

```
1: /**
2:  * Brandon Rodriguez
3:  * CS 3240
4:  * 09-19-17
5:  * a1 (Assignment 1)
6:  */
7:
8:
9: /**
10:  * Description:
11:  *
12:  * Reads in song data from Songs.csv and saves in two "permanent" database files.
13:  */
14:
15:
16: /**
17:  * Known Issues:
18:  *
19:  * Due to linear sort, valgrind may "freeze" up for a bit.
20:  * Should finish in no more than a minute or two.
21:  */
22:
23:
24: // Import headers.
25: #include <fcntl.h>
26: #include <stdio.h>
27: #include <sys/stat.h>
28: #include <sys/types.h>
29: #include <unistd.h>
30: #include "apue.h"
31: #include "HelperHeader.h"
32:
33:
34: // Define Vars.
35: #define BUFFER_SIZE 4096
36: #define ARRAY_SIZE 500
37:
38:
39: // Variables.
40: typedef struct {
41:     char* artist;
42:     char* song_name;
43:     char* album_name;
44:     float* duration;
45:     int* year;
46:     double* hotttnesss;
47: } songs_struct;           // Songs struct.
48:
49: int test_file_bool = 0;   // Check if using smaller test csv or not.
50: int songs_array_max = ARRAY_SIZE; // Saves current size of songs array.
51: int song_index = 0;       // Current song index.
52: songs_struct** songs_array; // The array which holds all song pointers.
53: songs_struct** temp_array; // The temp array for sorting.
54:
55:
56: // Method Declaration.
57: int open_file();           // Initially opens file to read.
58: void read_file();          // Reads in chunk of file.
59: void read_line();          // Separates file chunk into lines.
60: void tokenize_line();      // Separates lines into fields.
61: void resize_array();       // Increase buffer size of song array.
62: void populate_array();     // Populates array with fields.
63: void print_all_songs();    // Prints info for all songs in array.
```

105
100

Very well done

```
64: void print_song_info();           // Prints all info for provided song.
65: void sort_array();               // Linearly sorts array by song name.
66: void save_array();               // Saves entire array to file(s).
67: void save_line();                // Creates/saves single line of song data.
68: void write_song_field();         // Writes a single song_filed to file.
69: void exit_program();             // Frees memory and closes program.
70:
71:
72: /**
73:  * Program's main.
74:  * Initializes and runs program.
75:  */
76: int main(int argc, char* argv[]) {
77:     write(1, "Reading csv...\n", 16);
78:     songs_array = calloc(songs_array_max, sizeof(songs_struct *));
79:     int file_descriptor;
80:
81:     // Check test csv bool. If true, read smaller csv.
82:     if (test_file_bool) {
83:         file_descriptor = open_file("Data/SongCSV_Small.csv", O_RDONLY);
84:     } else { // Else use full data set.
85:         file_descriptor = open_file("Data/SongCSV.csv", O_RDONLY);
86:     }
87:
88:     read_file(file_descriptor);
89:     write(1, "Sorting arrays...\n", 19);
90:     sort_array();
91:     //print_all_songs();
92:     write(1, "Saving to file...\n", 19);
93:     save_array();
94:     write(1, "Finished.\n", 11);
95:     write(1, "Closing program...\n", 20);
96:     close(file_descriptor);
97:     exit_program();
98: }
99:
100:
101: /**
102:  * Opens file.
103:  */
104: int open_file(char* file_location, int operator_flags) {
105:     int file_descriptor;
106:     // Handle for creating file.
107:     if ((operator_flags == (O_CREAT | O_APPEND | O_WRONLY)) ||
108:         (operator_flags == (O_CREAT | O_TRUNC | O_WRONLY))) {
109:         // Umask for issues with other_write privledges not assigning.
110:         mode_t oldval = umask(0);
111:         file_descriptor = open(file_location, operator_flags, 0777);
112:         umask(oldval);
113:         if (file_descriptor < 0) {
114:             err_sys("Failed to open file.");
115:         }
116:     } else { // Handle for non-creation.
117:         file_descriptor = open(file_location, operator_flags);
118:         if (file_descriptor < 0) {
119:             err_sys("Failed to open file.");
120:         }
121:     }
122:     return file_descriptor;
123: }
124:
125:
126: /**
```

```
127:  * Read next chunk of file.
128:  *
129:  * Return: Next chunk of file read.
130:  */
131: void read_file(int file_descriptor) {
132:     char *read_buffer;
133:     char *temp_buffer;
134:     int current_read_int = 0;
135:     int index;
136:     int offset_amount;
137:     off_t read_value;
138:     off_t offset_size;
139:
140:     read_buffer = calloc(1, BUFFER_SIZE);
141:
142:     while ((read_value = read(file_descriptor, read_buffer, (BUFFER_SIZE - 1))) != 0) {
143:         if (read_value < 0) {
144:             err_sys("Failed to read line.");
145:         }
146:
147:         // Start from end of buffer. Read backwards until first null terminator
148:         // is found.
149:         index = BUFFER_SIZE - 1;
150:         offset_amount = 0;
151:         temp_buffer = read_buffer;
152:         while (index > 0) {
153:             if (temp_buffer[index] == '\n') {
154:                 // Found newline terminator. Save location and exit loop.
155:                 offset_amount = index;
156:                 index = 0; // To exit loop.
157:             }
158:             index--;
159:         }
160:
161:         // Rewind to start of current read, then go to actual read value
162:         // of buffer after cutoff.
163:         size_t actual_read_size = strlen(read_buffer);
164:         if (actual_read_size != 4095) {
165:             actual_read_size--;
166:         }
167:
168:         offset_size = lseek(file_descriptor, -(actual_read_size), SEEK_CUR);
169:         if (offset_size < 0) {
170:             err_sys("Failed to update pointer on read in.");
171:         }
172:         offset_size = lseek(file_descriptor, offset_amount, SEEK_CUR);
173:         if (offset_size < 0) {
174:             err_sys("Failed to update pointer on read in.");
175:         }
176:
177:         // Remove "garbage" data at end of read and pass forward.
178:         index = offset_amount + 1;
179:         while (index < BUFFER_SIZE) {
180:             read_buffer[index] = '\0';
181:             index++;
182:         }
183:         read_line(read_buffer, offset_size);
184:
185:         // Reset read buffer for next read.
186:         free(read_buffer);
187:         read_buffer = calloc(1, BUFFER_SIZE);
188:
189:         current_read_int++;
```

```
190:     }
191:     free(read_buffer);
192: }
193:
194:
195: /**
196:  * Separates individual lines out of file chunk.
197:  */
198: void read_line(char *read_buffer, size_t read_size) {
199:     int chunk_index = 0;
200:     int line_index = 0;
201:     char *line_buffer;
202:
203:     line_buffer = calloc(1, BUFFER_SIZE);
204:
205:     // Loop through entire chunk.
206:     while ((chunk_index < read_size) && (strcmp(&read_buffer[chunk_index], "\0") != 0))
{
207:         // Loop through individual line and copy values.
208:         if (read_buffer[chunk_index] == '\r') {
209:             // Newline so current line is done being read.
210:             line_buffer[line_index] = '\0';
211:             tokenize_line(line_buffer);
212:             line_index = 0;
213:             chunk_index++;
214:
215:             // Free and reallocate memory for next line.
216:             free(line_buffer);
217:             line_buffer = calloc(1, BUFFER_SIZE);
218:         } else {
219:             // Newline not found so save value and keep reading.
220:             line_buffer[line_index] = read_buffer[chunk_index];
221:             line_index++;
222:             chunk_index++;
223:         }
224:     }
225:     free(line_buffer);
226: }
227:
228:
229: /**
230:  * Separates individual fields out of line.
