# ASSIGNMENT TWO: PHONE BOOK

ADVANCED PROGRAMMING TECHNIQUES – SEMESTER 1, 2017

## SUMMARY

In this assignment, you will use your C programming skills to create a phone book. The phone book loads its entries from external CSV text files; manipulates the entries with certain commands (i.e. add, remove, sort, find features); and writes the output back to CSV text files.

The following sections contain the details of this assignment. You are expected to understand every requirement explained in this document and implement all of them accordingly.

This assignment has a strong emphasis on doubly linked lists. You should get yourself familiar with this concept as soon as possible.

Similar to the previous assignment, a start-up code is provided to guide your implementation. **You must use all the functions provided in the start-up in your assignment.**

## SUBMISSION

**Release date:** End of week 7 – Friday, 21 April 2017

**Total mark:** 30% of the final mark + 10% bonus marks

**Assignment demo:** During tutorial / lab sessions of week 10 (8-12 May 2017)

**Final submission:** End of week 12 – Saturday, 27 May 2017 – **10:00 PM**

## PLAGIARISM NOTICE

Plagiarism is a very serious offence. The minimum first time penalty for plagiarised assignments is zero marks for that assignment.

Please keep in mind that RMIT University uses plagiarism detection software to detect plagiarism and that all assignments will be tested using this software.

More information about plagiarism can be found here: http://www1.rmit.edu.au/academicintegrity/.

# DESCRIPTION OF THE PROGRAM

Throughout this assignment specification, you will see the names "address book" and "phone book" are used interchangeably. These are both referring to this project.

In this section, the functional and non-functional requirements of the program are described. Marks allocated for each requirement is available in a separate section.

**Colours and input/output messages:** In the sample output shown above, the text that is written in `yellow colour` is the user's input. When the program is expecting an input from the user, it should always show the message `Enter your command:` and await input. As you will see in the following sections, where a command generates output messages, the messages always start with a `>` character. This convention is used in the rest of this document. Some requirements contain an Implementation Guide section that provides extra information about how you can solve that requirement and where the relevant and additional information is available.

# CORE REQUIREMENTS

## REQ1: PRINT STUDENT INFORMATION WHEN THE PROGRAM STARTS

After compiling the program and running its executable, the following output should be displayed.

```
./addressbook

------------------------------------------------------------------
Student name: Firstname Lastname
Student number: 1234567
Advanced Programming Techniques, Assignment Two, Semester 1, 2017
------------------------------------------------------------------

Enter your command:
```

The Firstname and Lastname keywords are placeholders for your first name and surname. The 1234567 is the placeholder for your student number.

## REQ2: LOAD A TELEPHONE BOOK FILE INTO MEMORY

There are three sample files provided with the start-up code. They are named `sml.txt`, `med.txt` and `lrg.txt`. They are simple CSV files containing the entries of sample address books.

On the top of each file there is a section for comments. When loading the file into memory, these commented lines should be ignored. That is, your program should not load these sections of the file.

The following is the first 11 lines of the `sml.txt` file shown as an example.

```
# This is the small address book file.
# Each line of this file contains an address book entry.
# The order of the data is: ID, Name, Telephone
```

```
#
# Note: Lines starting with # are comments and should be ignored.
#
100,Alice,0411112221
101,Bob,0411112222,0422221111
102,Ali,0411112223,0422221112,0422221113
103,Reza,0411112224
104,Ryan,0411112225,0422221114,0422221115,0422221116
...
```

The data stored in the address book files contains three columns:

1) ID: A unique identifier for the current entry
2) Name: Name of a person
3) Telephones: a set of 10-digit telephone numbers (can be empty)

The user should be able to use the **load** command to load one of these files (or any other file with the same format and structure) into memory. The name of the file to load will be specified right after the **load** command as shown in the following example.

```
Enter your command: load sml.txt
> Opening the file sml.txt.
> Loading the file ...
> 21 phone book entries have been loaded from the file.
> Closing the file.
```

When a file is being loaded, there are a number of output messages that should be displayed in the console according to the above example. Note that the number of items read from the file must be shown.

The number of entries in the file is not fixed and can vary. Therefore, you cannot make any assumptions about the number of items which are stored in the file. Similarly, the number of telephone numbers associated with each entry can vary (zero or more). Your program must be able to read any file with a valid format regardless of the number of items in that file.

