University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**Advanced Programming H**			
Coursework Number	**1a**			
Deadline	**Time:**	**4:30**	**Date:**	**27 October 2016**
% Contribution to final course mark	**8%**			
Solo or Group ✓	**Solo**	✓	**Group**	
Anticipated Hours	**15**			
Submission Instructions	**Described in section 5 of the attached handout.**			
Please Note: This Coursework cannot be Re-Done				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below.

The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

You must complete an "Own Work" form via
https://webapps.dcs.gla.ac.uk/ETHICS for all coursework
UNLESS submitted via Moodle

Find Duplicate Entries in a Mailing List

1 Requirement

A mailing list is a collection of people's names and addresses, used for bulk mailings. Mailing lists from different sources are often merged. A merged mailing list might contain duplicate entries, with only trivial differences, such as alternative forms of names, missing parts of addresses, etc. Duplicate entries result in excessive postage costs, and irritation for recipients of multiple copies of a single mailing. A program is required to report potential duplicate entries in a given mailing list.

2 Specification

The program's function is to report potential duplicate entries in a given mailing list. Two entries are to be treated as potential duplicates if they have identical surnames, post-codes, and house numbers; these comparisons are made in a case-insensitive way.

A mailing list file is a text file in which each entry occupies three successive lines:

- full name
- address
- post-code

The full name consists of an individual's surname, followed by a comma (','), followed by an optional title, followed by one or more forenames. The address line consists of a house number, street name, and city; the city is separated from the rest of the address by a comma (','). This file will be read from the standard input.

The program's output is a report on the standard output showing pairs of potentially duplicate entries. Both entries of a pair should be written out in full, **exactly as they appear in the mailing list file**. For example:

```
Potential duplicate
===================
Meldrew, Margaret
1 Happenstance Place, Epping, London
N11 3SR
==========
Meldrew, Victor
1 Happenstance Place, London
N11 3SR

Potential duplicate
===================
Sventek, Joe
University of Glasgow, Glasgow
G12 8QQ
==========
Sventek, Prof. Joseph
University of Glasgow, Glasgow
G12 8QQ
```

3 Design

You are provided with the source file for main(), and header files for two abstract data types – mentry.h and mlist.h.

mentry.h

```
#ifndef _MENTRY_H_
#define _MENTRY_H_

#include <stdio.h>

typedef struct mentry {
    char *surname;
```

```
    int house_number;
    char *postcode;
    char *full_address;
} MEntry;

/* me_get returns the next file entry, or NULL if end of file*/
MEntry *me_get(FILE *fd);

/* me_hash computes a hash of the MEntry, mod size */
unsigned long me_hash(MEntry *me, unsigned long size);

/* me_print prints the full address on fd */
void me_print(MEntry *me, FILE *fd);

/* me_compare compares two mail entries, returning <0, 0, >0 if
 * me1<me2, me1==me2, me1>me2
 */
int me_compare(MEntry *me1, MEntry *me2);

/* me_destroy destroys the mail entry
 */
void me_destroy(MEntry *me);

#endif /* _MENTRY_H_ */
```

The **struct mentry**, and the corresponding typedef **MEntry**, define the data structure for a mailing list entry. For those members of the structure that are character pointers, you will have to use **malloc()** to allocate storage for these members, and store the pointers into the structure to return to the user. The entire 3-line address is stored as a single string at **full_address**; the **surname** and **postcode** members are there to store single case versions of the surname, and the postcode from line 3 – I recommend that you remove non alpha-numeric characters from the postcode, as well. The **house_number** member is to hold the integer house number at the beginning of the 2nd line of the address.

