COMP 2510
Assignment 3

1 Introduction

The purpose of this assignment is to write a program to sort records. Each record consists of an ID, a name & a score. The program reads records from standard input & stores them in a dynamic array. After finish reading, the program sorts the array & prints back the records in some sorted order.

2 Data Structures

We'll be using the following structures to store records:

```
#define IDSIZE    10
#define NAMESIZE  20

typedef struct {
  char  last[NAMESIZE];    /* last name (1 word), e.g., "simpson" */
  char  first[NAMESIZE];   /* first name (1 word), e.g., "homer" */
} name;

typedef struct {
  char  id[IDSIZE];        /* ID ('a' followed by 8 digits), e.g., "a12345678" */
  name  name;              /* name (as defined above) */
  int   score;             /* score (between 0 & 100 inclusive), e.g., 25 */
} record;
```

Each record consists of an ID, a name & a score. A name in turn consists of a last name & a first name.

The program uses a dynamic array of records. This dynamic array starts with 0 record, goes to 1 record & then doubles in size in each later re-allocation. This means that the number of records in the array goes from 0 to 1, then to 2, then to 4, then to 8, then to 16 & so on. The array is resized whenever the user needs to add a record but all "slots" in the dynamic array are already in use.

3 The Program

We'll call our program `rs` (for "record sort"). `rs` reads records from standard input & stores them in the dynamic array mentioned above. This reading & storing of records continues until end-of-file is reached. If the program is invoked without arguments, it then proceeds to sort the records in ascending order of IDs & prints the sorted records to standard output.

The program accepts one command-line argument to specify the sorting order. In general, the argument consists of one `n`-option (either `n+` or `n-`) & one `s`-option (either `s+` or `s-`) concatenated together. But there are 4 shortcuts.

When an `n`-option & an `s`-option are combined together, they have the following meaning:

- `n+` (respectively `n-`) specifies ascending (respectively descending) order of names. What this means is that records are sorted in ascending (respectively descending) order of last names, & if two or more records have the same last name, those records are then sorted in ascending (respectively descending) order of their first names.
- `s+` (respectively `s-`) is used to specify ascending (respectively descending) order of scores.

The order of the `n`-option & `s`-option in the command-line argument is important. For example, `rs s-n+` has a different meaning from `rs n+s-`:

- `rs s-n+` sorts in descending order of scores; if two or more records have the same score, they are then sorted in ascending order of their names.
- `rs n+s-` sorts in ascending order of names; if two or more records have the same name, they are then sorted in descending order of their scores.

This means that the first option in the command-line argument specifies the main (or primary) sorting order; if several records are equivalent under this sorting order, the second option specifies how those records should be sorted (the secondary sorting order).

There are 4 shortcuts where only an `n`-option or only an `s`-option is specified:

- `rs n+` is the same as `rs n+s+`
- `rs n-` is the same as `rs n-s-`
- `rs s+` is the same as `rs s+n+`
- `rs s-` is the same as `rs s-n-`

Essentially, what this means is that if only one option is specified, the secondary sorting order defaults to be the same as the primary sorting order.

The following show some examples of valid invocations:

```
rs s-
rs s-n+
rs n+s-
```

The following invocations are invalid:

```
rs s-s+
rs s-n+s-
rs s+ n-
```

If an invocation is invalid, a usage message is printed to standard error & the program exits.

4 Input/Output

As mentioned above, records are read from standard input. However, the program does not prompt for data. Input is read a line at a time. Each (valid) line contains information about a record. A valid line, i.e., a line that contains a valid record, consists of an ID, a first name, a last name, a score & an optional comment, all separated by whitespaces & in that order. This means that there must be at least 4 words in a valid line. Furthermore, the ID must consist of an ‘a’ (lowercase) followed by 8 digits, the score must be an integer between 0 & 100 inclusive, the first & last names must each be a word whose length is strictly less than `NAMESIZE`. Note that there is no restriction on the characters in the two names. However, letters in them are converted to lowercase before they are stored in a record. The program silently skips any line that is not valid. You may assume that each line has fewer than `LINESIZE` characters where `LINESIZE` is a macro with a value of 256.