If a non-existing file is specified, a correct error message should be shown, such as the one displayed in the following example.

```
Enter your command: load non-existing-file.txt
> Opening the file non-existing-file.txt.
> Error: Unable to find the specified file.
```

If the file exists but contains corrupt content (e.g. invalid format), a correct error message should be shown, such as the one displayed in the following example.

```
Enter your command: load existing-but-corrupt-file.txt
> Opening the file existing-but-corrupt-file.txt.
> Loading the file ...
> Error: The specified file is in the wrong format and cannot be loaded.
```

If the linked-list currently contains nodes (i.e. another file has already been loaded), **you should first release the entries of that list** and then attempt to read another file. Loading one file after another should result in the first one be unloaded and the second one be loaded into the list.

In the event that a file is corrupt, you should not add any nodes to the doubly linked list. If you have already added any nodes to the list (e.g. in the event that the first few lines of data are valid

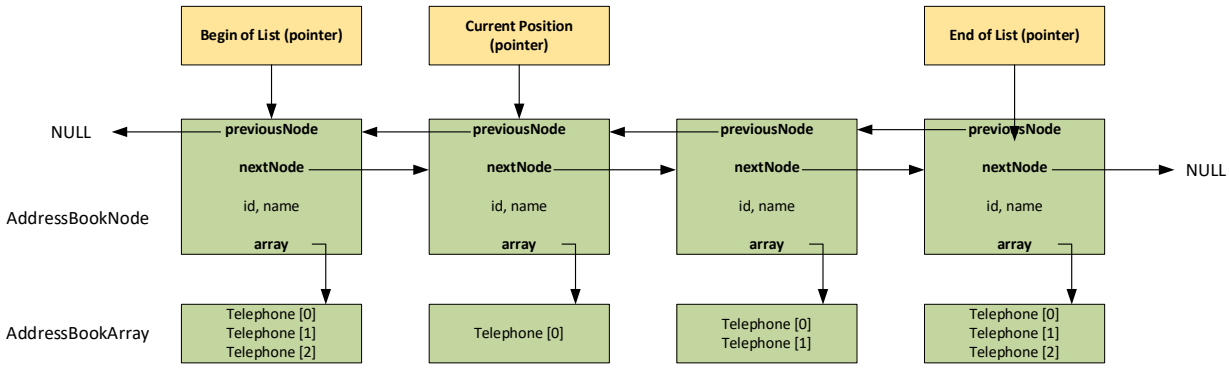
before reaching a corrupt part of the file), you should delete those elements after failing to read the rest of the file.

**Implementation Guide**

In order to load the content of the file into memory, you will need to create a Doubly Linked List of nodes. Each node in the list stores ID, Name and Telephone number of an entry in the file. Each node also stores a pointer to its previous and next element in the list. The data structure you need to use is as follows.

```
/* Taken from addressbook_list.h */
typedef struct addressBookNode
{
    int id;
    char name[NAME_LENGTH];
    AddressBookArray * array;
    /* The previous node in the linked-list */
    struct addressBookNode * previousNode;
    /* The next node in the linked-list */
    struct addressBookNode * nextNode;
} AddressBookNode;
```

The memory structure of a four-element doubly linked-list resembles the following figure. You should maintain a pointer to the first element of the list (head), another pointer to the last element of the list (tail), and a third pointer to the current location in the list.




In each entry in the list, you need to have a pointer to the list of telephone numbers too (i.e. to an instance of `AddressBookArray`). The structure of `AddressBookArray` is as follows, where the `size` indicates how many items is currently stored in the array and `telephones` is a double pointer used to interact with (i.e. create, delete, resize etc) the array of strings containing the telephone numbers.

```
/* Taken from addressbook_array.h */
typedef struct addressBookArray
{
    int size;
    char ** telephones;
} AddressBookArray;
```

Each node in the doubly linked list is a new item created in memory from the `AddressBookNode` data type. Each node's `nextNode` points to the next element in the

list, and each node's `previousNode` point to the previous element in the list. The `previousNode` of the first item and the `nextNode` of the last item point to NULL.

In order to store the pointers to head, tail and current position in the list, the following data structure should be used. It also includes a variable for the current size of the list.

```
/* Taken from addressbook_list.h */
typedef struct addressBookList
{
    int size;
    AddressBookNode * head;
    AddressBookNode * tail;
    AddressBookNode * current;
} AddressBookList;
```

When a file is being loaded, your program should read each line of the file and (if read successfully) then create a new instance of `AddressBookNode` and store ID, Name and Telephone members in the newly created structure. Thereafter, you should set the pointers of that struct (`previousNode` and `nextNode`) correctly: you add the newly added entries (notes) to the end of the list, so you must (1) set the `previousNode` of the newly created node to be equal to the tail pointer, (2) set the `nextNode` of the last element of the list to point to the newly created node, and (3) update the tail pointer to point to the newly created node (instead of the now one-before-the-last node).