The constructor for this ADT is **me_get()**; it reads the next mailing list entry from **fd**, and returns a pointer to an **MEntry** structure containing the mailing list entry; if there is an error, or you have reached the end of file, **NULL** is returned. You will also have to use **malloc()** to allocate this **MEntry** structure to return to the user.

me_hash() computes a hash value for the surname+postcode+house_number, returning a value between 0 and **size-1**. The textbook shows you how to build a simple hash table, and discusses hash functions for strings.

me_print() prints the full address on the specified file descriptor.

me_compare() compares two mail entries, returning **<0, 0, >0** if **me1<me2**, **me1==me2**, **me1>me2**, respectively.

me_destroy() returns all heap-allocated storage associated with the mailing list entry.

mlist.h

```
#ifndef _MLIST_H_
#define _MLIST_H_

#include "mentry.h"

typedef struct mlist MList;
```

```
extern int ml_verbose;          /* if true, prints diagnostics on stderr */

/* ml_create - created a new mailing list */
MList *ml_create(void);

/* ml_add - adds a new MEntry to the list;
 * returns 1 if successful, 0 if error (malloc)
 * returns 1 if it is a duplicate */
int ml_add(MList **ml, MEntry *me);

/* ml_lookup - looks for MEntry in the list, returns matching entry or NULL */
MEntry *ml_lookup(MList *ml, MEntry *me);

/* ml_destroy - destroy the mailing list */
void ml_destroy(MList *ml);

#endif /* _MLIST_H_ */
```

The typedef for MList is to hide the representation of a mailing list.

The extern declaration of ml_verbose is to enable the main program to indicate to the mailing list source code that it should print diagnostics on stderr as it works. The source code in finddupl.c shows you how this is set.

ml_create() is the constructor for this ADT, called with no arguments, and returns an MList. It will return NULL if it is unsuccessful.

ml_add() adds an MEntry to the list, returning 1 if it is successful, 0 if not (due to malloc() failures); if you request that a duplicate to an existing entry be added, ml_add ignores the request, yet still returns 1.

ml_lookup() looks for an entry in the list that matches the 2nd argument; if found, it is returned as the function value; if not, NULL is returned.

ml_destroy() returns all heap-allocated storage associated with the entries in the list. The implementation must be sure to also return heap-allocated storage associated with the individual entries.

finddupl.c

```
#include <stdio.h>
#include "mentry.h"
#include "mlist.h"

static void usage(void) {
        fprintf(stderr, "usage: finddupl [-v]\n");
}

int main(int argc, char *argv[]) {
        MEntry *mep, *meq;
        MList *ml;
        char *p;
        int c;
        int varg = 0;

        if (argc > 2) {
                usage(); return -1;
        }
        if (argc == 2) {
                p = argv[1];
```

```
                if (*p++ != '-') {
                        usage(); return -1;
                }
                while ((c = *p++) != '\0') {
                        if (c =='v' || c == 'V')
                                varg++;
                        else {
                                fprintf(stderr, "Illegal flag: %c\n", c);
                                usage(); return -1;
                        }
                }
        }
        ml_verbose = varg;

        ml = ml_create();
        while ((mep = me_get(stdin)) != NULL) {
                meq = ml_lookup(ml, mep);
                if (meq == NULL)
                        (void) ml_add(&ml, mep);
                else {
                        printf("Potential duplicate\n");
                        printf("==================\n");
                        me_print(mep, stdout);
                        printf("==========\n");
                        me_print(meq, stdout);
                        printf("\n");
                }
        }
        ml_destroy(ml);
        return 0;
}
```

The main program is invoked as

```
./finddupl [-v]
```

If the **-v** argument is present, the **extern** variable "ml_verbose" is set to TRUE, and the mailing list ADT is expected to print diagnostics on **stderr** as it works.

Most of the code in finddupl.c is to process arguments. The mainline functionality consists of the following pseudocode:

```
create a mailing list
while (get next entry is successful)
    lookup this entry
    if a potential duplicate is found
        print out the two potential duplicates
    else
        add this entry to the list
destroy the mailing list
```

4 Implementation

You are to implement **mentry.c** and **mlist.c**. The implementations must match the function prototypes in the headers listed in section 3 above.

The implementation of `mlist` must be a hash table. For full credit, the hash table must resize itself whenever any one hash bucket exceeds 20 entries. If `verbose == TRUE`, then you should print diagnostics on `stderr` whenever you have to resize the table.

An archive containing source files, test input and output files is available on the Moodle page. In addition to `finddupl.c`, `mentry.h` and `mlist.h`, there is also a linked list implementation of `mlist.c` in the archive. This will permit you to test your implementation of `mentry.c` against a working, albeit inefficient, implementation of `MList`.

