Program 1:

Name: Amundson, Brendon

Runtime:

Command Line Arguments		____ 5 Points
Name		____ 5 Points
Address		____ 5 Points
City		____ 5 Points
State		
	Code	____ 5 Points
	Capital	
Zip Code		
	5 digit	~~10~~ 5 Points
	4 digit	0 5 Points
Dates		
	Month	4-5 15 Points
	Day	
	Year	
	Expiration after	____ 5 Points
Radio Class		____ 5 Points
Call Sign		
	Characters	0 5 Points
	Middle digit	____ 5 Points
	Not uppercase	0 5 Points

Runtime sub total 55 **75 Points**

Specifications

Read till EOF was encountered processing		
One record at a time	____	10 Points
Errors indented, one record – multiple errors under it		
Proper Error Messages	5	10 Points
Used one read and one write to fill structure	____	5 Points

did it display record

SubTotal 20 **25 Points**

Program 1 Runtime and specification total 75 **100 Points**

Documentation and coding standards **(minus)** 5 **Points**

Final Program Grade 70 **100 Points**

Program Header (40 Points)

Main Page & @file (5 Points) _____
Course Information(10 Points)
- Author _____
- Date Due _____
- Professor _____
- Course _____
- Location _____

Program Information (10 Points)
- Details _____

Compiling (5 Points)
- Compile & usage _____

Todo, Bugs (10 Points)
- Bugs _____
- Todo _____

Subtotal _____ (40)

Function Header (10 Points)

Author _____
Description _____
Params (when needed) _____
Returns (when needed) _____
Subtotal _____ (10)

Others (15 Points)

Meaningful variable names _____
Whitespace around _____
- functions _____
- blocks of code & operators _____

Comments, -5
- must be adequate to cover the algorithm if the code was removed

Documented _____
- Global Constants _____
- Structures _____
- Enums _____
- Typedefs _____

-5
Subtotal 10 (15)

Other items not mentioned above

Functions.cpp should be there missing
_____ (-)

Multiple Files and files named correctly (10 Points) _____
Global Variables (10 Points per variable) (Max of 30 Points) _____
Bugs (30 Points) if it fails on given data and no bugs are recorded _____
Compiler Warnings (5 points per warning) (max 20 Points) _____
80 character line limit (10 Points) _____

Total Documentation (max 100 Points) 85

```
================================================================================
Name:                Brenda K Kowalke
Address:             1964 Christenson Rd.
City State Zip:      Rapid City, Sd 57702-1965
Birthdate:           2/4/1983
Licensing Dates:     1/15/1999 - 1/15/2015
Class - Callsign:    N - NZ4DZ

================================================================================
Name:                Invalid Char Call Case Ignore
Address:             934 Graceland
City State Zip:      Pheonix, Az 34583-8833
Birthdate:           4/14/1993
Licensing Dates:     3/14/2005 - 3/15/2008
Class - Callsign:    p - N58RT

================================================================================
Name:                Call Sign not uppercase
Address:             3 Roundup
City State Zip:      Murdo, Sd 58702-9873
Birthdate:           6/15/1979
Licensing Dates:     6/18/1991 - 6/19/1997
Class - Callsign:    x - NT8Tz
```