When the address book file is completely loaded, your linked-list should contain the same number of nodes as the number of data lines in the text file. That is, if 21 lines of data are stored in the file, you should have 21 nodes in your list.

**It is recommended that you complete the *load* functionality in two stages.** Firstly, implement everything necessary to load the list without writing any code to load the telephone numbers. This will help you build the core logic of loading and adding new items to the list. Afterwards, extend your code to accommodate for the list of telephone numbers.

## REQ3: Load a Telephone Book File into Memory Through Command-Line

The **load** command that was discussed in the previous requirement (REQ2) should be available to the user through a command-line argument.

The user can specify the name of the address book file which should be loaded into memory as the argument to executable of the program. In this case, the program will start and display the student information, and then attempts to load the specified file. The following is an example.

```
./addressbook sml.txt

------------------------------------------------------------------
Student name: Firstname Middlename Lastname
Student number: 1234567
Advanced Programming Techniques, Assignment Two, Semester 1, 2017
------------------------------------------------------------------

> File name specified through the command-line argument.
> Opening the file sml.txt.
> Loading the file ...
> 21 phone book entries have been loaded from the file.
```

```
> Closing the file.

Enter your command:
```

If the specified file does not exist, a correct error message should be displayed. The following example shows this scenario.

```
./addressbook non-existing-file.txt

------------------------------------------------------------------
Student name: Firstname Middlename Lastname
Student number: 1234567
Advanced Programming Techniques, Assignment Two, Semester 1, 2017
------------------------------------------------------------------

> File name specified through the command-line argument.
> Opening the file non-existing-file.txt.
> Error: Unable to find the specified file.

Enter your command:
```

Similar to REQ2, in the event that the specified file can be opened but contains corrupt data, the correct error message should be displayed to the user.

**Implementation Guide**

It is important to pay attention to modular abstraction when implementing requirements REQ2 and REQ3. Note that these two requirements are doing the same things behind the scene and, therefore, the code for loading the file into memory should be *shared* between these two requirements.

Think about breaking the code for REQ2 and REQ3 into a set of functions that can be called in both scenarios. You must not write the same code for these requirements twice.

## REQ4: Unload List Items and Release Memory

After a file is loaded into memory, the user should be able to unload that file from memory through the **unload** command.

```
Enter your command: unload
> The list is unloaded.
```

This command will remove all the nodes from the linked-list and release their memory and return it to the Operating System. You should carefully handle deleting each element of the list and releasing the memory allocated to it. Your program should not crash.

**Hint:** After the current list has been unloaded from the memory, the display command should (REQ6) will display an empty list.

## REQ5: Quit the Program

At any stage in the program user should be able to quit the program using the **quit** command.

```
Enter your command: quit
> Goodbye.
```

It is very important to note that you must unload the list from memory prior to exiting from your program.

## REQ6: DISPLAY TELEPHONE BOOK CONTENT

A loaded address book file should be displayed on the screed using the **display** command. The following display is an example but does not match any of the existing files.

```
Enter your command: display

-------------------------------------------
| Pos | Serial | ID  | Name    | Telephones |
-------------------------------------------
| CUR | 1      | 100 | Alice   | 0411112221
|     | 2      | 101 | Bob     | 0411112222, 0422221111
|     | 3      | 102 | Ali     | 0411112223, 0422221112
|     | 4      | 103 | Reza    | 0411112224
|     | 5      | 104 | Ryan    | 0411112225, 0422221114
-------------------------------------------
| Total phone book entries: 5              |
-------------------------------------------
```

In addition to ID, Name and Telephone, there are other elements in the output:

- **Pos:** this shows an indicator (CUR) in front of the list element which is currently pointed to by the Current Position pointer. When *inserting* or *deleting* items, the current position will be used to execute the command.
- **Serial:** a serial number for each entry of the list in sequence.
- **Total phone book entries:** the current total elements in the address book. This counter should be updated if an item is removed from the list or added to it.

The size of **Name** columns in the list is equivalent to the size of the biggest element in that column + 2 (for the space characters on the left/right). You should always maintain this correct size and spacing.