The following are examples of valid lines:

```
a13579246 homer SIMPSON12 25  # names can contain any character
a98765432 ned flanders  099    # score is 99 - leading 0’s are allowed
a66666666  Montgomery   Burns 89
a43218765 Bart Simpson  35
a12345678 Lisa  Simpson  90
a32768901  bart simpson 25
```

And here are some invalid ones:

```
a1234567  maggie simpson # invalid ID
a55555555 marge simpson # invalid score (# is not a valid score)
a22222222 bart simpson 45abc # 35abc is not an integer
a44444444 bart simpson  34.5 # 34.5 is not an integer
a12345678 lisa simpson 105 # score exceeds 100
a33333333 apu nahasapeemapetilonian 95 # last name too long
```

The sorted records are printed to *standard output*. With the 6 valid lines above, & with the program invoked as `rs n+s-`, the output is:

```
a66666666 : burns, montgomery : 89
a98765432 : flanders, ned : 99
a43218765 : simpson, bart : 35
a32768901 : simpson, bart : 25
a12345678 : simpson, lisa : 90
a13579246 : simpson12, homer : 25
```

Note the output format for a record: ID, space, colon, space, last name, comma, space, first name, space, colon, space, score, followed by a newline.

5 Additional Requirements

This assignment is basically an exercise in using dynamic memory & `qsort` — *your program must use dynamic memory in the manner specified in order to get any credit for this assignment.*

Your program must be able to print "debug" information. This printing of debug information is controlled by a macro named `DEBUG`. If `DEBUG` is defined when your program is compiled, it should print a "hash" symbol (`#`) on its own line to standard error each time the array of records is resized (including when it is first allocated).

If your program does not print this debug information, we may assume that you are not using dynamic memory in the manner specified by the assignment.

Your program must also reside in multiple C source files. At a minimum, the comparison functions used by `qsort` must be in a separate C source file from the main program. You must also provide at least one header file. However, do not use external variables.

Dynamic memory must be explicitly deallocated before the program exits. The `qsort` function in the standard C library must be used to sort the records.

6 A Useful Data Structure

It may be a good idea to encapsulate the dynamic array of records the program uses in a structure. This makes it simpler to pass it into functions. One possibility is to use something like the following:

```
typedef struct {
  record  *data;    /* pointer to dynamic array of records */
  size_t   nalloc;  /* number of records allocated */
  size_t   nused;   /* number of records used */
} record_list;
```

7 Submission & Grading

This assignment is due at 11pm on Saturday, April 1, 2017. Submit a zip file to `ShareIn` in the directory:

```
\COMP\2510\a3\set<X>\
```

where `<X>` is your set. Your zip file must be named `<lastname><firstname>_<id>.zip`, where `<lastname>` is your last name, `<firstname>` your first name & `<id>` your student ID. (See below if you need to make multiple submissions.) For example, `SimpsonHomer_a12345678.zip`. Do not use spaces in this file name.

Your zip file must unzip directly to your source files *without creating any directories.* We'll compile your files using

```
  gcc -ansi -W -Wall -pedantic *.c
```

after unzipping (where `gcc` is the GNU C compiler).

Do not put executables in your zip file. Also, do not submit rar files—they will not be accepted.

If you need to submit more than one version, name the zip file of each later version with a version number after your ID, e.g., `SimpsonHomer_a12345678_v2.zip`. If more than one version is submitted, we'll only mark the version with the highest version number. (When submitting multiple versions, there must be exactly one zip file with the highest version number. If it is unclear to us which version to mark, you may fail to get credit for the assignment.)

The restriction to ANSI C & its standard library applies as in previous assignments. Your program must reside in multiple C source files & must compile without warnings or errors with the command shown above. Furthermore, you must use dynamic memory in the manner specified in section 2 in order to get any credit for this assignment.

If the above requirements are met, the grade breakdown for this assignment is approximately as follows:

Design & code clarity	10%
Command-line validation	10%
Input (includes validation)	20%
Handling dynamic memory (requires debug mode)	20%
Sorting (includes output)	40%

Note: Your program will basically be tested using I/O redirection & comparing your output with a reference output using a file comparison program. Your program will be deemed incorrect if its redirected output does not match exactly the reference output. Sample input & output files will be provided.