231:  */
232: void tokenize_line(char *line_buffer) {
233:     int line_index = 0;
234:     int field_index = 0;
235:     int field_number = 0;
236:     char* field_buffer;
237:
238:     // Check to make sure there are still open spots in songs_array.
239:     if (song_index >= songs_array_max) {
240:         resize_array();
241:     }
242:
243:     field_buffer = calloc(1, BUFFER_SIZE);
244:     songs_array[song_index] = calloc(1, sizeof(songs_struct));
245:
246:     while (strcmp(&line_buffer[line_index], "\0") != 0) {
247:         // Check for comma to denote next field.
248:         if (line_buffer[line_index] == ',') {
249:             // Found comma so end of field. Handle appropriately.
250:             field_index = 0;
251:             line_index++;
```

```
252:
253:     populate_array(field_number, field_buffer);
254:     field_number++;
255:
256:     free(field_buffer);
257:     field_buffer = calloc(1, BUFFER_SIZE);
258: } else {
259:     // No comma found.
260:     field_buffer[field_index] = line_buffer[line_index];
261:     field_index++;
262:     line_index++;
263: }
264: }
265: // Last field so populate values one last time.
266: populate_array(field_number, field_buffer);
267: song_index++;
268: free(field_buffer);
269: }
270:
271:
272: /**
273:  * Increase current buffer size and rebuffer songs array.
274:  * Call to initialize, and then anytime the array runs out of space.
275:  */
276: void resize_array() {
277:     songs_array_max = songs_array_max * 2;
278:
279:     songs_array = realloc(songs_array, songs_array_max * sizeof(songs_struct* ));
280:     if (songs_array != NULL) {
281:         //write(1, "Allocation successful.\n", 24);
282:     } else {
283:         err_dump("Could not allocate memory for realloc.");
284:     }
285: }
286:
287:
288: /**
289:  * Takes provided field value and associates with song in struct.
290:  */
291: void populate_array(int field_number, char *field_buffer) {
292:     int temp_int;
293:     int* temp_int_ptr;
294:     float temp_float;
295:     float* temp_float_ptr;
296:     double temp_double;
297:     double* temp_double_ptr;
298:
299:     // Remove unnecessary quotes from string value fields.
300:     if (field_number == 3 || field_number == 8 || field_number == 17) {
301:         field_buffer = remove_quotes(field_buffer);
302:     }
303:
304:     // Go through provided field and assign values appropriately with switch.
305:     switch(field_number) {
306:         case 3: // Album name.
307:             songs_array[song_index]->album_name = copy_string(field_buffer);
308:             //printf("Album: %s\n", songs_array[song_index]->album_name);
309:             break;
310:         case 8: // Artist name.
311:             songs_array[song_index]->artist = copy_string(field_buffer);
312:             //printf("Artist: %s\n", songs_array[song_index]->artist);
313:             break;
314:         case 10: // Duration.
```

```
315:     temp_float = atof(field_buffer);
316:     temp_float_ptr = &temp_float;
317:     songs_array[song_index]->duration = copy_float(temp_float_ptr);
318:     //printf("Duration: %.2f\n", *songs_array[song_index]->duration);
319:     break;
320: case 14: // Hottnesss.
321:     temp_double = atof(field_buffer);
322:     // Check if NAN.
323:     if (temp_double != temp_double) {
324:         temp_double = 0;
325:     }
326:     temp_double_ptr = &temp_double;
327:     songs_array[song_index]->hottnesss = copy_double(temp_double_ptr);
328:     //printf("Hottnesss: %.2f\n", *songs_array[song_index]->hottnesss);
329:     break;
330: case 17: // Song name.
331:     songs_array[song_index]->song_name = copy_string(field_buffer);
332:     //printf("Name: %s\n", songs_array[song_index]->song_name);
333:     break;
334: case 18: // Year.
335:     temp_int = atoi(field_buffer);
336:     temp_int_ptr = &temp_int;
337:     songs_array[song_index]->year = copy_int(temp_int_ptr);
338:     //printf("Year: %d\n", *songs_array[song_index]->year);
339:     break;
340: default:
341:     // Do nothing. This is not a field we care about.
342:     break;
343: }
344: }
345:
346:
347: /**
348:  * Uses linear sort to arrange array by song name.
349:  */
350: void sort_array() {
351:     int sorted_bool = 0;    // Holds if array has been sorted during this call.
352:     int index = 1;         // Current index.
353:     songs_struct* temp_song;
354:
355:     // Loop through all array elements and sort if necessary.
356:     // Starts at 1 due to comparing two elements at once.
357:     while ((songs_array[index] != NULL) && (songs_array[index]->song_name != NULL)) {
358:         if (strcmp(songs_array[index - 1]->song_name, songs_array[index]->song_name) >
0) {
359:             temp_song = songs_array[index - 1];
360:             songs_array[index - 1] = songs_array[index];
361:             songs_array[index] = temp_song;
362:             sorted_bool = 1;
363:         }
364:         index++;
365:     }
366:
367:     if (sorted_bool == 1) {
368:         sort_array();
369:     }
370: }
371:
372:
373: /**
374:  * Prints all songs.
375:  */
376: void print_all_songs() {
```

```
377:     int index = 0;
378:     while (songs_array[index] != NULL) {
379:         print_song_info(songs_array[index]);
380:         index++;
381:     }
382: }
383:
384:
385: /**
386:  * Attempts to find song with provided name. Prints result.
387:  */
388: void print_song_info(songs_struct* song) {
389:     printf("Song Name: %s\n", song->song_name);
390:     printf("Album Name: %s\n", song->album_name);
391:     printf("Artist Name: %s\n", song->artist);
392:     printf("Duration: %.2f\n", *song->duration);
393:     printf("Year: %d\n", *song->year);
394:     printf("Hottness: %f\n", *song->hottness);
395:     printf("\n");
396: }
397:
398:
399: /**
400:  * Saves array to file. Supposed to act as a permanent "database" of sorts.
401:  */
402: void save_array() {
403:     int song_directory_descriptor;
404:     int song_binary_descriptor;
405:
406:
407:     // Check test csv bool. If true, write to smaller csv.
408:     if (test_file_bool) {
409:         song_directory_descriptor = open_file("Data/SongDirectory_Small", O_CREAT | O_T
RUNC | O_WRONLY);
410:         song_binary_descriptor = open_file("Data/BinarySongData_Small", O_CREAT | O_TRU
NC | O_WRONLY);
411:     } else { // Else use full data set.
412:         song_directory_descriptor = open_file("Data/SongDirectory", O_CREAT | O_TRUNC |
O_WRONLY);
413:         song_binary_descriptor = open_file("Data/BinarySongData", O_CREAT | O_TRUNC | O
_WRONLY);
414:     }
415:
416:     song_index = 0;
417:     while (songs_array[song_index] != NULL) {
418:         save_line(song_directory_descriptor, song_binary_descriptor);
419:         song_index++;
420:     }
421:
422:     close(song_directory_descriptor);
423:     close(song_binary_descriptor);
424: }
425:
426:
427: /**
428:  * Takes a single song from the songs array and creates buffer to save.