In the event that there are no elements in the list, the **display** command generates the following output.

```
Enter your command: display

-------------------------------------------
| Pos | Serial | ID  | Name    | Telephones |
-------------------------------------------
|                                          |
-------------------------------------------
| Total phone book entries: 0              |
-------------------------------------------
```

**Implementation Guide**

In order to implement this requirement, you need to think about the following tasks: (1) going through the list items one by one; (2) finding the right width for the columns with dynamic width; and (3) printing the correct values to the output.

For the tasks (1) and (3), you are going to deal with loops and pointers to go through the doubly linked list and print the output.

For the task (2), you need to write an algorithm to find out the correct length of the Name field. Pay attention to all the sample outputs presented in this document and notice that the length of the Name column is **not** a fixed length (e.g. it's not fixed on the maximum). Therefore, you should think about how to calculate the correct length based currently stored in the doubly linked list. In order to do that, take a look at the tutorial and lab material and solutions of the second week of this course and review the *format strings* and *length specifiers* used with `printf()` function. You want to start by making sure you understand how these *format and length specifiers* work: "`%d, %03d, %0*d`".

To receive a complete mark for this requirement, you need to generate a correct output according to the given output samples. You might want to spend more time on calculating the length of the **Telephones** field too, but that is not compulsory.

## REQ7: MOVE FORWARD AND BACKWARD IN THE LIST

After loading a phone book file, the Current Position pointer should initially point to the first element of the linked-list (which is equal to the First Element pointer).

User can change the position of the Current Position pointer using the **forward** and **backward** commands. These two commands receive the number of items to move forward or backwards and move the position of the Current Position pointer accordingly.

In the example below, the Current Position pointer has been moved forward by 5 nodes. Remember that it initially pointed to the first node. Therefore, after moving 5 nodes forward it now points to the 6th node in the linked-list.

```
Enter your command: forward 5
Enter your command: display

-----------------------------------------
| Pos | Serial | ID  | Name    | Telephone  |
-----------------------------------------
|     | 1      | 100 | Alice   | 0411112221
|     | 2      | 101 | Bob     | 0411112222
|     | 3      | 102 | Ali     | 0411112223
|     | 4      | 103 | Reza    | 0411112224
|     | 5      | 104 | Ryan    | 0411112225
| CUR | 6      | 105 | Aaron   | 0411112226
|     | 7      | 106 | Jessica | 0411112227
|     | 8      | 107 | Xudong  | 0411112228
|     | 9      | 108 | James   | 0411112229
|     | 10     | 109 | Shaahin | 0411112230
|     | 11     | 110 | Matthew | 0411112231
-----------------------------------------
| Total phone book entries: 11          |
-----------------------------------------
```

The above position in now changed by further moves forward and then backward.

```
Enter your command: forward 2
Enter your command: backward 4
Enter your command: display

-----------------------------------------
| Pos | Serial | ID  | Name    | Telephone  |
-----------------------------------------
```

```
|      | 1      | 100 | Alice   | 0411112221
|      | 2      | 101 | Bob     | 0411112222
|      | 3      | 102 | Ali     | 0411112223
| CUR  | 4      | 103 | Reza    | 0411112224
|      | 5      | 104 | Ryan    | 0411112225
|      | 6      | 105 | Aaron   | 0411112226
|      | 7      | 106 | Jessica | 0411112227
|      | 8      | 107 | Xudong  | 0411112228
|      | 9      | 108 | James   | 0411112229
|      | 10     | 109 | Shaahin | 0411112230
|      | 11     | 110 | Matthew | 0411112231
-----------------------------------------------
| Total phone book entries: 11              |
-----------------------------------------------
```

The Current Position pointer must not go beyond the boundaries of the linked-list. That is, the **forward** command can move the Current Position pointer up to the last element in the list. After the last element is reached, issuing further **forward** commands will generate an error message. The **backward** command, similarly, can take the Current Position pointer up to the first node of the list. When the first element is reached, further **backward** commands will generate a correct error messages.

If moving forward or backward with the specified number of steps will go beyond the boundaries of the list, in addition to printing out a good error message, you should make sure that the CUR pointer will stay at its current position.

Pay a special attention to the scenario where there is no item in the list. You should handle these commands correctly regardless of the number of items in the list.

## REQ8: INSERT A NEW PHONE BOOK ENTRY

The phone book should allow the insertion of a new entry by using the **insert** command. The new node must have a unique ID. In the event that the specified ID exists in the list, a proper error message should be displayed. The new entry must be inserted at the end of the list (i.e. add to tail).

