ProductManager::traverseFile() {
    //------------------------------------------------------------------------------//
    //Traverse the entire binary record file from beginning to end. Display the contents
    //of the file to the console, beginning with the header record indicating the number
    //of records in the file (also used to determine the number of records to iterate
    //through for printing).
    //Preconditions: binaryRecordFile class variable with valid header record.
    //Postconditions: the entire binaryRecordFile is traversed and displayed to the
    //console.
    //------------------------------------------------------------------------------//
    binaryRecordFile.seekg(0);

    DataRecord header;
    std::cout<<"DATA FILE: " << "\n";
    int numberOfRecords = getNumberOfRecords();
    std::cout << "NUMBER OF RECORDS: " << numberOfRecords <<"\n";
    DataRecord record;
    for (int i = 0; i < numberOfRecords; i++) {
        //std::cout << "READING: ";
        binaryRecordFile.read(reinterpret_cast< char * >(&record), sizeof(DataRecord));
        record.printRecord();
    }
}

int ProductManager::getNumberOfRecords() {
    //------------------------------------------------------------------------------//
    //Seek to the beginning of the binary record file and read in the header record.
    //The header record is read and converted into an integer. This integer is returned
    //to the caller (the number of records in the binary file.
    //Preconditions: binaryRecordFile class variable
    //Postconditions: the header record is read from the file, converted to an integer,
    //and returned to the caller.
    //------------------------------------------------------------------------------//
    DataRecord header;

    binaryRecordFile.seekg(0);
    binaryRecordFile.read(reinterpret_cast< char * >(&header), sizeof(DataRecord));

    int numberOfRecords = header.getKey();

    headerNumber = numberOfRecords;
    //std::cout << "SETTING HEADER NUMBER TO: " <<headerNumber << std::endl;
    return numberOfRecords;
```

```
}

#pragma mark - Write To File

void ProductManager::writeDataRecordToBinaryFile(DataRecord newRecord, int line) {
    //-------------------------------------------------------------------------------//
    //Duplicate the text record to a fixed field and write the record to the binary
    //file.
    //Preconditions: record string and line to write to passed as parameters.
    //Postconditions: record string converted to data record and written to file.
    //-------------------------------------------------------------------------------//
    //DataRecord newRecord = getDataRecordForString(record);
    std::cout << "INSERTING RECORD: ";
    newRecord.printRecord();

    int recordOffset = sizeof(DataRecord) * line;

    binaryRecordFile.seekp(recordOffset);
    binaryRecordFile.write(reinterpret_cast< char * >(&newRecord), sizeof(DataRecord));
}

void ProductManager::updateHeader(int numberOfRecords) {
    //-------------------------------------------------------------------------------//
    //Update the header record of the binary record file.
    //Preconditions: global headerNumber representing the previous number of records.
    //Postconditions: header record is updated by incrementing the header number. The
    //header is stored in the key field of a datarecord and written as the first record
    //in the binary record file.
    //-------------------------------------------------------------------------------//
    DataRecord header;
    header.setKey(++headerNumber);
    std::cout<<"Setting header to: " << header.getKey() << " Header Number is: " << headerNumber << "\n";
    binaryRecordFile.seekp(0);
    binaryRecordFile.write(reinterpret_cast< char * >(&header), sizeof(DataRecord));
}

int ProductManager::addDataRecordToBinaryFile(DataRecord record) {
    //-------------------------------------------------------------------------------//
    //Update the header record of the binary record file.
    //Preconditions: global headerNumber representing the previous number of records.
    //Postconditions: header record is updated by incrementing the header number. The
    //header is stored in the key field of a datarecord and written as the first record
    //in the binary record file.
    //-------------------------------------------------------------------------------//
    int RRN = getNumberOfRecords() + 1;
    writeDataRecordToBinaryFile(record, RRN);
    updateHeader(getNumberOfRecords()+1);
    return RRN;
}

