

```
1. /*****
2.  * scramble.c
3.  *
4.  * David J. Malan <malan@harvard.edu>
5.  * Nate Hardison <nate@cs.harvard.edu>
6.  *
7.  * Implements Scramble with CS50.
8.  *
9.  * Usage: scramble [#]
10. *
11. * where # is an optional grid number.
12. *****/
13.
14. #include <cs50.h>
15. #include <ctype.h>
16. #include <libgen.h>
17. #include <stdio.h>
18. #include <string.h>
19. #include <time.h>
20.
21. // duration of a game in seconds
22. #define DURATION 30
23.
24. // grid's dimensions
25. #define DIMENSION 4
26.
27. // maximum number of words in any dictionary
28. #define WORDS 172806
29.
30. // maximum number of letters in any word
31. #define LETTERS 29
32.
33. // default dictionary
34. // http://www.becomeawordgameexpert.com/wordlists.htm
35. #define DICTIONARY "words"
36.
37. // for logging
38. FILE* logfile;
39.
40. // grid
41. char grid[DIMENSION][DIMENSION];
42.
43. // flags with which we can mark grid's letters while searching for words
44. bool marks[DIMENSION][DIMENSION];
45.
46. // defines a word as having an array of letters plus a flag
47. // indicating whether word has been found on grid
48. typedef struct
```

```
49. {
50.     bool found;
51.     char letters[LETTERS + 1];
52. }
53. word;
54.
55. // defines a dictionary as having a size and an array of words
56. struct
57. {
58.     int size;
59.     word words[WORDS];
60. }
61. dictionary;
62.
63. // prototypes
64. void clear(void);
65. bool crawl(string letters, int x, int y);
66. void draw(void);
67. bool find(string s);
68. void initialize(void);
69. bool load(string s);
70. bool lookup(string s);
71. void scramble(void);
72.
73. // This is Scramble.
74. int main(int argc, string argv[])
75. {
76.     // ensure proper usage
77.     if (argc > 2)
78.     {
79.         printf("Usage: %s [#]\n", basename(argv[0]));
80.         return 1;
81.     }
82.
83.     // seed pseudorandom number generator
84.     if (argc == 2)
85.     {
86.         int seed = atoi(argv[1]);
87.         if (seed <= 0)
88.         {
89.             printf("Invalid grid.\n");
90.             return 1;
91.         }
92.         srand(seed);
93.     }
94.     else
95.         srand(time(NULL));
96.
```

```
97. // determine path to dictionary
98. string directory = dirname(argv[0]);
99. char path[strlen(directory) + 1 + strlen(DICTIONARY) + 1];
100. sprintf(path, "%s/%s", directory, DICTIONARY);
101.
102. // load dictionary
103. if (!load(path))
104. {
105.     printf("Could not open dictionary.\n");
106.     return 1;
107. }
108.
109. // initialize the grid
110. initialize();
111.
112. // initialize user's score
113. int score = 0;
114.
115. // calculate time of game's end
116. int end = time(NULL) + DURATION;
117.
118. // open log
119. logfile = fopen("./log.txt", "w");
120. if (logfile == NULL)
121. {
122.     printf("Could not open log.\n");
123.     return 1;
124. }
125.
126. // accept words until timer expires
127. while (true)
128. {
129.     // clear the screen
130.     clear();
131.
132.     // draw the current state of the grid
133.     draw();
134.
135.     // logfile board
136.     for (int row = 0; row < DIMENSION; row++)
137.     {
138.         for (int col = 0; col < DIMENSION; col++)
139.             fprintf(logfile, "%c", grid[row][col]);
140.         fprintf(logfile, "\n");
141.     }
142.
143.     // get current time
144.     int now = time(NULL);
```

```
145.
146.     // report score
147.     printf("Score: %d\n", score);
148.     fprintf(logfile, "%d\n", score);
149.
150.     // check for game's end
151.     if (now >= end)
152.     {
153.         printf("\033[31m"); // red
154.         printf("Time:  %d\n\n", 0);
155.         printf("\033[39m"); // default
156.         break;
157.     }
158.
159.     // report time remaining
160.     printf("Time:  %d\n\n", end - now);
161.