This command supports adding only one telephone number. Handling additional telephone numbers is handed by other commands.

```
Enter your command: insert 111,Test  Name1,1234567890
Enter your command: display

-------------------------------------------------
| Pos | Serial | ID  | Name        | Telephone  |
-------------------------------------------------
|     | 1      | 100 | Alice       | 0411112221
|     | 2      | 101 | Bob         | 0411112222
|     | 3      | 102 | Ali         | 0411112223
|     | 4      | 103 | Reza        | 0411112224
|     | 5      | 104 | Ryan        | 0411112225
|     | 6      | 105 | Aaron       | 0411112226
|     | 7      | 106 | Jessica     | 0411112227
|     | 8      | 107 | Xudong      | 0411112228
|     | 9      | 108 | James       | 0411112229
|     | 10     | 109 | Shaahin     | 0411112230
|     | 11     | 110 | Matthew     | 0411112231
```

```
| CUR | 12     | 111 | Test Name 1 | 1234567890
-------------------------------------------------
| Total phone book entries: 12                  |
-------------------------------------------------
```

When inserting new items to the list, user's input should be validated: (1) the inserted ID, Name and Telephone fields should have a valid length, and (2) the telephone field should contain only numbers.

**Implementation Guide**

Insertion to the linked lists are discussed in lectures. You need to create a new node in memory, and then change the relevant pointers so that the newly created node is added to the end of the list. You also need to update the *tail* pointer so that it points to the newly added node. Specifically, think about how to do the following steps:

1) Validate the given input fields to make sure they comply with the length and size requirements;

2) Create a new `AddressbookBookNode` node in memory dynamically, and store the entered data (in step 1) in the newly created node. This also involves adding an item to the corresponding `AddressBookArray`.

3) Modify the `previousNode` and `nextNode` pointers of the last node in the list (i.e. before insertion) and the newly created node (in step 2) such that they point to each other. There are no more nodes after the newly added node and, therefore, the next node of the newly added node has to be set to point to `null`.

4) Update the *tail* pointer to point to the newly added node at the end.

If you have completed the logic of loading the data file, adding a new node to the list involves, for the most part, reusing of many of the already written functions.

## REQ9: ADDING AND REMOVING TELEPHONE NUMBERS

The user of the address book should be able to use `add` and `remove` commands to, respectively, add a new telephone number to the address book and remove an existing number from the address book.

The changes will be applied to the *current node*, that is, the node that the CUR pointer is currently pointing to.

In the below example, the specified number will be added to the current node.

```
add 0441016182
```

If the specified number already exists for the current node, it should not be added again and a proper error message should be displayed.

In the below example, the specified number will be removed from the current node.

```
remove 0441016182
```

If the specified number does not exist for the current node, a proper error message should be displayed.

## REQ10: FIND AN ENTRY IN THE LIST

The user of the address book should be able to search for an entry by its *name*. This can be done by the **find** command.

If the specified name can be found, the Current Position pointer will be moved to the position of the node with the specified name. In the below example, the user searched for Jessica and the Current Position pointer has been moved to point to the node with (ID=106, Name=Jessica).

```
Enter your command: find Jessica
Enter your command: display

-------------------------------------------------
| Pos | Serial | ID  | Name      | Telephone  |
-------------------------------------------------
|     | 1      | 100 | Alice     | 0411112221 |
|     | 2      | 101 | Bob       | 0411112222 |
|     | 3      | 102 | Ali       | 0411112223 |
|     | 4      | 103 | Reza      | 0411112224 |
|     | 5      | 104 | Ryan      | 0411112225 |
|     | 6      | 105 | Aaron     | 0411112226 |
| CUR | 7      | 106 | Jessica   | 0411112227 |
|     | 8      | 107 | Xudong    | 0411112228 |
|     | 9      | 108 | James     | 0411112229 |
|     | 10     | 109 | Shaahin   | 0411112230 |
|     | 11     | 110 | Matthew   | 0411112231 |
|     | 12     | 111 | TestName1 | 1234567890 |
-------------------------------------------------
| Total phone book entries: 12                  |
-------------------------------------------------
```

If the specified name does not exist, a correct error message should be displayed as shown in the below example.

```
Enter your command: find non-existing-name
> Error: Unable to find node.
```