429:  */
430: void save_line(int directory_descriptor, int binary_descriptor) {
431:     void* line_buffer;
432:     char* char_pointer;
433:     float* float_pointer;
434:     double* double_pointer;
435:     int* int_pointer;
```

```
436:     int index;
437:     int current_line_size = 0;
438:     size_t name_size = ((strlen(songs_array[song_index]->song_name) * sizeof(char)) + 1
);
439:     size_t album_size = ((strlen(songs_array[song_index]->album_name) * sizeof(char)) +
1);
440:     size_t artist_size = ((strlen(songs_array[song_index]->artist) * sizeof(char)) + 1)
;
441:     size_t duration_size = (sizeof(float));
442:     size_t hotttnesss_size = (sizeof(double));
443:     size_t year_size = (sizeof(int));
444:     int* current_line_size_ptr = &current_line_size;
445:
446:     line_buffer = calloc(1, BUFFER_SIZE);
447:     char_pointer = (char*)line_buffer;
448:
449:     // printf("SongName:          %s\n", songs_array[song_index]->song_name);
450:     // printf("Size of SongName:    %ld\n", (strlen(songs_array[song_index]->song_name)
* sizeof(char));
451:     // printf("AlbumName:          %s\n", songs_array[song_index]->album_name);
452:     // printf("Size of AlbumName:   %ld\n", (strlen(songs_array[song_index]->album_name)
) * sizeof(char));
453:     // printf("Artist:             %s\n", songs_array[song_index]->artist);
454:     // printf("Size of Artist:      %ld\n", (strlen(songs_array[song_index]->artist)) *
sizeof(char));
455:     // printf("Duration:           %.2f\n", *songs_array[song_index]->duration);
456:     // printf("Size of Duration:    %ld\n", sizeof(float));
457:     // printf("Hotttnesss:         %f\n", *songs_array[song_index]->hotttnesss);
458:     // printf("Size of Hotttnesss:  %ld\n", sizeof(double));
459:     // printf("Year:               %d\n", *songs_array[song_index]->year);
460:     // printf("Size of Year:        %ld\n", sizeof(int));
461:
462:     // Get total song value size by adding field memory together.
463:     current_line_size += name_size;
464:     current_line_size += album_size;
465:     current_line_size += artist_size;
466:     current_line_size += duration_size;
467:     current_line_size += hotttnesss_size;
468:     current_line_size += year_size;
469:
470:     // printf("Song Struct Size: %d\n", current_line_size);
471:     // printf("\n\n");
472:
473:     // Save directory values to file.
474:     ssize_t write_size = write(directory_descriptor, current_line_size_ptr, sizeof(int)
);
475:     if (write_size < sizeof(int)) {
476:         err_sys("Failed to write to song_directory file.");
477:     }
478:
479:     off_t offset_size = lseek(directory_descriptor, sizeof(int), SEEK_CUR);
480:     if (offset_size < 0) {
481:         err_sys("Failed to update song_directory write pointer.");
482:     }
483:
484:     // Create song buffer to save values to file.
485:     // Song Name.
486:     index = 0;
487:     while (index < name_size) {
488:         *char_pointer = songs_array[song_index]->song_name[index];
489:         ++char_pointer;
490:         index++;
491:     }
```



```
492:
493:     // Album Name.
494:     index = 0;
495:     while (index < album_size) {
496:         *char_pointer = songs_array[song_index]->album_name[index];
497:         ++char_pointer;
498:         index++;
499:     }
500:
501:     // Artist Name.
502:     index = 0;
503:     while (index < artist_size) {
504:         *char_pointer = songs_array[song_index]->artist[index];
505:         ++char_pointer;
506:         index++;
507:     }
508:
509:     // Duration.
510:     index = 0;
511:     float_pointer = (float*)char_pointer;
512:     *float_pointer = songs_array[song_index]->duration[index];
513:     ++float_pointer;
514:
515:     // Hottnesss.
516:     index = 0;
517:     double_pointer = (double*)float_pointer;
518:     *double_pointer = songs_array[song_index]->hottnesss[index];
519:     ++double_pointer;
520:
521:     // Year.
522:     index = 0;
523:     int_pointer = (int*)double_pointer;
524:     *int_pointer = songs_array[song_index]->year[index];
525:     ++int_pointer;
526:
527:     // Write buffer to file.
528:     write_size = write(binary_descriptor, line_buffer, current_line_size);
529:     if (write_size < current_line_size) {
530:         err_sys("Failed to write to song_binary file.");
531:     }
532:
533:     offset_size = lseek(binary_descriptor, current_line_size, SEEK_CUR);
534:     if (offset_size < 0) {
535:         err_sys("Failed to update song_binary write pointer.");
536:     }
537:
538:     free(line_buffer);
539: }
540:
541:
542: /**
543:  * Writes single field to file.
544:  */
545: void write_song_field(int binary_descriptor, void* field, int field_size) {
546:     ssize_t write_size = write(binary_descriptor, field, field_size);
547:     if (write_size < field_size) {
548:         err_sys("Failed to write to song_binary file.");
549:     }
550:
551:     off_t offset_size = lseek(binary_descriptor, field_size, SEEK_CUR);
552:     if (offset_size < 0) {
553:         err_sys("Failed to update song_binary write pointer.");
554:     }
```

```
555: }
556:
557:
558: /**
559:  * Frees remaining memory and so program can close cleanly.
560:  */
561: void exit_program() {
562:     int index = 0;
563:     while (songs_array[index] != NULL) {
564:         free(songs_array[index]->album_name);
565:         free(songs_array[index]->artist);
566:         free(songs_array[index]->song_name);
567:         free(songs_array[index]->duration);
568:         free(songs_array[index]->year);
569:         free(songs_array[index]->hotttnesss);
570:         free(songs_array[index]);
571:         index++;
572:     }
573:     free(songs_array);
574: }
```

```
1: #include      <errno.h>          /* for definition of errno */
2: #include      <stdarg.h>         /* ANSI C header file */
3: #include      "apue.h"
4:
5: static void    err_doit(int, const char *, va_list);
6:
7: char          *pname = NULL;      /* caller can set this from argv[0] */
8:
9: /* Nonfatal error related to a system call.
10:  * Print a message and return. */
11:
12: void
13: err_ret(const char *fmt, ...)
14: {
15:     va_list     ap;
16:
17:     va_start(ap, fmt);
18:     err_doit(1, fmt, ap);
19:     va_end(ap);
20:     return;
21: }
22:
23: /* Fatal error related to a system call.
24:  * Print a message and terminate. */
25:
26: void
27: err_sys(const char *fmt, ...)
28: {
29:     va_list     ap;
30:
31:     va_start(ap, fmt);
32:     err_doit(1, fmt, ap);
33:     va_end(ap);
34:     exit(1);
35: }
36:
37: /* Fatal error related to a system call.
38:  * Print a message, dump core, and terminate. */
39:
40: void
41: err_dump(const char *fmt, ...)
42: {
43:     va_list     ap;
44:
45:     va_start(ap, fmt);
46:     err_doit(1, fmt, ap);
47:     va_end(ap);
48:     abort();          /* dump core and terminate */
49:     exit(1);          /* shouldn't get here */
50: }
51:
52: /* Nonfatal error unrelated to a system call.
53:  * Print a message and return. */
54:
55: void
56: err_msg(const char *fmt, ...)
57: {
58:     va_list     ap;
59:
60:     va_start(ap, fmt);
61:     err_doit(0, fmt, ap);
62:     va_end(ap);
63:     return;
```

```
64: }
65:
66: /* Fatal error unrelated to a system call.
67:  * Print a message and terminate. */
68:
69: void
70: err_quit(const char *fmt, ...)
71: {
72:     va_list      ap;
73:
74:     va_start(ap, fmt);
75:     err_doit(0, fmt, ap);
76:     va_end(ap);
77:     exit(1);
78: }
79:
80: /* Print a message and return to caller.