Invalid

Invalid

missing 3

```
Roger L. Schrader
     Invalid 4 digit zip code

Toni Logan
     Invalid first character in call sign

Ed Greenfield Jr.
     Invalid day in Birthday

Don Busack Sr.
     Invalid day in License

Bad [L] Name
     Invalid character in the name field

Bad Address
     Invalid character in the address field
     Invalid day in Birthday
     Invalid first character in call sign

Bad City
     Invalid character in the city field
     Invalid 4 digit zip code

Bad State
     Invalid State Code
     Invalid day in Birthday

Bad five zip code
     Invalid 5 digit zip code
     Invalid day in Expiration

Bad four zip code
     Invalid 4 digit zip code

Invalid Months
     Invalid month in Birthday
     Invalid month in License
     Invalid month in Expiration

Invalid Days
     Invalid 4 digit zip code
     Invalid day in Birthday
     Invalid first character in call sign

Invalid Years
     Invalid 4 digit zip code
     Invalid year in Birthday
     Invalid year in License
     Invalid year in Expiration

Expiration after License
     Expiration Date is not after the License Date
     Invalid first character in call sign

Invalid Radio Class
     Invalid 4 digit zip code
     Radio Class code is invalid

First Char Call Sign Invalid
     Invalid first character in call sign

Invalid Digit in Call Sign
     Invalid day in Expiration
     Invalid digit in call sign

State code not upper case
     First character of State Code not capitalized
     Invalid 4 digit zip code
```



```
N,A,C,BD Invalid
     Invalid character in the name field
     Invalid character in the address field
     Invalid character in the city field
     Invalid 4 digit zip code
     Invalid month in Birthday
     Invalid day in Birthday
     Invalid year in Birthday
```

```
/*****************************************************************************//**
 * @file
 * @brief This file calls the function to open the neccessary files for this,
 * program
 * in the database file.
 *
 * @mainpage Program 1 - Database Checker
 *
 * @section course_section Course Information
 *
 * @authors Brandon Amundson
 *
 * @date February 27, 2014
 *
 * @par Instructor:
 *         Roger Schrader
 *
 * @par Course:
 *         CSC 250 - Section 1 - 11:00 am
 *
 * @par Location:
 *         Classrom Building - Room 204W
 *
 * @section program_section Program Information
 *
 * @details
 * This program takes a database file and opens it up for reading.  Then the
 * program processes each record in the database file and checks for errors.
 * If there are any errors, they are output to a text file.  If there are no
 * errors, they are output to a seperate file.
 *
 * To keep track of all errors, there is a string variable called checkFalse
 * where each portion is changed to a one for any errors, if there are no
 * errors in that part of the record, it is changed to a zero.  This program
 * checks the record structure that consists of a name, address, city, state,
 * zipcode, birthdate, license date, expiration date, radio class, and a
 * callsign.  This program was create essentially to check the database of the
 * FCC's licensed ham radio operators for errors as they transistion from an
 * older system that did not do errorchecking upon entry to a new system that
 * will.
 *
 * @section compile_section Compiling and Usage
 *
 * @par Compiling Instructions:
 *      None
 *
 * @par Usage:
   @verbatim
   c:\> prog1.exe data.bin good.bin bad.txt
             data.bin is the database file that needs to be checked for errors.
             good.bin is the binary file to output the correct records to.
             bad.txt is the text file to output all records that have errors.
   @endverbatim
 *
 * @section todo_bugs_modifications_section Todo, Bugs, and Modifications
 *
 *
 * @par Modifications and Developing Timeline:
   @verbatim
   Date          Modification
   ------------  --------------------------------------------------------------
   February 10   Have ability to open and close files and begin to read in
                 structure
   February 14   Be able to check the name and city fields for errors as well
                 as check the address for errors.
   February 18   Be able to successfully extract date and zipcode and check
                 them for errors.
   February 20   Check the Call Sign for errors
   February 25   Output all errors and all good records.
```



```
   February 26   Fix all bugs.
   February 27   Document Program
   @endverbatim
 *
 *****************************************************************************/

#pragma pack(1)
#include "functions.h"

/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the main function of the program.  It calls the functions to read
 * and write from, and to files, it also calls the function that calls the
 * function that checks for errors in all portions of the records in the
 * database file.
 *
 * @param[in]        argc - argument count from when the program was started.
 * @param[in]        argv - contains the arguments from when the program was
 *                          started.
 *
 * @returns 0 - program ran successfully.
 * @returns 10 - Not enough command line arguments.
 *
 *****************************************************************************/
int main(int argc, char *argv[])
{
    ifstream fin;
    ofstream good, bad;
    Record record;
    string countFalse;
    if (argc == 4)
    {
        openFiles(fin, good, bad, argv);
    }
    else
    {
        cout << "Please run as prog1.exe, database file, good file, bad file.";
        return(10);
    }

    while (fin.read((char*)&record, sizeof(Record)))
    {
        callFunctions(record, countFalse);
        errorCheck(countFalse, good, bad, record);
        countFalse.clear();
