COMP 2510

Assignment 3

1 Introduction

The purpose of this assignment is to write a program to sort records. Each record consists of a name & a score. The program uses a dynamic array of pointers to records. It reads & dynamically allocates each record & stores its address in the array. After finishing reading, it 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 NAMESIZE  20

typedef struct {
  char  last[NAMESIZE];    /* last name (1 word) */
  char  first[NAMESIZE];   /* first name (1 word) */
} name;

typedef struct {
  name  name;
  int   score;             /* score (between 0 & 100 inclusive) */
} record;
```

Each record contains a name & a score. A name in turn consists of a last name & a first name.

For this assignment, each record structure is dynamically allocated & the addresses of the dynamically-allocated structures are stored in a dynamic array of pointers. This dynamic array of pointers is resized in blocks of `BLOCK` pointers, where `BLOCK` is a macro configurable at compile time & which has a default value of 2. This means that the number of pointers in the array goes from 0 to `BLOCK`, then to `2 * BLOCK`, then to `3 * BLOCK`, & so on. The array is resized when all pointers in it are used up & there are more records to store.

Note that we are using a dynamic array of pointers to structures, not a dynamic array of structures. A reason for this is that sorting an array of pointers requires less copying than sorting an array of structures.

3 The Program

We'll call our program `rsort`. `rsort` reads records from standard input & stores (the addresses of dynamically-allocated copies of) them in the dynamic array mentioned above. This reading & storing of records continues until end-of-file is encountered. If the program is invoked without arguments, it then proceeds to print all the records (to standard output) in the order they were read.

Command-line arguments can be used to specify the sorting order. The supported arguments fall into 2 groups: the `n`-options (`+n` & `-n`) & the `s`-options (`+s` & `-s`).

- `+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.

WINTER 2016

At most one `n`-option & at most one `s`-option can be specified. (This means that there can be at most 2 options.) The following are examples of valid invocations:

```
rsort -s
rsort -s +n
rsort +n -s
```

The following invocations are invalid:

```
rsort -s +s
rsort -s +n -s
```

If an invocation is invalid, a usage message should be printed to standard error.

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

- `rsort -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.
- `rsort +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 specifies the main sorting order; if several records are equivalent under this sorting order, the second option specifies how those records should be sorted. (Recall that there can be at most 2 options.)

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 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 3 words in a valid line. Furthermore, 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`. (But there is no restrictions on the characters in the word.) The program silently skips any line that is not valid. You may assume that each line has fewer than 256 characters.

The following are examples of valid lines:

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

And here are some invalid ones:

```
marge simpson # invalid score (# is not a valid score)
bart simpson  34.5 # 34.5 is not an integer
lisa simpson 105 # score exceeds 100
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 `rsort +n -s`, the output is:

WINTER '16

```
burns montgomery 89
flanders ned 99
simpson bart 35
simpson bart 25
simpson lisa 90
simpson12 homer 25
```

Note that

- there are no leading zeros in the score
- the last & first names are in all lowercase with the last name preceding the first name
- case-insensitive string comparison is used when determining sorting order (this makes sense since names are printed in all lowercase)
- record members are separated by single spaces

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 the following additional information to *standard error*:

- a "percent" symbol (`%`) each time a record structure is dynamically allocated;
- an "hash" symbol (`#`) each time the array of pointers is resized (including when it is first allocated);
- an "at" symbol (`@`) each time a block of dynamic memory is deallocated. (This includes record structures & the array of pointers.)

Each of the above symbols should be followed by a newline character when printed.

If your program does not print "debug" information, it may be assumed 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 file from the main program. You must also provide at least one header file. However, do not use "global" 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 pointers 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;    /* points to dynamic array of pointers */
  size_t    nalloc;  /* number of pointers allocated */
  size_t    nused;   /* number of pointers used */
  /* add any additional members you think necessary */
} record_list;
```

WINTER '16

7 Submission & Grading

This assignment is due at 11 pm, Saturday, March 26, 2016. 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 `<name_id>.zip`, where `<name_id>` is your name and student ID separated by an underscore (e.g., `SimpsonHomer_a12345678.zip`). Do not use spaces to separate your last and first names. Your zip file must unzip directly to your source files without creating any directory. We'll basically compile your files using

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

after unzipping.

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 name, e.g., `SimpsonHomer_a12345678_v2.zip`. If more than one version is submitted, we'll only mark the version with the highest version number. This means that there must be exactly one zip file with the highest version number; if there are two or more of them, you will not receive credit for the assignment.

The restriction to ANSI C & its standard library applies as in previous assignments. Your program must compile without warnings or errors under `gcc` with the following options:

```
-ansi -W -Wall -pedantic
```

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%
Validation (input & command-line)	20%
Handling dynamic memory (requires debug mode)	15%
Sorting	45%
Output format	10%

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.

WINTER 16