81:  * Caller specifies "errnoflag". */
82:
83: static void
84: err_doit(int errnoflag, const char *fmt, va_list ap)
85: {
86:     int          errno_save;
87:     char         buf[MAXLINE];
88:
89:     errno_save = errno;           /* value caller might want printed */
90:     vsprintf(buf, fmt, ap);
91:     if (errnoflag)
92:         sprintf(buf+strlen(buf), ": %s", strerror(errno_save));
93:     strcat(buf, "\n");
94:     fflush(stdout);                /* in case stdout and stderr are the same */
95:     fputs(buf, stderr);
96:     fflush(stderr);              /* SunOS 4.1.* doesn't grok NULL argument */
97:     return;
98: }
```

```
1: /**
2:  * Brandon Rodriguez
3:  * CS 3240
4:  * 09-19-17
5:  */
6:
7:
8: /**
9:  * Helper functions to have consistent value copying and error handling.
10:  *
11:  *
12:  * Copy Functions:
13:  *     Copy functions use memcpy to copy data, and error.c's err_dump on
14:  *     failure.
15:  *
16:  *     Copy functions require a "source" pointer of equivalent typing, and
17:  *     return a pointer to copied item's location.
18:  *
19:  *
20:  * String Functions:
21:  *     String manipulator functions to make string handling easier.
22:  *
23:  *     String manipulator functions require an "input" string and return
24:  *     a modified copy of original string.
25:  */
26:
27:
28: #include <ctype.h>
29: #include <string.h>
30: #include "apue.h"
31:
32:
33: // ***** //
34: // Copy Functions //
35: // ***** //
36:
37: /**
38:  * Copies string from destination to source.
39:  *
40:  * Return: Copy of source string.
41:  */
42: char* copy_string(char *source_ptr) {
43:     char *copy_ptr;
44:     copy_ptr = calloc((strlen(source_ptr) + 1), sizeof(char));
45:     if (copy_ptr != NULL) {
46:         memcpy(copy_ptr, source_ptr, ((strlen(source_ptr) + 1) * sizeof(char)));
47:     } else {
48:         err_dump("Could not allocate memory for string calloc.");
49:     }
50:     return copy_ptr;
51: }
52:
53:
54: /**
55:  * Copies float from destination to source.
56:  *
57:  * Return: Copy of source int.
58:  */
59: int* copy_int(int *source_ptr) {
60:     int *copy_ptr;
61:     copy_ptr = calloc(1, (sizeof(int) + 1));
62:     if (copy_ptr != NULL) {
63:         memcpy(copy_ptr, source_ptr, sizeof(int));
```

```
64:     } else {
65:         err_dump("Could not allocate memory for int calloc.");
66:     }
67:     return copy_ptr;
68: }
69:
70:
71: /**
72:  * Copies float from destination to source.
73:  *
74:  * Return: Copy of source float.
75:  */
76: float* copy_float(float *source_ptr) {
77:     float *copy_ptr;
78:     copy_ptr = calloc(1, (sizeof(float) + 1));
79:     if (copy_ptr != NULL) {
80:         memcpy(copy_ptr, source_ptr, sizeof(float));
81:     } else {
82:         err_dump("Could not allocate memory for float calloc.");
83:     }
84:     return copy_ptr;
85: }
86:
87:
88: /**
89:  * Copies double from destination to source.
90:  *
91:  * Return: Copy of source double.
92:  */
93: double* copy_double(double *source_ptr) {
94:     double *copy_ptr;
95:     copy_ptr = calloc(1, (sizeof(double) + 1));
96:     if (copy_ptr != NULL) {
97:         memcpy(copy_ptr, source_ptr, sizeof(double));
98:     } else {
99:         err_dump("Could not allocate memory for double calloc.");
100:     }
101:     return copy_ptr;
102: }
103:
104:
105: // ***** //
106: // String Manipulation Functions //
107: // ***** //
108:
109: /**
110:  * Converts provided string to lowercase.
111:  *
112:  * Return: Lowercase version of initial string.
113:  */
114: char* to_lower_case(char* input_string) {
115:     int index = 0;
116:     char* return_string = calloc((strlen(input_string) + 1), sizeof(char));
117:     while (input_string[index]) {
118:         return_string[index] = tolower(input_string[index]);
119:         index++;
120:     }
121:     return return_string;
122: }
123:
124:
125: /**
126:  * Converts provided string to uppercase.
```

```
127:  *
128:  * Return: Uppercase version of initial string.
129:  */
130: char* to_upper_case(char* input_string) {
131:     int index = 0;
132:     char* return_string = calloc((strlen(input_string) + 1), sizeof(char));
133:     while (input_string[index]) {
134:         return_string[index] = toupper(input_string[index]);
135:         index++;
136:     }
137:     return return_string;
138: }
139:
140:
141: /**
142:  * Converts provided string to:
143:  *     First letter of each word is uppercase.
144:  *     All other letters lowercase.
145:  *
146:  * Return: Converted version of initial string.
147:  */
148: char* first_letter_upper(char* input_string) {
149:     int index = 0;
150:     char* return_string = calloc((strlen(input_string) + 1), sizeof(char));
151:     while (input_string[index]) {
152:         if (index == 0) { // First char is always upper.
153:             return_string[index] = toupper(input_string[index]);
154:         } else { // Check all other chars. If space precedes, then upper.
155:             if (input_string[index - 1] == ' ') {
156:                 return_string[index] = toupper(input_string[index]);
157:             } else { // No space found.
158:                 return_string[index] = tolower(input_string[index]);
159:             }
160:         }
161:         index++;
162:     }
163:     return return_string;
164: }
165:
166:
167: /**
168:  * Removes quotes from provided string.
169:  *
170:  * Return: Quote-less version of initial string.
171:  */
172: char* remove_quotes(char* input_string) {
173:     size_t string_length = strlen(input_string);
174:     int orig_index;
175:     int replace_index = 0;
176:
177:     for (orig_index = 0; orig_index < string_length; orig_index++) {
178:         if (input_string[orig_index] != '\"') {
179:             input_string[replace_index] = input_string[orig_index];
180:             replace_index++;
181:         }
182:     }
183:
184:     // Fill rest of values with null terminators.
185:     while (replace_index < string_length) {
186:         input_string[replace_index] = '\\0';
187:         replace_index++;
188:     }
189: }
```

```
190:     return input_string;
191: }
192:
193:
194: /**
195:  * Removes newline character from string.
196:  *
197:  * Return: Newline-less version of initial string.
198:  */
199: char* remove_newline(char* input_string) {
200:     size_t string_length = strlen(input_string);
201:     int orig_index;
202:     int replace_index = 0;
203:
204:     for (orig_index = 0; orig_index < string_length; orig_index++) {
205:         if (input_string[orig_index] != '\n') {
206:             input_string[replace_index] = input_string[orig_index];
207:             replace_index++;
208:         }
209:     }
210:
211:     // Fill rest of values with null terminators.
212:     while (replace_index < string_length) {
213:         input_string[replace_index] = '\0';
214:         replace_index++;
215:     }
216:
217:     return input_string;
218: }
```



```
1: /**
2:  * Brandon Rodriguez
3:  * CS 3240
4:  * 09-19-17
5:  * a1 (Assignment 1)
6:  */
7:
8:
9: /**
10:  * Description:
11:  *
12:  * Prompts user for input.
13:  *
14:  * Upon needing to access songs, program reads from two files:
15:  *     Directory: Stores the byte size of each song, in order, as they vary.
16:  *     Binary: Stores the actual data for each song.
17:  *
18:  * Data read from files is temporary, and only one song is read in at a time.
19:  */
20:
21:
22: /**
23:  * Known Issues:
24:  *
25:  * Since the songs are only sorted by song name, searching by album is slow.
26:  * Album search automatically occurs if provided "song name" is not found.
27:  */
28:
29:
30: // Import headers.
31: #include <fcntl.h>
32: #include <stdio.h>
33: #include <sys/stat.h>
34: #include <sys/types.h>
35: #include <unistd.h>
36: #include "apue.h"
37: #include "HelperHeader.h"
38:
39:
40: // Define Vars.