    }

    closeFiles(fin, good, bad);
    return(0);
}
```

```
/*****************************************************************************//**
 * @file
 *
 * @brief This file contains all the functions required to read and write from
 * files and also contains the functions that check for errors in the database
 * file.
 *
 * @author Brandon Amundson
 *
 *****************************************************************************/

#pragma pack(1)
#ifndef __FUNCTIONS__H__
#define __FUNCTIONS__H__
#define _CRT_SECURE_NO_DEPRICATE
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <algorithm>
#include <cstring>
#include <string>
#include <cctype>
using namespace std;

/**
 * @brief Structure to hold all of the client data.
 */
struct Record
{
    char name[30];              /*!< the name of the client*/
    char address[30];           /*!< the address of the client*/
    char city[28];              /*!< the city the client lives in*/
    char state[2];              /*!< the state the client lives in*/
    unsigned int zipCode;       /*!< the zipcode of the area the client lives*/
    unsigned int birthDate;     /*!< the birthdate of the client*/
    unsigned int licensedDate;  /*!< the license date of the client*/
    unsigned int expirationDate;/*!< the expiration date of the client's record*
/
    char radioClass;            /*!< the classification of the client*/
    char callSign[5];           /*!< the callSign of the client*/
};

/**
 * @brief The valid States that can be in the state field of the Record Structur
e
 */
const string validStates[50] = {"AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DE
",
                                "FL", "GA", "HI", "ID", "IL", "IN", "IA", "KS",
                                "KY", "LA", "ME", "MD", "MA", "MI", "MN", "M
S",
                                "MO", "MT", "NE", "NV", "NH", "NJ", "NM", "N
Y",
                                "NC", "ND", "OH", "OK", "OR", "PA", "RI", "SC"
,
                                "SD", "TN", "TX", "UT", "VT", "VA", "WA", "WV
",
                                "WI", "WY" };

/**
* @brief The valid characters that can be in the address field of the Record
*        Structure
*/
const string validAddress =
       { "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz#. " };

/**
* @brief The valid characters that can be in the name and city field of the
*        Record Structure
```



```
*/
const string validNames =
                { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz. " };

/**
* @brief The valid days for each month that can be in the date field of the
*        Record Structure
*/
const int validDay[20] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
                           31, 31, 31 };

/**
* @brief The valid characters that can be in the radioClass field of the
*        Record Structure
*/
const string validClass = {"npatgxNPATGX"};

/**
* @brief The valid characters that can be in the callSign field of the
*        Record Structure
*/
const string validChar = {"KWNkwn"};

/**
 * @brief The mask used to extract the four digit portion of the Zip Code from
 *        the zipCode field of the record structure.
 */
const int DIG4MASK = 16383;

/**
* @brief The mask used to extract the five digit portion of the Zip Code from
*        the zipCode field of the record structure.
*/
const int DIG5MASK = 262143;

/**
 * @brief This is the mask that is used to extract the day portion of the date
 *        from the date portions of the Record structure.
 */
const int dayMask = 31;

/**
* @brief This is the mask that is used to extract the month portion of the date
*        from the date portions of the Record structure.
*/
const int monthMask = 15;

/**
* @brief This is the mask that is used to extract the year portion of the date
*        from the date portions of the Record structure.
*/
const int yearMask = 4095;

void openFiles(ifstream &fin, ofstream &good, ofstream &bad, char **commandline)
;
void callFunctions(Record record, string &countFalse);
void nameCheck(string name, string &countFalse, int i);
void numCheck(string address, string &countFalse);
void state(string state, string &countFalse);
void zipCheck(unsigned int zip, string &countFalse);
void extractDate(unsigned int date, string &countFalse, int &i, string &format);
void monthCheck(int month, string &countFalse, int i);
void dayCheck(int day, int month, string &countFalse, int i);
void yearCheck(int year, string &countFalse, int i);
void checkIfAfter(string &countFalse, string license, string expire);
void classCheck(string radio, string &countFalse);
void sign(string sign, string &countFalse);
void errorCheck(string &countFalse, ofstream &good, ofstream &bad, Record record
);
```

```
void outputError(string &countFalse, ofstream &bad, Record record);
void outputGood(ofstream &good, Record record);
void closeFiles(ifstream &fin, ofstream &good, ofstream &bad);
#endif
```

```
/*****************************************************************************//**
 * @file
 *
 * @brief This file contains all the functions required to read and write from
 * files and also contains the functions that check for errors in the database
 * file.
 *
 * @author Brandon Amundson
 *
 ******************************************************************************/

#pragma pack(1)
#include "functions.h"


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the function that opens the files for writing the good records to a
 * specified binary file and the bad records to a specified text file.  All
 * records are read in from a specified binary database file.  If this function
 * is unable to open any file, it exits with a code one.
 *
 *
 * @param[in, out]       fin - Input File Stream to be opened.