## REQ11: DELETE AN EXISTING ENTRY (BONUS)

User of the phone book should be able to delete an entry by using the **delete** command. This command will delete the node to which the Current Pointer is pointing. Applying the delete to the list shown in REQ10, Jessica will be deleted after the user uses **delete** command as shown in the example below.

```
Enter your command: delete
Enter your command: display

-------------------------------------------------
| Pos | Serial | ID  | Name      | Telephone  |
-------------------------------------------------
|     | 1      | 100 | Alice     | 0411112221 |
|     | 2      | 101 | Bob       | 0411112222 |
|     | 3      | 102 | Ali       | 0411112223 |
|     | 4      | 103 | Reza      | 0411112224 |
|     | 5      | 104 | Ryan      | 0411112225 |
|     | 6      | 105 | Aaron     | 0411112226 |
| CUR | 7      | 107 | Xudong    | 0411112228 |
```

```
|     | 8       | 108 | James     | 0411112229 |
|     | 9       | 109 | Shaahin   | 0411112230 |
|     | 10      | 110 | Matthew   | 0411112231 |
|     | 11      | 111 | TestName1 | 1234567890 |
-----------------------------------------------
| Total phone book entries: 11                |
-----------------------------------------------
```

It is important to handle all the various conditions that the **delete** command may work at. For example, the **delete** command should not cause any problems when used with no entries in the list. In that event, a proper error message should be displayed.

## REQ12: Sort the Address Book Entries by Name or ID (Bonus)

The user should be able to sort the entries in the address book using the **sort name** or **sort id** commands. The below sample output shows a scenario where the list is sorted by the name.

```
Enter your command: sort name
Enter your command: display

-----------------------------------------------
| Pos | Serial | ID  | Name      | Telephone  |
-----------------------------------------------
|     | 1      | 105 | Aaron     | 0411112226 |
|     | 2      | 102 | Ali       | 0411112223 |
|     | 3      | 100 | Alice     | 0411112221 |
|     | 4      | 101 | Bob       | 0411112222 |
|     | 5      | 108 | James     | 0411112229 |
|     | 6      | 110 | Matthew   | 0411112231 |
|     | 7      | 103 | Reza      | 0411112224 |
|     | 8      | 104 | Ryan      | 0411112225 |
|     | 9      | 109 | Shaahin   | 0411112230 |
|     | 10     | 111 | TestName1 | 1234567890 |
| CUR | 11     | 107 | Xudong    | 0411112228 |
-----------------------------------------------
| Total phone book entries: 11                |
-----------------------------------------------
```

The position of Current Position pointer should not change from where it used to be before sorting the entries in the address book.

> **Implementation Guide**
>
> Note that the implementation of this requirement needs that you understand function pointers and can use them.

## REQ13: Save to File in CSV Format

When the **save** command is used, the current content of the doubly linked list should be stored in a text file in CSV format. The name of the output file is specified right after the **save** command as shown in the following example.

```
Enter your command: save new_filename.txt
> Opening file new_filename.txt
> Writing list content to file...
> Closing file.
```

If the file cannot be opened or any other I/O error occurs, a correct error message should be displayed to the user. Your program should not crash.

The format of the output file is similar to the format of the three input files provided to you as part of the start-up code (refer to REQ2). In fact, once you save the list into a file, your program should be able to load the same file without any problem (that is REQ2 and REQ3).

## REQ14: Adding Dynamic Compilation to the Makefile

There is a Makefile provided as part of the start-up code, which **must** be used to compile your program. The body of this Makefile is as follows.

```
SOURCES=addressbook.c addressbook_list.c addressbook_array.c commands.c helpers.c
HEADERS=addressbook.h addressbook_list.h addressbook_array.h commands.h helpers.h
PROGRAM=addressbook
FLAGS=-ansi -pedantic -Wall

all:
	gcc $(FLAGS) -o $(PROGRAM) $(SOURCES)

clean:
	rm $(PROGRAM)

archive:
	zip $(USER)-a2 $(SOURCES) $(HEADERS) Makefile
```

The code that you develop for this project must compile with the provided Makefile without any errors or warnings. Specifically, you should be able to compile your program with the following command executed within the directory of your project.

```
./make
```

The following command must remove the compiled executable.

```
./make clean
```

Finally, you should create the archive files necessary for your submission to the blackboard using the following command.

```
./make archive
```

All the above commands have already been made available to you and requires no development work. To fulfil this requirement and receive the mark for it, you are required to modify the Makefile and add **dynamic compilation** to it.

When this task is completed, you will submit the modified Makefile as part of your archive (.zip) file.