41: #define BUFFER_SIZE 4096
42: #define ARRAY_SIZE 500
43:
44:
45: // Variables.
46: typedef struct {
47:     char* artist;
48:     char* song_name;
49:     char* album_name;
50:     float* duration;
51:     int* year;
52:     double* hotttnesss;
53: } songs_struct; // Songs struct.
54:
55: int test_file_bool = 0; // Check if using smaller test csv or not.
56: int total_song_count; // Total number of songs in database.
57: songs_struct* song_holder_struct; // The holder of current song data.
58:
59:
60: // Method Declaration.
61: int open_file(); // Initially opens file to read.
62: void read_song_of_index(); // Gets song value of index.
63: void get_song_count(); // Gets count of total songs in database.
```

```
64: void free_song_holder();           // Frees memory used by song holder.
65: void populate_song();              // Populates song holder with fields.
66: void print_song_info();            // Prints all info for provided song.
67: void print_all_songs();            // Prints info for all songs in array.
68: void print_help_text();            // Prints helper text for user.
69: void find_song();                  // Find song with given name.
70: songs_struct* search_array_by_song(); // Binary search for name.
71: songs_struct* search_array_by_album(); // Linear search for album.
72:
73:
74: /**
75:  * Program's main.
76:  * Initializes and runs program.
77:  */
78: int main(int argc, char* argv[]) {
79:     int run_program_bool;           // Bool to keep program running.
80:     int index;                      // Buffer index;
81:     int atoi_index;                 // Used if user provides song index.
82:     char* user_input_buffer;        // Buffer of user input.
83:     char* temp_buffer;              // Temp buffer.
84:     char* user_input_string;        // Current string of user's input.
85:     char* input_to_lower;           // User input as all lower.
86:     off_t read_value;               // Return value of read attempt.
87:
88:     // Initialize vars.
89:     run_program_bool = 1;
90:     song_holder_struct = calloc(1, sizeof(songs_struct));
91:
92:     get_song_count();
93:     print_help_text();
94:
95:     // Actually run program for user input.
96:     while (run_program_bool) {
97:         user_input_buffer = calloc(1, BUFFER_SIZE);
98:
99:         // Read input from console.
100:        // Puts into buffer incase user inputs multiple lines at once,
101:        // such as from a test file.
102:        write(1, "Enter input: ", 14);
103:        read_value = read(0, user_input_buffer, BUFFER_SIZE);
104:        if (read_value < 0) {
105:            err_sys("Failed to read line.");
106:        }
107:        write(1, "\n", 2);
108:        temp_buffer = user_input_buffer;
109:
110:
111:        while (*temp_buffer != '\0') {
112:            index = 0;
113:            user_input_string = calloc(1, BUFFER_SIZE);
114:
115:            // Get current value from buffer.
116:            while (*temp_buffer != '\n') {
117:                user_input_string[index] = *temp_buffer;
118:                ++temp_buffer;
119:                index++;
120:            }
121:            ++temp_buffer;
122:
123:            // Read user input and handle accordingly.
124:            if ((user_input_string != NULL) && (strcmp(user_input_string, "") != 0)) {
125:                input_to_lower = to_lower_case(user_input_string);
126:
```

```
127:         // Check if user needs reprinting of help text.
128:         if (strcmp(input_to_lower, "help") == 0) {
129:             print_help_text();
130:         }
131:         // Check if user has requested to exit program.
132:         else if (strcmp(input_to_lower, "zzz") == 0) {
133:             run_program_bool = 0;
134:             printf("Exiting Program...\n");
135:         }
136:         // Check if user has requested to print all songs.
137:         else if (strcmp(input_to_lower, "all") == 0) {
138:             print_all_songs();
139:         }
140:         // Check if song index was provided
141:         else if ((atoi_index = atoi(input_to_lower)) != 0) {
142:             read_song_of_index(atoi_index);
143:             print_song_info();
144:         }
145:         else { // Attempt to find song with given name.
146:             find_song(input_to_lower);
147:         }
148:         free(input_to_lower);
149:     }
150:     free(user_input_string);
151: }
152: free(user_input_buffer);
153: }
154:
155: // // Free memory and close program.
156: free_song_holder();
157: }
158:
159:
160: /**
161:  * Opens file.
162:  */
163: int open_file(char* file_location, int operator_flags) {
164:     int file_descriptor;
165:     file_descriptor = open(file_location, operator_flags);
166:     if (file_descriptor < 0) {
167:         err_sys("Failed to open file.");
168:     }
169:     return file_descriptor;
170: }
171:
172:
173: void read_song_of_index(int desired_index) {
174:     int song_directory_descriptor; // File descriptor for directory data.
175:     int song_binary_descriptor;   // File descriptor for binary song data.
176:     int current_index = 0;
177:     int* song_size;
178:     ssize_t read_value;
179:     off_t offset_size;
180:     void* read_buffer;
181:
182:     song_size = calloc(1, sizeof(int));
183:     read_buffer = calloc(1, BUFFER_SIZE);
184:
185:
186:     // Check test csv bool. If true, read smaller csv.
187:     if (test_file_bool) {
188:         song_directory_descriptor = open_file("Data/SongDirectory_Small", O_RDONLY);
189:         song_binary_descriptor = open_file("Data/BinarySongData_Small", O_RDONLY);
```

```
190:     } else { // Else use full data set.
191:         song_directory_descriptor = open_file("Data/SongDirectory", O_RDONLY);
192:         song_binary_descriptor = open_file("Data/BinarySongData", O_RDONLY);
193:     }
194:
195:     while (current_index < (desired_index + 1)) {
196:         // Read directory file.
197:         read_value = read(song_directory_descriptor, song_size, sizeof(int));
198:         if (read_value < 0) {
199:             err_sys("Failed to read line.");
200:         }
201:
202:         if (current_index < desired_index) {
203:             // Read binary file.
204:             // I don't understand why but this read seems necessary
205:             // or else lseek seems to behave oddly.
206:             // Spent hours on this, am baffled.
207:             read_value = read(song_binary_descriptor, read_buffer, *song_size);
208:             if (read_value < 0) {
209:                 err_sys("Failed to read line.");
210:             }
211:
212:             // Offset directory file.
213:             offset_size = lseek(song_directory_descriptor, sizeof(int), SEEK_CUR);
214:             if (offset_size < 0) {
215:                 err_sys("Failed to update pointer on read in.");
216:             }
217:
218:             // Offset binary file.
219:             offset_size = lseek(song_binary_descriptor, *song_size, SEEK_CUR);
220:             if (offset_size < 0) {
221:                 err_sys("Failed to update pointer on read in.");
222:             }
223:         }
224:         current_index++;
225:     }
226:
227:     // Read binary file.
228:     read_value = read(song_binary_descriptor, read_buffer, *song_size);
229:     if (read_value < 0) {
230:         err_sys("Failed to read line.");
231:     }
232:
233:     // Populate song struct.
234:     populate_song(read_buffer);
235:
236:     free(song_size);
237:     free(read_buffer);
238:     close(song_directory_descriptor);
239:     close(song_binary_descriptor);
240: }
241:
242:
243: /**
244:  * Populates song holder with values from provided buffer.
245:  */
246: void populate_song(ssize_t song_buffer) {
247:     int index;
248:     int* int_buffer;
249:     int* int_pointer;
250:     char* char_buffer;
251:     char* char_pointer;
252:     float* float_buffer;
```

```
253:     float* float_pointer;
254:     double* double_buffer;
255:     double* double_pointer;
256:
257:     if (song_holder_struct != NULL ) {
258:         free_song_holder();
259:     }
260:     song_holder_struct = calloc(1, sizeof(songs_struct));
261:
262:     // Get Song Name.