 *                             Used to read in contents from binary file to be
 *                             checked for accuracy.
 * @param[in, out]      good - Output File Stream to be opened.  Used to output
 *                             good records to a binary file.
 * @param[in, out]       bad - Output File Stream to be opened.  Used to output
 *                             bad records to a text file.
 * @param[in]    commandline - commandline arguments of what filenames to use
 *                             are to be opened.
 ******************************************************************************/
void openFiles(ifstream &fin, ofstream &good, ofstream &bad, char **commandline)
{
        fin.open(commandline[1], ios::in | ios::binary);
        if (!fin)
        {
            cout << "Unable to open " << commandline[1] << ". Closing Program Now"
                ;
            exit(1);
        }
        good.open(commandline[2], ios::out | ios::trunc | ios::binary);
        if (!good)
        {
            cout << "Unable to open " << commandline[2] << ". Closing Program Now"
                ;
            exit(1);
        }
        bad.open(commandline[3], ios::out | ios::trunc);
        if (!bad)
        {
            cout << "Unable to open " << commandline[3] << ". Closing Program Now"
                ;
            exit(1);
        }
}


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the function that calls all of the other functions that do not
 * involve opening or closing files and do not deal with outputting the good or
 * bad records that are read in from a file.
 *
```



```
 * @param[in]          record - This is passed in to allow access to the records
 *                              that were read into the structure in main.
 *                              Program breaks if this is declared as a local
 *                              variable.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                              errors found in the database file.  Each portion
 *                              is set to a 1 if there are any errors.
 ******************************************************************************
/
void callFunctions(Record record, string &countFalse)
{
    int i = 0;
    string birth;
    string license;
    string expire;
    countFalse.resize(30);
    nameCheck(record.name, countFalse, i);
    i += 2;
    numCheck(record.address, countFalse);
    nameCheck(record.city, countFalse, i);
    state(record.state, countFalse);
    zipCheck(record.zipCode, countFalse);
    i = 7;
    extractDate(record.birthDate, countFalse, i, birth);
    extractDate(record.licensedDate, countFalse, i, license);
    extractDate(record.expirationDate, countFalse, i, expire);
    checkIfAfter(countFalse, license, expire);
    classCheck((char*) &record.radioClass, countFalse);
    sign(record.callSign, countFalse);
}


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the function that checks for the correct set of characters,
 * (ie. letters a-z both uppercase and lowercase, and spaces or periods) in the
 * name field and is used on the name and city portion of the record.
 *
 * @param[in]            name - This typecasts the name and city portions of the
 *                              record from a character array to a string
 *                              allowing the usage of the string class member
 *                              functions.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                              errors found in the database file.  Each portion
 *                              is set to a 1 if there are any errors.
 * @param[in]               i - This variable is used to keep track of how many
 *                              times a function is called and to help keep
 *                              track of all errors.
 ******************************************************************************/
void nameCheck(string name, string &countFalse, int i)
{
    int idx;

    idx = name.find_first_not_of(validNames);
    if (idx != name.npos)
    {
        countFalse[i] = 1;
    }
    else
        countFalse[i] = 0;
}


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
```

```
 * This is the function that checks for the correct set of characters,
 * (ie. letters a-z both uppercase and lowercase,numbers, and spaces, periods
 * or the pound symbol) in the address field and is used on the address portion
 * of the record.
 *
 * @param[in]        address - This typecasts the address portion of the
 *                             record from a character array to a string
 *                             allowing the usage of the string class member
 *                             functions.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 *****************************************************************************/
void numCheck(string address, string &countFalse)
{
    int idx;

    idx = address.find_first_not_of(validAddress);
    if (idx != address.npos)
    {
        countFalse[1] = 1;
        return;
    }
    else
        countFalse[1] = 0;
}


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the function that checks if the 1st character of the state is
 * capitalized and if the state is a valid state.  If the 1st character is not
 * capitalized or if the, the state is not valid variable countFalse is
 * incremented to a 1 accordingly.
 *
 * @param[in]          state - This typecasts the state portion of the record
 *                             from a character array to a string allowing the
 *                             use of the string class member functions,
 *                             specifically the find_first_not_of function.
 *
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 *****************************************************************************/
