

Project Report  
CS 4350.251  
Spring 2017  
Group 12  
Project 1  
Contact Manager

Mark McDermott  
Jason Flinn  
Carlo Rodriguez  
Benjamin Winston

## **Section I**

Our group split up the different tasks of the database management code and the accompanying report. For the code, Mark built the menu; Jason built the add record function; Carlo built the find record function; and Benjamin built the update, and remove record functions, as well as the display function. For this report, Mark wrote Section I, II and III; Benjamin contributed function descriptions and update screen shot to Section III; and Carlo wrote section IV. Our group communicated with the Slack chat app and used git and github to keep the code under version control.

## **Section II**

Our database is a flat .txt file with fields separated by colons and each entry has its own row. We have a name field which has first and last names. We have an address field with the street address. We have a phone field with a user's seven digit phone number (no dashes or other characters, just seven numbers). We have an email field with the user's email.

Multiple entries with the same name were created so that we can test for the possibility of multiple records being displayed within the Find() function. The two "Joe Smith" entries were given different e-mail addresses so that we can differentiate the two records.

```
Name:Address:Phone:Email
Joe Smith:123 Appletree Rd:1234567:joesmith@gmail.com
Joe Smith:123 Appletree Rd:1234567:joesmiththesecond@yahoo.com
Leslie James:8123 Fancy Rd:8763467:lesliejames@mail.com
Ralph Randall:863 Fair Pl:8963432:ralphrandall@gmail.com
```

## **Section III**

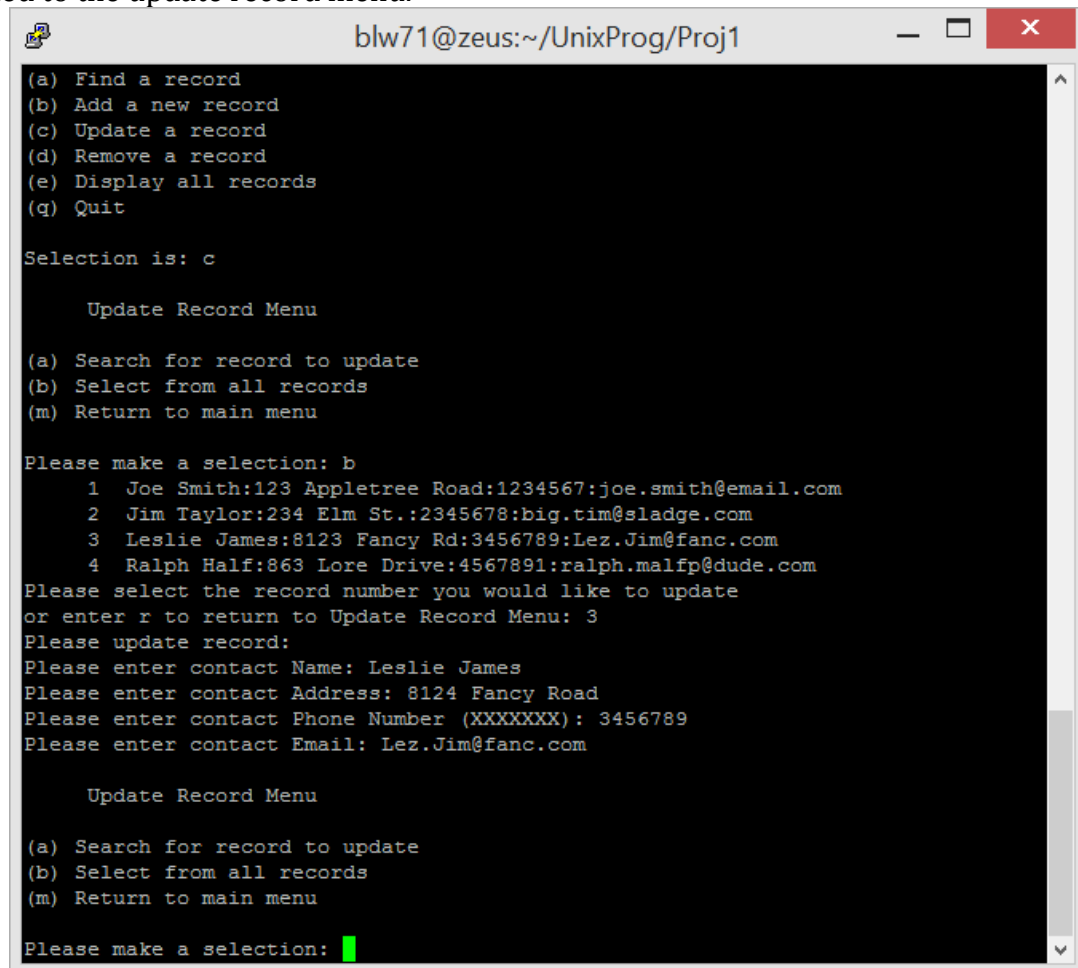
We designed our code to be object oriented, so each task is broken out into its own function. Our menu calls a task function, depending on what input is entered by the user. Our find function takes user input for name, address, phone or email and then finds the record with that info or returns that the record is not found. To search the through database, for our find() function, we use grep.