162.     // prompt for word
163.     printf("> ");
164.     string s = GetString();
165.
166.     // quit playing if user hits ctrl-d
167.     if (s == NULL)
168.         break;
169.
170.     // capitalize word
171.     for (int i = 0, n = strlen(s); i < n; i++)
172.         s[i] = toupper(s[i]);
173.
174.     // logfile word
175.     fprintf(logfile, "%s\n", s);
176.
177.     // check whether to scramble grid
178.     if (strcmp(s, "SCRAMBLE") == 0)
179.         scramble();
180.
181.     // or to look for word on grid and in dictionary
182.     else
183.     {
184.         if (find(s) && lookup(s))
185.             score += strlen(s);
186.     }
187. }
188.
189. // close log
190. fclose(logfile);
191.
192. return 0;
```

```
193. }
194.
195. /**
196.  * Clears screen.
197.  */
198. void clear()
199. {
200.     printf("\033[2J");
201.     printf("\033[%d;%dH", 0, 0);
202. }
203.
204. /**
205.  * Crawls grid recursively for letters starting at grid[x][y].
206.  * Returns true iff all letters are found.
207.  */
208. bool crawl(string letters, int x, int y)
209. {
210.     // if out of letters, then we must've found them all!
211.     if (strlen(letters) == 0)
212.         return true;
213.
214.     // don't fall off the grid!
215.     if (x < 0 || x >= DIMENSION)
216.         return false;
217.     if (y < 0 || y >= DIMENSION)
218.         return false;
219.
220.     // been here before!
221.     if (marks[x][y])
222.         return false;
223.
224.     // check grid[x][y] for current letter
225.     if (grid[x][y] != letters[0])
226.         return false;
227.
228.     // mark location
229.     marks[x][y] = true;
230.
231.     // look left and right for next letter
232.     for (int i = -1; i <= 1; i++)
233.     {
234.         // look down and up for next letter
235.         for (int j = -1; j <= 1; j++)
236.         {
237.             // check grid[x + i][y + j] for next letter
238.             if (crawl(&letters[1], x + i, y + j))
239.                 return true;
240.         }
```

```
241.     }
242.
243.     // unmark location
244.     marks[x][y] = false;
245.
246.     // fail
247.     return false;
248. }
249.
250. /**
251.  * Prints the grid in its current state.
252.  */
253. void draw(void)
254. {
255.     printf("\n");
256.     for (int row = 0; row < DIMENSION; row++)
257.     {
258.         printf(" ");
259.         for (int col = 0; col < DIMENSION; col++)
260.         {
261.             printf("%2c", grid[row][col]);
262.         }
263.         printf("\n");
264.     }
265.     printf("\n");
266. }
267.
268. /**
269.  * Returns true iff word, s, is found in grid.
270.  */
271. bool find(string s)
272. {
273.     // word must be at least 2 characters in length
274.     if (strlen(s) < 2)
275.         return false;
276.
277.     // search grid for word
278.     for (int row = 0; row < DIMENSION; row++)
279.     {
280.         for (int col = 0; col < DIMENSION; col++)
281.         {
282.             // reset marks
283.             for (int i = 0; i < DIMENSION; i++)
284.                 for (int j = 0; j < DIMENSION; j++)
285.                     marks[i][j] = false;
286.
287.             // search for word starting at grid[i][j]
288.             if (crawl(s, row, col))
```

```
289.         return true;
290.     }
291. }
292.     return false;
293. }
294.
295. /**
296.  * Initializes grid with letters.
297.  */
298. void initialize(void)
299. {
300.     // http://en.wikipedia.org/wiki/Letter\_frequency
301.     float frequencies[] = {
302.         8.167, // a
303.         1.492, // b
304.         2.782, // c
305.         4.253, // d
306.         12.702, // e
307.         2.228, // f
308.         2.015, // g
309.         6.094, // h
310.         6.966, // i
311.         0.153, // j
312.         0.747, // k
313.         4.025, // l
314.         2.406, // m
315.         6.749, // n
316.         7.507, // o
317.         1.929, // p
318.         0.095, // q
319.         5.987, // r
320.         6.327, // s
321.         9.056, // t
322.         2.758, // u
323.         1.037, // v
324.         2.365, // w
325.         0.150, // x
326.         1.974, // y
327.         0.074 // z
328.     };
329.     int n = sizeof(frequencies) / sizeof(float);
330.