void state(string state, string &countFalse)
{
    int size = 0;
    int i;
    int idx;

    size = state.size();
    if (size > 2)
        state.erase(2);
    if (state[0] >= 65 && state[0] <= 90)
        countFalse[4] = 0;
    else
        countFalse[4] = 1;
    for (i = 0; i < 2; i++)
    {
        state[i] = toupper(state.at(i));
    }
    for (i = 0; i < 50; i++)
    {
        idx = state.compare(validStates[i]);
        if (idx != 0)
        {
            countFalse[3] = 1;
        }
        else
        {
            countFalse[3] = 0;
            return;
        }
    }
}


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the function that checks if the four digit and five digit portions
 * of the zipcode are valid.  If the 5 digit portion or the four digit portion
 * of the zip code is not valid variable countFalse is incremented to a 1
 * accordingly
 * @param[in]            zip - This is the zipcode portion of the record that
 *                             is to be extracted.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 *****************************************************************************/
void zipCheck(unsigned int zip, string &countFalse)
{
    unsigned int fourDigit;
    unsigned int fiveDigit;
    unsigned int temp;

    fourDigit = zip & DIG4MASK;
    temp = zip >> 14;
    fiveDigit = temp & DIG5MASK;

    if (fiveDigit >= 10000 && fiveDigit <= 99999)
        countFalse[5] = 0;
    else
    {
        countFalse[5] = 1;
    }
    if (fourDigit >= 1000 && fourDigit <= 9999)
        countFalse[6] = 0;
    else
    {
        countFalse[6] = 1;
    }
}


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the function that extracts the date in its portions of month, day,
 * and year.  The function then checks to find if the month, day, and year are
 * valid.  If any part of the date is not valid, the  variable countFalse is
 * incremented to a 1 accordingly.
 *
 * @param[in]            date - This is the date portion of the record that
 *                              is to be extracted into its seperate parts.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 * @param[in,out]          i - This variable keeps count of where in the string
 *                             to change to a 1 or 0 depending on whether there
 *                             is an error or not.
 * @param[in,out]     format - This keeps track of the date which is put in as
 *                             year/month/day.  This allows the program to
 *                             compare the dates in another function.
```

```
 ***************************************************************************/
void extractDate(unsigned int date, string &countFalse, int &i, string &format)
{
    int day;
    int month;
    int year;
    unsigned int shift1;
    unsigned int shift2;

    day = date & dayMask;
    shift1 = date >> 6;
    month = shift1 & monthMask;
    shift2 = shift1 >> 6;
    year = shift2 & yearMask;

    format.push_back(year);
    format.push_back('/');
    format.push_back(month);
    format.push_back('/');
    format.push_back(day);

    monthCheck(month, countFalse, i);
    i++;
    dayCheck(day, month, countFalse, i);
    i++;
    yearCheck(year, countFalse, i);
    i++;
}


/***************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This function takes the month that is sent in and checks
 * to make sure that it is a valid month.  If the month is not valid,
 * countFalse is incremented to reflect the error.  Otherwise it is set to 0 if
 * it is valid.
 *
 * @param[in]          month - This is the month that has been extracted from
 *                             the date field of the Record structure.  It is
 *                             compared against the valid months to see if it
 *                             is true or false.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 * @param[in]              i - This variable keeps count of where in the string
 *                             to change to a 1 or 0 depending on whether there
 *                             is an error or not.
 ***************************************************************************/
void monthCheck(int month, string &countFalse, int i)
{
    if (month > 0 && month <= 12)
        countFalse[i] = 0;
    else
        countFalse[i] = 1;
}


/***************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This function takes the month that is sent in and checks to make sure that
 * it is a valid month.  If the month is not valid, countFalse is incremented
 * to reflect the error.  Otherwise it is set to 0 if it is valid.
 *
 * @param[in]          month - This is the month that has been extracted from
 *                             the date field of the Record structure.  It is
```



```
 *                             used to see if the day portion of the date field
 *                             is valid.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 * @param[in]              i - This variable keeps count of where in the string
 *                             to change to a 1 or 0 depending on whether there
 *                             is an error or not.
 * @param[in]            day - This is the day that has been extracted from the
 *                             date portion of the record structure.  It is
 *                             compared against the valid days of the
 *                             particular month that has been extracted from
 *                             the date field.  If it is not valid, the
 *                             countFalse variable is incremented accordingly.
 ***************************************************************************/
void dayCheck(int day, int month, string &countFalse, int i)