263:     char_pointer = ((char*)song_buffer);
264:     char_buffer = calloc(1, BUFFER_SIZE);
265:     index = 0;
266:     while (char_pointer[0] != '\0') {
267:         char_buffer[index] = *char_pointer;
268:         ++char_pointer;
269:         index++;
270:     }
271:     song_holder_struct->song_name = copy_string(char_buffer);
272:     ++char_pointer;
273:     free(char_buffer);
274:
275:     // Get Album.
276:     char_buffer = calloc(1, BUFFER_SIZE);
277:     index = 0;
278:     while (char_pointer[0] != '\0') {
279:         char_buffer[index] = *char_pointer;
280:         ++char_pointer;
281:         index++;
282:     }
283:     song_holder_struct->album_name = copy_string(char_buffer);
284:     ++char_pointer;
285:     free(char_buffer);
286:
287:     // Get Artist.
288:     char_buffer = calloc(1, BUFFER_SIZE);
289:     index = 0;
290:     while (char_pointer[0] != '\0') {
291:         char_buffer[index] = *char_pointer;
292:         ++char_pointer;
293:         index++;
294:     }
295:     song_holder_struct->artist = copy_string(char_buffer);
296:     ++char_pointer;
297:     free(char_buffer);
298:
299:     // Get Duration.
300:     float_buffer = calloc(1, (sizeof(float) + 1));
301:     float_pointer = (float*)char_pointer;
302:     *float_buffer = *float_pointer;
303:     song_holder_struct->duration = copy_float(float_buffer);
304:     ++float_pointer;
305:     free(float_buffer);
306:
307:     // Get Hottnesss.
308:     double_buffer = calloc(1, (sizeof(double) + 1));
309:     double_pointer = (double*)float_pointer;
310:     *double_buffer = *double_pointer;
311:     song_holder_struct->hottnesss = copy_double(double_buffer);
312:     ++double_pointer;
313:     free(double_buffer);
314:
315:     // Get Year.
```

```
316:     int_buffer = calloc(1, (sizeof(int) + 1));
317:     int_pointer = (int*)double_pointer;
318:     *int_buffer = *int_pointer;
319:     song_holder_struct->year = copy_int(int_buffer);
320:     ++int_pointer;
321:     free(int_buffer);
322: }
323:
324:
325: void get_song_count() {
326:     int song_directory_descriptor; // File descriptor for directory data.
327:     int* dummy_holder;
328:     ssize_t read_value;
329:     off_t offset_size;
330:
331:     total_song_count = 0;
332:     dummy_holder = calloc(1, sizeof(int));
333:
334:     // Check test csv bool. If true, read smaller csv.
335:     if (test_file_bool) {
336:         song_directory_descriptor = open_file("Data/SongDirectory_Small", O_RDONLY);
337:     } else { // Else use full data set.
338:         song_directory_descriptor = open_file("Data/SongDirectory", O_RDONLY);
339:     }
340:
341:     // Read directory file until no more values and count index.
342:     while ((read_value = read(song_directory_descriptor, dummy_holder, sizeof(int))) !=
0) {
343:         if (read_value < 0) {
344:             err_sys("Failed to read line.");
345:         }
346:
347:         // Offset directory file.
348:         offset_size = lseek(song_directory_descriptor, sizeof(int), SEEK_CUR);
349:         if (offset_size < 0) {
350:             err_sys("Failed to update pointer on read in.");
351:         }
352:         total_song_count++;
353:     }
354:     printf("Total songs: %d\n\n", total_song_count);
355:
356:     free(dummy_holder);
357:     close(song_directory_descriptor);
358: }
359:
360:
361: /**
362:  * Frees memory used by song holder.
363:  */
364: void free_song_holder() {
365:     if (song_holder_struct != NULL) {
366:         free(song_holder_struct->album_name);
367:         free(song_holder_struct->artist);
368:         free(song_holder_struct->song_name);
369:         free(song_holder_struct->duration);
370:         free(song_holder_struct->year);
371:         free(song_holder_struct->hotttnesss);
372:         free(song_holder_struct);
373:     }
374: }
375:
376:
377: /**
```

```
378:  * Prints info of provided song.
379:  */
380: void print_song_info() {
381:     printf("Song Name: %s\n", song_holder_struct->song_name);
382:     printf("Album Name: %s\n", song_holder_struct->album_name);
383:     printf("Artist Name: %s\n", song_holder_struct->artist);
384:     printf("Duration: %.2f\n", *song_holder_struct->duration);
385:     printf("Year: %d\n", *song_holder_struct->year);
386:     printf("Hottness: %f\n", *song_holder_struct->hottness);
387:     printf("\n\n");
388: }
389:
390:
391:
392: /**
393:  * Prints all songs.
394:  */
395: void print_all_songs() {
396:     int index = 0;
397:     read_song_of_index(index);
398:     while (index < total_song_count) {
399:         printf("Song #: %d\n", index);
400:         print_song_info();
401:         index++;
402:         read_song_of_index(index);
403:     }
404: }
405:
406:
407: /**
408:  * Prints helper text.
409:  * Called on program start and again if user types "Help".
410:  */
411: void print_help_text() {
412:     printf("To display help text again, type 'Help'.\n");
413:     printf("To print all songs, type 'All'.\n");
414:     printf("To exit program, type 'ZZZ'.\n");
415:     printf("Otherwise, type name of song you wish to locate.\n\n");
416: }
417:
418:
419: // /**
420: //  * Searches through array and finds song based on user input.
421: //  */
422: void find_song(char* user_input_string) {
423:     char* temp_string;
424:     songs_struct* song;
425:     songs_struct* temp_song = calloc(1, sizeof(songs_struct));
426:
427:     // // Search for user's input, first by song_name, then by album name.
428:     song = search_array_by_song(temp_song, 0, total_song_count, user_input_string);
429:
430:     // Check returned song.
431:     if (song != NULL) {
432:         print_song_info();
433:         free(temp_song);
434:     } else { // Not found. Search again by album instead of song_name.
435:         printf("Could not find song with name. Searching by album.\n");
436:         printf("(This may take a bit. Album search is slower.)\n");
437:         song = search_array_by_album(0, total_song_count, user_input_string);
438:         if (song != NULL) {
439:             print_song_info();
440:         } else {
```

```
441:         // Could not find song or album.
442:         temp_string = first_letter_upper(user_input_string);
443:         err_msg("Could not find song or album with name: %s\n", temp_string);
444:         free(temp_string);
445:     }
446: }
447: }
448:
449:
450: /**
451:  * Uses binary search to locate song with desired name.
452:  *
453:  * Return: Either valid song or NULL, depending on if record is found or not.
454:  */
455: songs_struct* search_array_by_song(songs_struct* return_song,
456:     int first_index, int last_index, char* desired_record) {
457:
458:     char* temp_string;
459:
460:     // First check if this is the end of search or not.
461:     if (first_index >= last_index) {
462:
463:         // End of search. Handle accordingly.
464:         read_song_of_index(first_index);
465:         temp_string = to_lower_case(song_holder_struct->song_name);
466:         if ((strcmp(desired_record, temp_string)) == 0 ) {
467:             // Match found. Use given struct.
468:             free(temp_string);
469:             return_song = song_holder_struct;
470:         } else {
471:             // End of search and no match found. Use NULL.
472:             free(temp_string);
473:             free(return_song);
474:             return_song = NULL;
475:         }
476:
477:     } else { // Still more values to search. Handle accordingly.
478:         int mid_index = (first_index + last_index) / 2;
479:         read_song_of_index(mid_index);
480:         temp_string = to_lower_case(song_holder_struct->song_name);
481:         if ((strcmp(desired_record, temp_string)) == 0 ) {
482:             // Match found. Use given struct.