5 Submission

You will submit your solutions electronically by submitting the following files on Aropa2: http://aropa2.gla.ac.uk/aropa/

Students in Glasgow should submit via the University of Glasgow Aropa site, and students in Singapore should submit via the SIT site.

- mentry.c
- mlist.c
- report.pdf – A brief report in PDF outlining the state of your solution, and identifying any bugs or limitations you are aware of. It is important that the report is accurate. For example it is offensive to report that everything works when it won't even compile.

If you have produced a static hash table implementation, then submit to Exercise 1a Static Table. If you have produced a dynamic hash table implementation, then submit to Exercise 1a Dynamic Table. Only submit to one or the other!

Each of your source files must start with an "authorship statement" as follows:

- state your name, your login, and the title of the assignment (APH Exercise 1)
- state either "This is my own work as defined in the Academic Ethics agreement I have signed." or "This is my own work except that …", as appropriate.

You must complete an "Own Work" form via https://webapps.dcs.gla.ac.uk/ETHICS.

Assignments will be checked for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. There are very good tools for detecting collusion.

6 Marking Scheme

Your submission will be marked on a 100 point scale. There is substantial emphasis on **WORKING** submissions, and you will note that a large fraction of the points are reserved for this aspect. It is to your advantage to ensure that whatever you submit compiles, links, and runs correctly. The information returned to you will indicate the number of points awarded for the submission, and will also inform you the band associated with that number of points. Note that the mapping from points to bands is different for Designated Degree students than for Honours, so you should not be surprised if the same number of points yield different bands when comparing notes with your classmates.

You must be sure that your code works correctly on the lab 64-bit Linux systems, regardless of which platform you use for development and testing. Leave enough time in your development to fully test on the lab machines before submission.

As indicated in the handout, you can choose to turn in two forms of mlist:

- if it is implemented as a static hash table (i.e. it does not resize itself as it runs), only 60 of the 100 total points are available to you.
- if it is implemented as a dynamic hash table, resizing itself whenever one of the hash buckets exceeds 20 entries, all 100 total points are available to you.

The static hash table marking scheme is as follows:

Points	Description
10	Your report – honestly describes the state of your submission
20	MEntry ADT 6 for workable solution (looks like it should work) 2 if it successfully compiles 2 if it compiles with no warnings 6 if it works correctly on S.txt 4 if there are no memory management issues
30	MList ADT 9 for workable solution (looks like it should work) 1 if it successfully compiles 2 if it compiles with no warnings 1 if it successfully links with finddupl 2 if it links with no warnings 3 if it works correctly with S.txt and M.txt 3 if it works correctly with 10,000 entry unseen file 3 if it works correctly with 1,000,000 entry unseen file 6 if there are no memory leaks

The dynamic hash table marking scheme is as follows:

Points	Description
10	Your report – honestly describes the state of your submission
20	MEntry ADT 6 for workable solution (looks like it should work) 2 if it successfully compiles 2 if it compiles with no warnings 6 if it works correctly on S.txt 4 if there are no memory management issues
70	MList ADT 18 for workable solution (looks like it should work) 2 if it successfully compiles 4 if it compiles with no warnings 2 if it successfully links with finddupl 4 if it links with no warnings 5 if it works correctly with S.txt and M.txt 8 if it works correctly with 10,000 entry unseen file 15 if it works correctly with 1,000,000 entry unseen file 12 if there are no memory leaks

Several things should be noted about the marking schemes:

- Your report needs to be honest. Stating that everything works and then finding that it won't even compile is offensive. The 10 points associated with the report are probably the easiest 10 points you will ever earn as long as you are honest.
- If your solution does not look workable, then the points associated with successful compilation and lack of compilation errors are **not** available to you. This prevents you from handing in a stub implementation for each of the methods in each ADT and receiving points because they compile without errors, but do nothing.
- The points associated with "workable solution" are the maximum number of points that can be awarded. If the reviewer deems that only part of the solution looks workable, then you will be awarded a portion of the points in that category.