{
    int maxDay;

    maxDay = validDay[month];
    if (day > 0 && day <= maxDay)
        countFalse[i] = 0;
    else
        countFalse[i] = 1;
}


/***************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This function takes the month that is sent in and checks to make  sure that
 * it is a valid month.  If the month is not valid, countFalse is incremented
 * to reflect the error.  Otherwise it is set to 0 if it is valid.
 *
 * @param[in]           year - This is the year that has been extracted from
 *                             the date field of the Record structure.  It is
 *                             used to see if the year portion of the date
 *                             field is valid.  A valid year is between 1900
 *                             and 2015.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 * @param[in]              i - This variable keeps count of where in the string
 *                             to change to a 1 or 0 depending on whether there
 *                             is an error or not.
 ***************************************************************************/
void yearCheck(int year, string &countFalse, int i)
{
    if (year >= 1900 && year <= 2015)
    {
        countFalse[i] = 0;
    }
    else
    {
        countFalse[i] = 1;
    }
}


/***************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 *
 *
 * @param[in]         license - This is the license date part of the structure
 *                             and is in the format of year/month/day and is
 *                             checked to see if it is before the expire date.
```

```
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 * @param[in]         expire - This is the expiration date part of the
 *                             structure and is in the format of year/month/day
 *                             and is checked to see if it is after the license
 *                             date.
 ******************************************************************************/
void checkIfAfter(string &countFalse, string license, string expire)
{
    int expireAfter;
    expireAfter = license.compare(expire);
    if (expireAfter >= 0)
        countFalse[16] = 1;
    else
        countFalse[16] = 0;
}


/******************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the function that checks for the correct set of characters in the
 * Radio Class field and is used on the Radio Class portion of the record to
 * check that it is a valid class.  If the class is not valid, countFalse is
 * incremented accordingly.
 *
 * @param              radio - This is the radio class portion of the record
 *                             that is typecasted from a character array to a
 *                             string when it is passed in.  It is compared
 *                             against all of the valid classes and countFalse
 *                             is incremented accordingly if the class is not
 *                             valid.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 ******************************************************************************/
void classCheck(string radio, string &countFalse)
{
    radio.erase(1);
    int idx;

    idx = radio.find_first_not_of(validClass);
    if (idx != radio.npos)
    {
        countFalse[17] = 1;
    }
    else
        countFalse[17] = 0;
}


/******************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the function that checks to see if the callSign portion of the
 * record is valid.  To be valid, the sign must start with a K, W, or N, the
 * second, fourth, and fifth characters must be any letter A-Z and the third
 * character must be any single digit number 0-9.  All letters must be
 * capitalized.  If any portion of the call sign is invalid, countFalse is
 * incremented accordingly.
 *
 * @param[in]          sign - this is the callSign portion of the record
 *                            that is read from file.  It is typecasted to a
 *                            string as it is passed into the function and
 *                            it is then checked to make sure that it meets
 *                            all the requirements given.  If any part of the
```



```
 *                            callSign is invalid, the valid countFalse is
 *                            incremented accordingly.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 ******************************************************************************/
void sign(string sign, string &countFalse)
{
    sign.erase(5);
    countFalse.resize(24);
    int idx;
    int upper;
    idx = sign.find_first_not_of(validChar);
    if (idx == sign.npos || idx == 1)
        countFalse[18] = 0;
    else
        countFalse[18] = 1;
    if (sign[1] >= 65 && sign[1] <= 90)
        countFalse[19] = 0;
    else
        countFalse[19] = 1;
    if (sign[2] >= 48 && sign[2] <= 57)
        countFalse[20] = 0;
    else
        countFalse[20] = 1;
    if (sign[3] >= 65 && sign[3] <= 90)
        countFalse[21] = 0;
    else
        countFalse[21] = 1;
    if (sign[4] >= 65 && sign[4] <= 90)
        countFalse[22] = 0;
    else
        countFalse[22] = 1;
    upper = isupper(sign[0]);
    if (upper == 0 || countFalse[19] == 1 || countFalse[21] == 1 || countFalse[2
2] == 1)
    {
        countFalse[19] = 0;
        countFalse[22] = 0;
        countFalse[21] = 1;
    }
    countFalse.resize(21);
}


/******************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This function processes the countFalse string looking to see if there are
 * any errors in the string, and if there are, the outputError function is
 * called to output the errors.  If there are no errors, the outputGood
 * function which will output the good records that do not have any errors.