331.     // iterate over grid
332.     for (int row = 0; row < DIMENSION; row++)
333.     {
334.         for (int col = 0; col < DIMENSION; col++)
335.         {
336.             // generate pseudorandom double in [0, 1]
```

```
337.         double d = rand() / (double) RAND_MAX;
338.
339.         // map d onto range of frequencies
340.         for (int k = 0; k < n; k++)
341.         {
342.             d -= frequencies[k] / 100;
343.             if (d < 0.0 || k == n - 1)
344.             {
345.                 grid[row][col] = 'A' + k;
346.                 break;
347.             }
348.         }
349.     }
350. }
351. }
352.
353. /**
354.  * Loads words from dictionary with given filename, s, into a global array.
355.  */
356. bool load(string s)
357. {
358.     // open dictionary
359.     FILE* file = fopen(s, "r");
360.     if (file == NULL)
361.         return false;
362.
363.     // initialize dictionary's size
364.     dictionary.size = 0;
365.
366.     // load words from dictionary
367.     char buffer[LETTERS + 2];
368.     while (fgets(buffer, LETTERS + 2, file))
369.     {
370.         // overwrite \n with \0
371.         buffer[strlen(buffer) - 1] = '\0';
372.
373.         // capitalize word
374.         for (int i = 0, n = strlen(buffer); i < n; i++)
375.             buffer[i] = toupper(buffer[i]);
376.
377.         // ignore SCRAMBLE
378.         if (strcmp(buffer, "SCRAMBLE") == 0)
379.             continue;
380.
381.         // copy word into dictionary
382.         dictionary.words[dictionary.size].found = false;
383.         strncpy(dictionary.words[dictionary.size].letters, buffer, LETTERS + 1);
384.         dictionary.size++;
```



```
385.     }
386.
387.     // success!
388.     return true;
389. }
390.
391. /**
392.  * Looks up word, s, in dictionary.  Iff found (for the first time), flags word
393.  * as found (so that user can't score with it again) and returns true.
394.  */
395. bool lookup(string s)
396. {
397.     int low = 0;
398.     int high = dictionary.size - 1;
399.
400.     while (low <= high)
401.     {
402.         // http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
403.         int mid = ((unsigned int)low + (unsigned int)high) / 2;
404.
405.         // see man page for strcmp for details on its return values!
406.         // make sure to test for >/< 0, not ==/!= 1
407.         int comparison = strcmp(s, dictionary.words[mid].letters);
408.         if (comparison == 0)
409.         {
410.             // check if already found
411.             if (dictionary.words[mid].found)
412.                 return false;
413.
414.             // flag as found
415.             dictionary.words[mid].found = true;
416.
417.             // found it!
418.             return true;
419.         }
420.         else if (comparison > 0)
421.             low = mid + 1;
422.         else
423.             high = mid - 1;
424.     }
425.
426.     // fail
427.     return false;
428. }
429.
430. /**
431.  * Scrambles the grid by rotating it 90 degrees clockwise, whereby grid[0][0]
432.  * rotates to grid[0][DIMENSION - 1]
```

```
433. *
434. * Best to instruct students to draw out all of the cases for a 4x4 grid to
435. * figure out the math below. Trying to do the rotation in-place is a mess,
436. * since moving one cell requires moving three others (e.g. 0,0 -> 0,3 -> 3,0
437. * -> 3,3).
438. */
439. void scramble(void)
440. {
441.     // build up a new grid with the rotation
442.     char rotated_grid[DIMENSION][DIMENSION];
443.     for (int row = 0; row < DIMENSION; row++)
444.     {
445.         for (int col = 0; col < DIMENSION; col++)
446.         {
447.             rotated_grid[col][DIMENSION - row - 1] = grid[row][col];
448.         }
449.     }
450.
451.     // copy the rotated grid into the global grid
452.     for (int row = 0; row < DIMENSION; row++)
453.     {
454.         for (int col = 0; col < DIMENSION; col++)
455.         {
456.             grid[row][col] = rotated_grid[row][col];
457.         }
458.     }
459. }
```