### Implementation Guide

The `make` utility and the `Makefile` configuration files were discussed in lectures. Dynamic Compilation refers to a compilation process that only the modified files (as opposed to all the files in the project) are compiled by the `make` utility. In order to achieve this, you are required to specify the dependencies of the project files inside the `Makefile` so that the `make` utility can decide which files should be compiled for a given change. For an example of how to do this, refer to slides 18 and 19 of lecture six.

### REQ15: Demonstration of Your Work

A demonstration is required for this assignment in order to show your progress. This will occur in your scheduled lab classes. Demonstrations will be very brief, and you must be ready to show your work in a two-minute period when it is your turn.

The requirements categories as **General Requirements** will not be assessed during the demonstration.

During the demo session you are required to show the following:

1) Compiling your code without errors or warnings using the provided `Makefile` and by the `make` utility.
2) A working implementation of RQ2, RQ4, RQ5 and REQ6. For the purpose of demonstrating REQ2, it will be sufficient if your code loads everything from the file except the telephone numbers.

## GENERAL REQUIREMENTS

### GR1: Buffer Handling

This requirement relates to how well your programs handle "extra input".

To aid you with this requirement, we want you to use the *readRestOfLine()* function provided in the start-up code. Use of this function with *fgets()* is demonstrated in the source codes shared on Blackboard in week 3.

Marks will be deducted for the following buffer-related issues:

- Prompts being printed multiple times or your program "skipping" over prompts because left over input was accepted automatically. These problems are a symptom of not using buffer overflow code often enough.
- Program halting at unexpected times and waiting for the user to hit enter a second time.
- Calling your buffer clearing code after every input routine without first checking if a buffer overflow has in fact occurred.
- Using *fflush(stdin)*. Do not use this function in your code.
- Using *rewind(stdin)*. This function is not appropriate as it is not intended to be used for buffer clearing.
- The use of buffer clearing code may be avoided with *gets()* or *scanf()* for scanning string input. These functions are not safe because they do not check for the amount of input being accepted. You must avoid using *gets()* and *scanf()* functions.
- Using overly long character arrays as a method of handling buffer overflow. We want you to use the *readRestOfLine()* function.
- Other buffer related problems.

For example, what happens in your program when you try to enter a string of 40 characters when the limit is 20 characters?

### GR2: Input Validation

For functionality that we ask you to implement that relies upon the user entering input, you will need to check the length, range and type of all inputs where applicable.

For example, you will be expected to check the length of strings (e.g. if the entered data is a number or character), the ranges of numeric inputs (e.g. if the entered value is within the acceptable range) and the type of data (e.g. if the entered data is numeric).

For any detected invalid inputs, you are asked to re-prompt for this input. You should not truncate extra data or abort the function.

### GR3: Coding Conventions and Practices

Marks are awarded for good coding conventions/practices such as:

- Avoiding global variables.
- Avoiding *goto* statements.
- Consistent use of spaces or tabs for indentation. We recommend 3 spaces for every level of indentation. Each "block" of code should be indented one level.
- Keeping the lengths of each line of code to a reasonable maximum. A typical guideline is that no line of code should be longer than 80 columns.
- Commenting (including function header comments).
- Appropriate identifier names.
- Avoiding magic numbers.
- Avoiding platform specific code with *system()*.
- Checking the return values of important functions such as *fopen()*, *malloc()* and so on.
- General code readability.

### GR4: Functional Abstraction

We encourage the use of functional abstraction throughout your code. This is considered to be a good practice when developing software with multiple benefits. Abstracting code into separate functions reduces the possibility of bugs in your project, simplifies programming logic and make the debugging less complicated. We will look for some evidence of functional abstraction throughout your code.

As a rule, if you notice that you have the same or similar block of code in two different locations of your source, you can abstract this into a separate function. Another easy rule is that you should not have functions with more than 50 lines of code. If you have noticeably longer functions, you are expected to break that function into multiple logical parts and create new functions accordingly.

## Start-up Code

We provide you with a start-up code that you must use in implementing your program. The start-up code includes a number of files, which you must carefully read, understand and complete. Refer to the following sections for further details.