483:             free(temp_string);
484:             return_song = song_holder_struct;
485:         } else {
486:             if ((strcmp(desired_record, temp_string)) < 0 ) {
487:                 // Current struct song name is higher letter than desired.
488:                 // Searching first half of provided structs.
489:                 free(temp_string);
490:                 return_song = search_array_by_song(return_song, first_index, mid_index,
desired_record);
491:             } else {
492:                 // Current struct song name is lower letter than desired.
493:                 // Searching later half of provided structs.
494:                 free(temp_string);
495:                 return_song = search_array_by_song(return_song, mid_index + 1, last_index,
desired_record);
496:             }
497:         }
498:     }
499:
500:     return return_song;
501: }
```



```
502:
503:
504: /**
505:  * Songs Array is not sorted by album, thus uses basic linear search.
506:  * This is called far less often so that should be acceptable.
507:  *
508:  * Return either valid album or NULL, depending on if record is found or not.
509:  */
510: songs_struct* search_array_by_album(int first_index, int last_index, char* desired_reco
rd) {
511:     int index = 0;
512:
513:     while (index < total_song_count) {
514:         read_song_of_index(index);
515:         char* temp_string = to_lower_case(song_holder_struct->album_name);
516:         if (strcmp(temp_string, desired_record) == 0) {
517:             free(temp_string);
518:             return song_holder_struct;
519:         }
520:         index++;
521:         free(temp_string);
522:     }
523:
524:     return NULL;
525: }
```

```

1:  /* Our own header, to be included after all standard system headers */
2:
3:  #ifndef __ourhdr_h
4:  #define __ourhdr_h
5:
6:  #include      <sys/types.h>    /* required for some of our prototypes */
7:  #include      <stdio.h>         /* for convenience */
8:  #include      <stdlib.h>        /* for convenience */
9:  #include      <string.h>        /* for convenience */
10: #include      <unistd.h>        /* for convenience */
11:
12: #define MAXLINE 4096             /* max line length */
13:
14: #define FILE_MODE      (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
15:                               /* default file access permissions for new file
s */
16: #define DIR_MODE       (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
17:                               /* default permissions for new directories */
18:
19: typedef void    Sigfunc(int);    /* for signal handlers */
20:
21:                               /* 4.3BSD Reno <signal.h> doesn't define SIG_ER
R */
22: #if      defined(SIG_IGN) && !defined(SIG_ERR)
23: #define SIG_ERR ((Sigfunc *)-1)
24: #endif
25:
26: #define min(a,b)      ((a) < (b) ? (a) : (b))
27: #define max(a,b)      ((a) > (b) ? (a) : (b))
28:
29:                               /* prototypes for our own functions */
30: char    *path_alloc(int *);      /* {Prog pathalloc} */
31: int      open_max(void);         /* {Prog openmax} */
32: void     clr_fl(int, int);        /* {Prog setfl} */
33: void     set_fl(int, int);        /* {Prog setfl} */
34: void     pr_exit(int);           /* {Prog prexit} */
35: void     pr_mask(const char *);   /* {Prog prmask} */
36: Sigfunc *signal_intr(int, Sigfunc *); /* {Prog signal_intr_function} */
37:
38: int      tty_cbreak(int);         /* {Prog raw} */
39: int      tty_raw(int);           /* {Prog raw} */
40: int      tty_reset(int);         /* {Prog raw} */
41: void     tty_atexit(void);        /* {Prog raw} */
42: #ifdef ECHO    /* only if <termios.h> has been included */
43: struct termios *tty_termios(void); /* {Prog raw} */
44: #endif
45:
46: void     sleep_us(unsigned int);  /* {Ex sleepus} */
47: ssize_t  readn(int, void *, size_t); /* {Prog readn} */
48: ssize_t  writen(int, const void *, size_t); /* {Prog writen} */
49: int      daemon_init(void);       /* {Prog daemoninit} */
50:
51: int      s_pipe(int *);           /* {Progs svr4_splice bsd_splice}
*/
52: int      recv_fd(int, ssize_t (*func)(int, const void *, size_t));
53:                               /* {Progs recvf
d_svr4 recvfd_43bsd} */
54: int      send_fd(int, int);       /* {Progs sendfd_svr4 sendfd_43
bsd} */
55: int      send_err(int, int, const char *); /* {Prog senderr} */
56: int      serv_listen(const char *); /* {Progs servlisten_svr4 servlisten_44
bsd} */
57: int      serv_accept(int, uid_t *); /* {Progs servaccept_svr4 servaccept_44

```

```

bsd} */
58: int          cli_conn(const char *);          /* {Progs cliconn_svr4 cliconn_44bsd} */
/
59: int          buf_args(char *, int (*func)(int, char **));
60:                                                     /* {Prog bufarg
s} */
61:
62: int          ptym_open(char *);                /* {Progs ptyopen_svr4 ptyopen_
44bsd} */
63: int          ptys_open(int, char *);           /* {Progs ptyopen_svr4 ptyopen_44bsd} */
/
64: #ifndef TIOCGWINSZ
65: pid_t        pty_fork(int *, char *, const struct termios *,
66:                      const struct winsize *);   /* {Prog ptyfork} */
67: #endif
68:
69: int          lock_reg(int, int, int, off_t, int, off_t);
70:                                                     /* {Prog lockre
g} */
71: #define read_lock(fd, offset, whence, len) \
72:         lock_reg(fd, F_SETLK, F_RDLCK, offset, whence, len)
73: #define readw_lock(fd, offset, whence, len) \
74:         lock_reg(fd, F_SETLKW, F_RDLCK, offset, whence, len)
75: #define write_lock(fd, offset, whence, len) \
76:         lock_reg(fd, F_SETLK, F_WRLCK, offset, whence, len)
77: #define writew_lock(fd, offset, whence, len) \
78:         lock_reg(fd, F_SETLKW, F_WRLCK, offset, whence, len)
79: #define un_lock(fd, offset, whence, len) \
80:         lock_reg(fd, F_SETLK, F_UNLCK, offset, whence, len)
81:
82: pid_t        lock_test(int, int, off_t, int, off_t);
83:                                                     /* {Prog lockte
st} */
84:
85: #define is_readlock(fd, offset, whence, len) \
86:         lock_test(fd, F_RDLCK, offset, whence, len)
87: #define is_writelock(fd, offset, whence, len) \
88:         lock_test(fd, F_WRLCK, offset, whence, len)
89:
90: void          err_dump(const char *, ...);      /* {App misc_source} */
91: void          err_msg(const char *, ...);
92: void          err_quit(const char *, ...);
93: void          err_ret(const char *, ...);
94: void          err_sys(const char *, ...);
95:
96: void          log_msg(const char *, ...);        /* {App misc_source} */
97: void          log_open(const char *, int, int);
98: void          log_quit(const char *, ...);
99: void          log_ret(const char *, ...);
100: void          log_sys(const char *, ...);
101:
102: void          TELL_WAIT(void);                  /* parent/child from {Sec race_conditions} */
103: void          TELL_PARENT(pid_t);
104: void          TELL_CHILD(pid_t);
105: void          WAIT_PARENT(void);
106: void          WAIT_CHILD(void);
107:
108: #endif /* __ourhdr_h */

```

```
1: /**
2:  * Brandon Rodriguez
3:  * CS 3240
4:  * 09-19-17
5:  */
6:
7:
8: /**
9:  * A personal header for helper-functions.
10: */
11:
12:
13: // Function prototypes.