The find function uses a series of if statements to do some error checking. First it checks that the database exists and if it doesn't, it prints an error and returns to the main function. Then it checks if it has not been passed arguments with if [ "\$#" -eq 0 ] and if there are no arguments it does read -p asking for the name, address, phone and email. If there is an argument, it sets it equal to the variable reply with reply="\$1". Then it greps reply. If grep finds nothing or if reply is empty or is just a colon, it notifies the user and returns 1. Then use iconv -l and grep to search for the value of reply in the database variable. Iconv makes the database searchable by grep. We then pipe this to head to print the first ten lines and to tr to replace colons with spaces.

For our add user functionality we ask the user a series of questions to prompt user input for the new name, address, phone and email. Each prompt is a read statement using a string and a variable as arguments. The string asks the user to enter the input. The variable is where the input is stored. Each input is then

validated to check if the input entered matches the type of data needed for that field. We validate the variable with a while loop. The name, address, phone and email use while [ \${#VARIABLE} -lt 1 ] to check if the user just hit enter instead of entering the needed input. Inside the while loop, we use read -p with a string argument to notify the user that no input was submitted and to try again. This while will continue to execute as long as the user keeps hitting return and entering no input. The error checking for phone is while [ \${#CONPHONE} -gt 7 ], which checks if the user entered more than the seven needed digits. If they did enter too many digits, then the while loop executes and uses read -p with a string argument that notifies the user that too many digits were entered. After we error check each input we then use printf with the stream extraction operator to put each new input, concatenated with a colon, into the database before moving on to the next. Here is an example, for putting the address input in the database: printf "\${CONADD}:" >> \$database

For the update record functionality, the user has the choice to search for a record to update, or to see a list of all records and in both cases, make a selection based on record number. Once the user has selected the desired record, they are prompted to provide new record information. After update information has been entered, it replaces the existing information in the database, and the user is returned to the update record menu.



```
blw71@zeus:~/UnixProg/Proj1
(a) Find a record
(b) Add a new record
(c) Update a record
(d) Remove a record
(e) Display all records
(q) Quit

Selection is: c

    Update Record Menu

(a) Search for record to update
(b) Select from all records
(m) Return to main menu

Please make a selection: b
  1  Joe Smith:123 Appletree Road:1234567:joe.smith@email.com
  2  Jim Taylor:234 Elm St.:2345678:big.tim@sladge.com
  3  Leslie James:8123 Fancy Rd:3456789:Lez.Jim@fanc.com
  4  Ralph Half:863 Lore Drive:4567891:ralph.malfp@dude.com

Please select the record number you would like to update
or enter r to return to Update Record Menu: 3
Please update record:
Please enter contact Name: Leslie James
Please enter contact Address: 8124 Fancy Road
Please enter contact Phone Number (XXXXXXX): 3456789
Please enter contact Email: Lez.Jim@fanc.com

    Update Record Menu

(a) Search for record to update
(b) Select from all records
(m) Return to main menu

Please make a selection: █
```

Walking through the update code in more detail, we first check if the database variable is there with `if ! [ -f $database ]`. If the database is not there, we notify the user and return to the main menu. Then we set some variables with strings for user notifications in the update menu:

```
menuTitle="Update Record Menu"
```

```
optionA="Search for record to update"
```

```
optionB="Select from all records"
```

We then have a while statement which checks if the CHOICE variable is not "m", which would mean the user wants to return to the main menu. Inside the while loop, a menu is printed using `printf` and the string variables above. The menu lets the user choose to search for a record to update or to select one record from a list of all the records. After the `printf` statements we use `read CHOICE` to read the users input for the CHOICE variable.