#pragma mark - Conversion and Split Methods

DataRecord ProductManager::getDataRecordForString(std::string record) {
    //-------------------------------------------------------------------------------//
    //Convert the string passed into a data record. Each field is checked during
    //conversion to see if it is valid, if it is not a message is displayed to the
    //console and -1 is set as the key field of the data record. If the record is valid,
    ///the new record is returned to the caller.
    //Preconditions: record to convert passed as a parameter.
    //Postconditions: The DataRecord represented by the string passed is created and
    //returned or the errors that occurred are displayed to the user.
    //-------------------------------------------------------------------------------//
    //Get the record elements split by space.
    std::vector<std::string> splitRecord;
    splitRecord = split(record, ' ', splitRecord);

    //splitRecord[0] = key
    //splitRecord[1] = name
    //splitRecord[2] = code
    //splitRecord[3] = cost

    int key = convertStringToInt(splitRecord.at(0));
    double cost = convertStringToDouble(splitRecord.at(3));
    const char *name = splitRecord.at(1).c_str();
    int code = convertStringToInt(splitRecord.at(2));
```

```
    if (key == -1) {
        //Invalid key passed.
        std::cout << "Invalid Primary Key. Must be integer.\n";
    } if(splitRecord.at(1).size() > 8) {
        //Invalid description passed.
        std::cout << "Invalid Description. Can be up to 8 characters.\n";
        key = -1;
        name = "INVALID";
    } if (cost < 0) {
        std::cout << "Invalid Cost. Must be double value.\n";
        key = -1;
        cost = -1;
    } if (code < 0) {
        std::cout << "Invalid Code. Must be integer value.\n";
        key = -1;
        code = -1;
    }


    DataRecord newRecord(key, name, code, cost);

    return newRecord;
}

int ProductManager::convertStringToInt(std::string intString) {
    //-------------------------------------------------------------------------------//
    //Convert the string passed into an integer and return the integer value.
    //Preconditions: string representation of an integer passed as parameter.
    //Postconditions: the string passed is parsed and the integer value of the string
    //is returned to the caller.
    //-------------------------------------------------------------------------------//
    int convertedInteger;

    std::istringstream converter(intString);
    if (converter >> convertedInteger) {
        return convertedInteger;
    } else {
        return -1;
    }

}

double ProductManager::convertStringToDouble(std::string doubleString) {
    //-------------------------------------------------------------------------------//
    //Convert the string passed into a double and return the double value.
    //Preconditions: string representation of a double passed as parameter.
    //Postconditions: the string passed is parsed and the double value of the string
    //is returned to the caller.
    //-------------------------------------------------------------------------------//
    double convertedDouble;

    std::istringstream converter(doubleString);
    if (converter >> convertedDouble) {
        return convertedDouble;
    } else {
        return -1;
    }

    return convertedDouble;
}

std::vector<std::string> ProductManager::split(const std::string &s, char delim, std::vector<std::string>
&elems) {
    //-------------------------------------------------------------------------
    //Split a string using the delimeter specified. The vector that will contain
    //the split string is passed by reference
    //Precondition: a string to split, delimeter to split by, and vector to store
    //the split strings in passed as parameters.
    //Postcondition: The method splits the string by the parameter and stores it
    //in the elems vector. This vector is also returned to the caller.
    //-------------------------------------------------------------------------

    std::stringstream ss(s);
    std::string item;
    while (getline(ss, item, delim)) {
        elems.push_back(item);
```

```
    }
    return elems;
}
```

DataRecord.h

```
/*
 --------------------------------------------------------------------------
 CLASS NAME: DataRecord
 PROGRAMMER:   Samuel Jentsch
 CLASS:        CSC 331.001, Fall 2014
 INSTRUCTOR:   Dr. R. Strader
 REFERENCES:   C++ How to Program
 Paul Deitel & Harvey Deitel
 Dr. Strader: assignment information sheet

 CLASS PURPOSE:
 a. This class is used to hold each of the attributes for a data record.