14:
15: char* copy_string(char* source_ptr);
16: int* copy_int(int* source_ptr);
17: float* copy_float(float* source_ptr);
18: double* copy_double(double* source_ptr);
19:
20: char* to_lower_case(char* input_string);
21: char* to_upper_case(char* input_string);
22: char* first_letter_upper(char* input_string);
23: char* remove_quotes(char* input_string);
24: char* remove_newline(char* input_string);
```

makefile

Thu Sep 28 15:31:02 2017

1

```
1: all:
2:      gcc -Wall -Wpedantic -std=c99 -g BuildDataBase.c error.c HelperFunctions.c -o B
uildDataBase
3:      gcc -Wall -Wpedantic -std=c99 -g UseDataBase.c error.c HelperFunctions.c -o Use
DataBase
4: build:
5:      ./BuildDataBase
6: use:
7:      ./UseDataBase < test-input.txt
```

```
1: gcc -Wall -Wpedantic -std=c99 -g BuildDataBase.c error.c HelperFunctions.c -o BuildData
Base
2: gcc -Wall -Wpedantic -std=c99 -g UseDataBase.c error.c HelperFunctions.c -o UseDataBase
```

```
1: ./BuildDataBase
2: Reading csv...
3: Sorting arrays...
4: Saving to file...
5: Finished.
6: Closing program...
7: ./UseDataBase < test-input.txt
8: Enter input:
9: Total songs: 10002
10:
11: To display help text again, type 'Help'.
12: To print all songs, type 'All'
13: To exit program, type 'ZZZ'.
14: Otherwise, type name of song you wish to locate.
15:
16: Song Name: We're Not Gonna Bow
17: Album Name: Ordinary Day
18: Artist Name: Jeff And Sheri Easter
19: Duration: 222.93
20: Year: 0
21: Hotttnesss: 0.240821
22:
23:
24: Song Name: Deep Sea Creature
25: Album Name: Call of the Mastodon
26: Artist Name: Mastodon
27: Duration: 280.22
28: Year: 2001
29: Hotttnesss: 0.597641
30:
31:
32: Song Name: Stand By Me
33: Album Name: Made In England
34: Artist Name: Atomic Rooster
35: Duration: 203.31
36: Year: 1972
37: Hotttnesss: 0.000000
38:
39:
40: Song Name: Ralph's Rhapsody
41: Album Name: The Best Of Ray Lynch
42: Artist Name: Ray Lynch
43: Duration: 283.74
44: Year: 1998
45: Hotttnesss: 0.547953
46:
47:
48: Song Name: Don't Mess With the IRS
49: Album Name: Love Death & Taxes
50: Artist Name: Dr. Elmo
51: Duration: 159.32
52: Year: 0
53: Hotttnesss: 0.000000
54:
55:
56: Could not find song with name. Searching by album.
57: (This may take a bit. Album search is slower.)
58: Song Name: Christmas Won't be the Same Without Johnnny
59: Album Name: Dr. Elmo's Twisted Christmas
60: Artist Name: Dr. Elmo
61: Duration: 152.37
62: Year: 0
63: Hotttnesss: 0.000000
```

```
64:
65:
66: Song Name: Too Many Choices
67: Album Name: Personal Business (Explicit)
68: Artist Name: Bad Azz
69: Duration: 285.94
70: Year: 0
71: Hottnesss: 0.405116
72:
73:
74: Song Name: These Days
75: Album Name: Hushabye Baby: Lullaby Renditions of Rascal Flatts
76: Artist Name: Hushabye Baby
77: Duration: 210.86
78: Year: 0
79: Hottnesss: 0.000000
80:
81:
82: Song Name: Armageddon's Raid
83: Album Name: Bondage Goat Zombie
84: Artist Name: Belphegor
85: Duration: 308.77
86: Year: 2008
87: Hottnesss: 0.624840
88:
89:
90: Song Name: Single Ladies (Put A Ring On It)
91: Album Name: Single Ladies (Put A Ring On It) - Dance Remixes
92: Artist Name: Beyonc  
93: Duration: 466.05
94: Year: 2008
95: Hottnesss: 0.000000
96:
97:
98: Song Name: Get On Top (Album Version)
99: Album Name: Californication
100: Artist Name: Red Hot Chili Peppers
101: Duration: 198.06
102: Year: 1999
103: Hottnesss: 0.000000
104:
105:
106: Song Name: Rudeboy
107: Album Name: Aswad vs. The Rhythm Riders
108: Artist Name: Aswad
109: Duration: 258.43
110: Year: 0
111: Hottnesss: 0.000000
112:
113:
114: Exiting Program...
```



```
1: ==4566== Memcheck, a memory error detector
2: ==4566== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
3: ==4566== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
4: ==4566== Command: ./BuildDataBase
5: ==4566==
6: ==4566== Conditional jump or move depends on uninitialised value(s)
7: ==4566==    at 0x401608: sort_array (BuildDataBase.c:357)
8: ==4566==    by 0x400DDB: main (BuildDataBase.c:90)
9: ==4566==
10: ==4566== Conditional jump or move depends on uninitialised value(s)
11: ==4566==    at 0x401608: sort_array (BuildDataBase.c:357)
12: ==4566==    by 0x40163D: sort_array (BuildDataBase.c:368)
13: ==4566==    by 0x400DDB: main (BuildDataBase.c:90)
14: ==4566==
15: ==4566== Conditional jump or move depends on uninitialised value(s)
16: ==4566==    at 0x401608: sort_array (BuildDataBase.c:357)
17: ==4566==    by 0x40163D: sort_array (BuildDataBase.c:368)
18: ==4566==    by 0x40163D: sort_array (BuildDataBase.c:368)
19: ==4566==    by 0x400DDB: main (BuildDataBase.c:90)
20: ==4566==
21: ==4566== Conditional jump or move depends on uninitialised value(s)
22: ==4566==    at 0x401608: sort_array (BuildDataBase.c:357)
23: ==4566==    by 0x40163D: sort_array (BuildDataBase.c:368)
24: ==4566==    by 0x40163D: sort_array (BuildDataBase.c:368)
25: ==4566==    by 0x40163D: sort_array (BuildDataBase.c:368)
26: ==4566==    by 0x400DDB: main (BuildDataBase.c:90)
27: ==4566==
28: ==4566== Conditional jump or move depends on uninitialised value(s)
29: ==4566==    at 0x401804: save_array (BuildDataBase.c:417)
30: ==4566==    by 0x400DF9: main (BuildDataBase.c:93)
31: ==4566==
32: ==4566== Conditional jump or move depends on uninitialised value(s)
33: ==4566==    at 0x401DB7: exit_program (BuildDataBase.c:563)
34: ==4566==    by 0x400E35: main (BuildDataBase.c:97)
35: ==4566==
36: ==4566==
37: ==4566== HEAP SUMMARY:
38: ==4566==    in use at exit: 0 bytes in 0 blocks
39: ==4566==    total heap usage: 280,983 allocs, 280,983 frees, 865,585,217 bytes allocated
40: ==4566==
41: ==4566== All heap blocks were freed -- no leaks are possible
42: ==4566==
43: ==4566== For counts of detected and suppressed errors, rerun with: -v
44: ==4566== Use --track-origins=yes to see where uninitialised values come from
45: ==4566== ERROR SUMMARY: 9964 errors from 6 contexts (suppressed: 0 from 0)
46: ==4569== Memcheck, a memory error detector
47: ==4569== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
48: ==4569== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
49: ==4569== Command: ./UseDataBase
50: ==4569==
51: ==4569==
52: ==4569== HEAP SUMMARY:
53: ==4569==    in use at exit: 0 bytes in 0 blocks
54: ==4569==    total heap usage: 25,627 allocs, 25,627 frees, 26,527,603 bytes allocated
55: ==4569==
56: ==4569== All heap blocks were freed -- no leaks are possible
57: ==4569==
58: ==4569== For counts of detected and suppressed errors, rerun with: -v
59: ==4569== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```