**You are required to use all of the functions provided in the start-up code.**

## STRUCTURE AND FILES

The start-up code includes the following 10 files:

- addressbook.c and addressbook.h: This is where the program starts. The *main()* function is inside this file.
- addressbook_list.c and addressbook_list.h: Contain the structures and functions necessary for creating and manipulating the doubly linked list.
- addressbook_array.c and addressbook_array.h: Contain the structures and functions necessary for creating and manipulating the array of telephone numbers for each address book entry.
- commands.c and commands.h: Contain the functions necessary for processing the commands that the user can enter.
- helpers.c and helpers.h: These files are used to declare the variables and functions which are used throughout the system and are shared among multiple other modules. A very good example is the *readRestOfLine()* function which must be used for reading input from the player.
- Makefile: A simple Makefile for building the project, removing the executable and building the archive file to submit the assignment.
- sml.txt, med.txt and lrg.txt: The address-book sample files that your program should be able to read and load into memory.

## HOW BONUS MARKS WILL BE APPLIED

Your first and second assignments are, together, worth 50% of your final mark in this course.

The bonus marks are included in this assignment to encourage you to further explore and advance your knowledge about the material presented in the course. If you complete the requirements that attract bonus marks, these additional marks will be used to compensate for the marks you lost in your assignments.

However, it must be noted that **the total marks of both of your assignments, together, will remain as 50% of your final mark**.

## PENALTIES

Marks will be deducted for the following:

- Compile errors and warnings.
- Fatal run-time errors such as segmentation faults, bus errors, infinite loops, etc.
- Files submitted in PC format (instead of Unix format). This can occur when transferring your files from home (PC) to yallara (Unix). You can remove extra end-of-line characters from PC formatted files using the *dos2unix* command. Read the dos2unix manual entry for more information.
- Missing files (affected components get zero marks).
- Files incorrectly named, or whole directories submitted.
- Not using all the functions of the start-up code.

Programs with compile errors that cannot be easily corrected by the marker will result in a maximum possible score of 40% of the total available marks for the assignment.

Any sections of the assignment that cause the program to terminate unexpectedly (i.e., segmentation fault, bus error, infinite loop, etc.) will result in a maximum possible score of 40% of the total available marks for those sections. Any sections that cannot be tested as a result of problems in other sections will also receive a maximum possible score of 40%.

It is not possible to resubmit your assignment after the deadline due to mistakes.

## MARK ALLOCATIONS

| Requirement | Mark | Bonus Mark |
|---|---|---|
| REQ1: PRINT STUDENT INFORMATION WHEN THE PROGRAM STARTS | 3 | |
| REQ2: LOAD A TELEPHONE BOOK FILE INTO MEMORY | 11 | |
| REQ3: LOAD A TELEPHONE BOOK FILE INTO MEMORY THROUGH COMMAND-LINE | 4 | |
| REQ4: UNLOAD LIST ITEMS AND RELEASE MEMORY | 6 | |
| REQ5: QUIT THE PROGRAM | 3 | |
| REQ6: DISPLAY TELEPHONE BOOK CONTENT | 12 | |
| REQ7: MOVE FORWARD AND BACKWARD IN THE LIST | 6 | |
| REQ8: INSERT A NEW PHONE BOOK ENTRY | 7 | |
| REQ9: ADDING AND REMOVING TELEPHONE NUMBERS | 7 | |
| REQ10: FIND AN ENTRY IN THE LIST | 6 | |
| REQ11: DELETE AN EXISTING ENTRY (BONUS) | | 4 |
| REQ12: SORT THE ADDRESS BOOK ENTRIES BY NAME OR ID (BONUS) | | 6 |
| REQ13: SAVE TO FILE IN CSV FORMAT | 5 | |
| REQ14: ADDING DYNAMIC COMPILATION TO THE MAKEFILE | 4 | |
| REQ15: DEMONSTRATION OF YOUR WORK | 10 | |
| GR1: BUFFER HANDLING | 4 | |
| GR2: INPUT VALIDATION | 4 | |
| GR3: CODING CONVENTIONS AND PRACTICES | 4 | |
| GR4: FUNCTIONAL ABSTRACTION | 4 | |
| **Total** | **100** | **10** |

## SUBMISSION INFORMATION

You are required to create a .zip file containing all of the files in your project. The Makefile provided to you allows you to create this .zip file without any problems. The procedure of creating such an archive file is explained in REQ14.

Submit the zip file to Blackboard before the submission deadline.

## LATE SUBMISSION PENALTY

Late submissions attract a marking deduction of 10% of the maximum mark attainable per day for the first 5 days (including weekends and public holidays). After this time, a 100% deduction is applied.

We recommend that you avoid submitting late where possible because it is difficult to make up the marks lost due to late submissions.