Next we have a case statement in `Update()`, handling the heavy lifting for choosing what field to update and then updating it. If the user entered "a", saying that they want to search for a field to update, we use `read -p` with a string argument and the CHOICE variable argument. This asks the user to enter a search string. We then use a while statement to error check if no string was entered. The while is `while [ -z $CHOICE ]` which uses `-z` to test if the string is null. If the string is empty, the while loop is entered, using `read -p` with a string and the CHOICE variable as arguments to keep asking for new input until the input is not blank. Then we run `grep -n $CHOICE $database | tr ':' ' '` to find the string the user wants to search. `-n` provides the line numbers. We use the pipe to pipe the found `grep` matches to `tr` which we are using to replace the colons with spaces for an easier reading experience. We then use an if statement with `[ $? -eq 0 ]` to check `grep` found results. `grep` returns zero if it finds matches. If matches are found, then we `printf` a message asking which record of the results the user would like to update. Then we read in the CHOICE variable. If CHOICE is "r" then we return to the main menu. If the choice isn't "r" then we take everything from before the CHOICE line number and put it in a variable called `$dbTemp` with `head -n `expr $CHOICE - 1` $database > $dbTemp`. `head -n` takes everything from the beginning of a file, or in this case a variable, to a specified line number. We pass `head` the argument ``expr $CHOICE - 1`` which evaluates `$CHOICE - 1` and then we also pass the argument `$database > $dbTemp` to pass the selection into `$dbTemp`. Then we take everything after choice and append it to `$dbTemp` with `tail -n +`expr $CHOICE + 1` $database >> $dbTemp`. `Tail` operates like `head`, but instead of selecting from the beginning to a line number, it selects from a line number to the end of a file. We then `mv $dbTemp $database` to replace the current database with the new temp variable. Then we run the `Add()` function to insert the new record, since we have removed the row the user wanted to update.

The other `Update()` case is "b", where the user wants to select from all the records to find one to update instead of searching for the one they want to update. In this case, we print out the whole database with `cat -n $database | tr ':' ' '`. `Cat` prints out `$database` with line numbers because of the `-n` option and is piped to `tr` which removes the colons since we use the `:` and `' '` arguments. Then we read the CHOICE variable. With an if statement, we say, if choice is "r", meaning they want to

return to the main menu, then printf a notification and do nothing, which will mean they return to the main menu. Otherwise our else is triggered and while loop runs which says, while [ -z \$CHOICE ], or while CHOICE is empty, read -p a notification that no record number was entered and then read CHOICE again. Once we have the CHOICE entered, we run mv \$dbTemp \$database and then Add() like in the “a” case of Update();

The remove function first checks if the \$database variable exists with if ! [ -f \$database ]. If it isn't present, then we printf a notification that it isn't present and return a value of 1, signifying an error. We then declare three string variables:

```
menuTitle="Remove Record Menu"
```

```
optionA="Search for record to remove"
```

```
optionB="Select from all records"
```

We then print a menu asking whether the user would like to search for a record to remove or to select from all records for a record to remove. We then get user input with read CHOICE and run a case statement, which is similar to the case in Update().

If the user enters “a”, saying that they want to search for a field to remove, we use read -p with a string argument and the CHOICE variable argument. This asks the user to enter a search string. We then use a while statement to error check if no string was entered. The while is while [ -z \$CHOICE ] which uses -z to test if the string is null. If the string is empty, the while loop is entered, using read -p with a string and the CHOICE variable as arguments to keep asking for new input until the input is not blank. Then we run grep -n \$CHOICE \$database | tr ':' ' ' to find the string the user wants to search. -n provides the line numbers. We use the pipe to pipe the found grep matches to tr which we are using to replace the colons with spaces for an easier reading experience. We then use an if statement with [ \$? -eq 0 ] to check grep found results. Grep returns zero if it finds matches. If matches are found, then we printf a message asking which record of the results the user would like to remove. Then we read in the CHOICE variable. If CHOICE is “r” then we return to the main menu. If the choice isn't “r” then we take everything from before the CHOICE line number and put it in a variable called \$dbTemp with head -n `expr \$CHOICE - 1` \$database > \$dbTemp. Head -n takes everything from the beginning of a file, or in this case a variable, to a specified line number. We pass head the argument `expr \$CHOICE - 1` which evaluates \$CHOICE - 1 and then we also pass the argument \$database > \$dbTemp to pass the selection into \$dbTemp. Then we take everything after choice and append it to \$dbTemp with tail -n +`expr \$CHOICE + 1` \$database >> \$dbTemp. Tail operates like head, but instead of selecting from the beginning to a line number, it selects from a line number to the end of a file. We then mv \$dbTemp \$database to replace the current database with the new temp variable.

The other Remove() case is “b”, where the user wants to select from all the records to find one to remove instead of searching for the one they want to remove. In this case, we print out the whole database with cat -n \$database | tr ':' ' '. Cat prints out \$database with line numbers because of the -n option and is piped to tr which removes the colons since we use the ':' and ' ' arguments. Then we read the CHOICE variable. With an if statement, we say, if choice is “r”, meaning they want to return to the main menu, then printf a notification and do nothing, which will mean

they return to the main menu. Otherwise our else is triggered and while loop runs which says, while [ -z \$CHOICE ], or while CHOICE is empty, read -p a notification that no record number was entered and then read CHOICE again. Once we have the CHOICE entered, we run mv \$dbTemp \$database.