 Records are in the form:

 INT CHAR[8] INT DOUBlE
 KEY DESCRIPTION COUNT NEXTPART

 VARIABLE DICTIONARY:

 key - int, contains the primary key for the record.
 name - char[8], an 8 character text name for the record.
 code - int, the code associated with the record.
 cost - double, the cost associated with the record.
 --------------------------------------------------------------------------
 */


#ifndef ___31_Project4__DataRecord__
#define ___31_Project4__DataRecord__

#include <stdio.h>
#include <iostream>

class DataRecord {
private:
    int key;
    char name[8];
    int code;
    double cost;
public:
    DataRecord();
    DataRecord(int key, const char *name, int code, double cost);

    //***key***//
    void setKey(int key);
    int getKey() const;

    //***name***//
    void setName(const char *name);
    std::string getName();

    //***code***//
    void setCode(int code);
    int getCode() const;

    //***cost***//
    void setCost(double cost);
    double getCost() const;

    void printRecord();
};

#endif /* defined(___31_Project4__DataRecord__) */
```

DataRecord.cpp

```
#include "DataRecord.h"

#pragma mark - Constructors
DataRecord::DataRecord() {
    //Default constructor.
    //Initialize data members to default values.
    setCost(0);
    setCode(0);
    setKey(0);

    char blankName[8] = {};
    setName(blankName);
}

DataRecord::DataRecord(int key, const char* name, int code, double cost)
:key(key), code(code), cost(cost)
{
    char blankName[8] = {};
    setName(blankName);
    setName(name);
}

#pragma mark - Getters/Setters
void DataRecord::setKey(int key) {
    //-------------------------------------------------------------------------//
    //Preconditions: key parameter passed as integer.
    //Postconditions: the key attribute is set to the parameter passed.
    //-------------------------------------------------------------------------//

    DataRecord::key = key;
}

int DataRecord::getKey() const {
    //-------------------------------------------------------------------------//
    //Postconditions: key is returned to the calling method.
    //-------------------------------------------------------------------------//

    return key;
}

void DataRecord::setName(const char *nameString) {
    //-------------------------------------------------------------------------//
    //Preconditions: count parameter passed as integer.
    //Postconditions: the count attribute is set to the parameter passed.
    //-------------------------------------------------------------------------//

    char c = nameString[0];
    int count = 0;
    while (isalnum(c)) {
        name[count] = c;
        c = nameString[++count];
    }
    name[count] = '\0';
}

std::string DataRecord::getName() {
    //-------------------------------------------------------------------------//
    //Preconditions: description string pointer passed as string
    //Postconditions: a string containing the values of the class attribute description
    //is created and returned to the caller.
    //-------------------------------------------------------------------------//

    std::string nameString(name);

    return nameString;
}

void DataRecord::setCode(int code) {
    //-------------------------------------------------------------------------//
    //Preconditions: count parameter passed as integer.
    //Postconditions: the count attribute is set to the parameter passed.
    //-------------------------------------------------------------------------//

    DataRecord::code = code;
```

```
}

int DataRecord::getCode() const {
    //-------------------------------------------------------------------------------//
    //Postconditions: count is returned to the calling method.
    //-------------------------------------------------------------------------------//

    return code;
}

void DataRecord::setCost(double cost) {
    //-------------------------------------------------------------------------------//
    //Preconditions: nextPart parameter passed as integer.
    //Postconditions: the nextPart attribute is set to the parameter passed.
    //-------------------------------------------------------------------------------//

    DataRecord::cost = cost;
}

double DataRecord::getCost() const {
    //-------------------------------------------------------------------------------//
    //Postconditions: nextPart is returned to the calling method.
    //-------------------------------------------------------------------------------//

    return cost;
}

#pragma mark - Print
void DataRecord::printRecord() {
    //-------------------------------------------------------------------------------//
    //Preconditions: none
    //Postconditions: the attributes of the DataRecord instance are printed to the
    //console.
    //-------------------------------------------------------------------------------//

    std::cout << key << " " << name << " " << code << " " << cost << std::endl;
}
```