 *
 * @param[in, out]       good - Output File Stream that has been opened.  Used
 *                              to output good records to a binary file.
 * @param[in, out]        bad - Output File Stream that has been opened.  Used
 *                              to output bad records to a text file.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 * @param[in]          record - This is passed in to allow access to the records
 *                              that were read into the structure in main.
 *                              Program breaks if this is declared as a local
 *                              variable.
 ******************************************************************************/
void errorCheck(string &countFalse, ofstream &good, ofstream &bad, Record record
)
{
```

```
    int idx;
    idx = countFalse.find_first_of(1);
    if (idx != countFalse.npos)
    {
        outputError(countFalse, bad, record);
        bad << endl;
    }
    else
        outputGood(good, record);
}


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This function outputs any errors to a text file.
 *
 * @param[in, out]       bad - Output File Stream that has been opened.  Used
 *                             to output bad records to a text file.
 * @param[in,out] countFalse - This is the string that keeps track of all
 *                             errors found in the database file.  Each portion
 *                             is set to a 1 if there are any errors.
 * @param[in]         record - This is passed in to allow access to the records
 *                             that were read into the structure in main.
 *                             Program breaks if this is declared as a local
 *                             variable.
 *****************************************************************************/
void outputError(string &countFalse, ofstream & bad, Record record)
{
    bad << record.name << endl;
    if (countFalse[0] == 1)
        bad << "   Invalid character in the name field" << endl;
    if (countFalse[1] == 1)
        bad << "   Invalid character in the address field" << endl;
    if (countFalse[2] == 1)
        bad << "   Invalid character in the city field" << endl;
    if (countFalse[3] == 1)
        bad << "   Invalid State Code" << endl;
    if (countFalse[4] == 1)
        bad << "   First character of State Code not capitalized" << endl;
    if (countFalse[5] == 1)
        bad << "   Invalid 5 digit zip code" << endl;
    if (countFalse[6] == 1)
        bad << "   Invalid 4 digit zip code" << endl;
    if (countFalse[7] == 1)
        bad << "   Invalid month in Birthday" << endl;
    if (countFalse[8] == 1)
        bad << "   Invalid day in Birthday" << endl;
    if (countFalse[9] == 1)
        bad << "   Invalid year in Birthday" << endl;
    if (countFalse[10] == 1)
        bad << "   Invalid month in License" << endl;
    if (countFalse[11] == 1)
        bad << "   Invalid day in License" << endl;
    if (countFalse[12] == 1)
        bad << "   Invalid year in License" << endl;
    if (countFalse[13] == 1)
        bad << "   Invalid month in Expiration" << endl;
    if (countFalse[14] == 1)
        bad << "   Invalid day in Expiration" << endl;
    if (countFalse[15] == 1)
        bad << "   Invalid year in Expiration" << endl;
    if (countFalse[16] == 1)
        bad << "   Expiration Date is not after the License Date" << endl;
    if (countFalse[17] == 1)
        bad << "   Radio Class code is invalid" << endl;
    if (countFalse[18] == 1)
        bad << "   Invalid first character in call sign" << endl;
```



```
    if (countFalse[19] == 1)
        bad << "   Invalid second, fourth, or fifth character in call sign"
            << endl;
    if (countFalse[20] == 1)
        bad << "   Invalid digit in call sign" << endl;
    if (countFalse[21] == 1)
        bad << "   Call Sign not uppercase" << endl;
}


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This function is called by outputError function to output all good records.
 *
 *
 * @param[in]         record - This is passed in to allow access to the records
 *                             that were read into the structure in main.
 *                             Program breaks if this is declared as a local
 *                             variable.
 * @param[in, out]      good - Output File Stream that has been opened.  Used
 *                             to output good records to a binary file.
 *****************************************************************************/
void outputGood(ofstream &good, Record record)
{
    good.write((char*)&record, sizeof(Record));
}


/*****************************************************************************//**
 * @author Brandon Amundson
 *
 * @par Description:
 * This is the function that opens the files for writing the good records to a
 * specified binary file and the bad records to a specified text file.  All
 * records are read in from a specified binary database file.  If this function
 * is unable to open any file, it exits with a code one.
 *
 *
 * @param[in, out]       fin - Input File Stream to be closed.
 *                             Used to read in contents from binary file to be
 *                             checked for accuracy.
 * @param[in, out]      good - Output File Stream to be closed.  Used to output
 *                             good records to a binary file.
 * @param[in, out]       bad - Output File Stream to be closed.  Used to output
 *                             bad records to a text file.
 *****************************************************************************/
void closeFiles(ifstream &fin, ofstream &good, ofstream &bad)
{
    good.close();
    bad.close();
    fin.close();
}
```