The display function is the simplest of our functions. First it checks if the database variable is there with if ! [ -f \$database ]. If the database is not there, we notify the user and return to the main menu. Then we get the number of records with numRecords=`cat \$database | wc -l` which gets the line numbers by piping the database to wc with the -l option. Since the first line of the database is a header line we remove that with numRecords=`expr \$numRecords - 1`, which sets numRecords equal to numRecords - 1. Then we show all records with cat -n \$database | tr ':' '' which prints the database variable with line numbers with cat -n \$database and pipes it to tr to remove the colons.

Our quit function uses printf to thank the user for using the program and then simply does exit 0. Returning zero says the program was successful.

## **Section IV**

```
Find() { #Function name to find a record within the database.
  if ! [ -f $database ] #checks for the existence of a database.
  then #If no database, enter if condition.
    printf "No Database present. Please populate the database before selecting
      this option.\n" #Prompt user to populate database.
    printf "Returning to main menu.\n" #Move user back to Main Menu.
    return 1 #Return out of the function; moves to Main Menu.
  fi

  if [ "$#" -eq 0 ]; then #If no arguments passed to Find, enter "if".
    read -p "Please enter a name, address, phone number, or e-mail: " reply
    #Prompt user to enter a field, and puts field into "reply".
  Else #If argument passed, enter else.
    reply = "$1" #Insert "$1", argument passed, into "reply".
  fi

  echo #Newline

  if ! grep -q "$reply" $database; then #enter 'if', if nothing is found.
    echo "Record not found." #Prompt user that record is not found.
    echo #Newline
    return 1 #Exit function and return to menu.
  fi

  if [ "$reply" = ":" ]; then #If user entered a ":", do not allow.
    echo "Record not found." #Prompt user that record is not found.
    echo #Newline
    return 1 #Exit function and return to menu.
  fi

  if [ -z "$reply" ]; then #If user enters nothing, do not allow.
    echo "Record not found." #Prompt user that record is not found.
    echo #Newline
    return 1 #Exit function and return to menu.
  fi
```

```

iconv -l | grep "$reply" $database | head | tr ':' ' '
#This line uses iconv to convert encoding of characters in the
#input file and writes to stdout, uses "grep" to get the records
#that match the reply, uses the "head" command to search the
#entire file, and uses "tr" to replace any ":" characters with a
#blank space.

echo #Newline
}

```

### 1) Exactly matching entry for a query:

```

Welcome to my contact database, please select in the following menu:

(a) Find a record
(b) Add a new record
(c) Update a record
(d) Remove a record
(e) Quit

Selection is: a
Please enter a name, address, phone number, or e-mail: joesmith@gmail.com

Joe Smith 123 Appletree Rd 1234567 joesmith@gmail.com

```

Our group decided that the best way to search for an exact entry within the Find() function was to search by the users e-mail. Currently, our program allows multiple entries with the same name, phone number, and address, but prohibits multiple entries with the same e-mail. Entering an e-mail will always give the user one record from the database.

### 2) Multiple matching records for a query:

```

Welcome to my contact database, please select in the following menu:

(a) Find a record
(b) Add a new record
(c) Update a record
(d) Remove a record
(e) Quit

Selection is: a
Please enter a name, address, phone number, or e-mail: Joe

Joe Smith 123 Appletree Rd 1234567 joesmith@gmail.com
Joe Smith 123 Appletree Rd 1234567 joesmiththesecond@yahoo.com

```

Because the database has two entries that have the same name, searching “Joe” will find both entries and display them to the user.

### 3) Finding nothing for a query:

```
(a) Find a record
(b) Add a new record
(c) Update a record
(d) Remove a record
(e) Quit

Selection is: a
Please enter a name, address, phone number, or e-mail: :

Record not found.
```

Since our database is designed to separate fields with colons, we needed to remove the ability to allow the user to enter a colon, which would naturally find every record within the database. Our Find() function removes this right from the user.

```
(a) Find a record
(b) Add a new record
(c) Update a record
(d) Remove a record
(e) Quit

Selection is: a
Please enter a name, address, phone number, or e-mail: Michael

Record not found.
```

Since there is no record with the name “Michael”, the function displays that the record is not found.

```
(a) Find a record
(b) Add a new record
(c) Update a record
(d) Remove a record
(e) Quit

Selection is: a
Please enter a name, address, phone number, or e-mail:

Record not found.
```



Removing the ability to search for nothing was removed, because searching for nothing displays all records within the database.