

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <string.h>
5
6  bool is_substring(char *source, char *sub)
7  {
8      int source_len = strlen(source);
9      int sub_len = strlen(sub);
10     int end = (source_len - sub_len) + 1;
11     bool sub_found;
12
13     // go through the source string, one char at a time
14     for(int i = 0; i < end; i++)
15     {
16         sub_found = true;
17         for(int j = 0; j < sub_len; j++)
18         {
19             if(sub[j] != source[i+j])
20             {
21                 sub_found = false;
22                 break;
23             }
24         }
25         if(sub_found) return true;
26     }
27     return false;
28 }
29
30 bool string_contains_char(char *source, char c)
31 {
32     for(int i = 0; i < strlen(source); i++)
33         if(source[i] == c) return true;
34
35     return false;
36 }
37
38 int char_occurences(char *source, char c)
39 {
40     int count = 0;
41     for(int i = 0; i < strlen(source); i++)
42     {
43         if(source[i] == c) count++;
44     }
45     return count;
46 }
47
48 void insert_char(char **string_ptr, int *len, int *capacity, char c)
49 {
50     if( (*len) >= (*capacity) )
51     {
52         (*capacity) *= 2;
53         char *new_addr = realloc((*string_ptr), sizeof(char) * (*capacity));
54         if(new_addr == NULL)
55         {
56             printf("Could not reallocate more memory to insert char '%c'\n", c);
57             return;
58         }
59         (*string_ptr) = new_addr;
60     }
61     (*string_ptr)[(*len)] = c;
62     (*len)++;
63 }
64
65 void concatenate(char **string_ptr, int *len, int *capacity, char *operand)

```

```

66 {
67     int limit = strlen(operand);
68     for(int i = 0; i < limit; i++)
69     {
70         if( (*len) >= (*capacity) )
71         {
72             (*capacity) *= 2;
73             char *new_addr = realloc((*string_ptr), sizeof(char) * (*capacity));
74             if(new_addr == NULL)
75             {
76                 printf("Could not allocate more memory to concatenate strings.\n");
77                 return;
78             }
79             (*string_ptr) = new_addr;
80         }
81         (*string_ptr)[(*len)] = operand[i];
82         (*len)++;
83     }
84 }
85
86
87 // 1. modified to use strdup to stop destruction of source
88 char** split_string(char *source, char *delimiter)
89 {
90     char *source_copy = strdup(source);
91     int len = 0;
92     int capacity = 10;
93     char **string_tokens = malloc(sizeof(char*) * capacity);
94
95     char *token = strtok(source_copy, delimiter);
96     while(token != NULL)
97     {
98         string_tokens[len] = strdup(token);
99         len = len + 1;
100         if(len >= capacity)
101         {
102             capacity *= 2;
103             string_tokens = realloc(string_tokens, sizeof(char*) * capacity);
104         }
105         token = strtok(NULL, delimiter);
106     }
107     // final element is NULL so that you can traverse
108     string_tokens[len] = NULL;
109
110     free(source_copy);
111     return string_tokens;
112 }
113
114 int determine_header_level(char *source)
115 {
116     if(source[0] != '#') return 0;
117     int i = 1;
118     while(source[i] == '#') i++;
119     return i;
120 }
121
122 int determine_code_level(char *source)
123 {
124     if(source[0] != '`') return 0;
125     int i = 1;
126     while(source[i] == '`') i++;
127     return i;
128 }
129
130 int determine_admonition_level(char *source)

```

```

131 {
132     if(source[0] != '!') return 0;
133     int i = 1;
134     while(source[i] == '!') i++;
135     return i;
136 }
137
138 bool is_unordered_list(char *source)
139 {
140     int tab_level = 0;
141     int len = strlen(source);
142
143     // must, at minimum, start with "- a" (a can be any char)
144     if(len < 3) return false;
145
146     // must start with tab or '-'
147     if(source[0] != '-' && source[0] != '\t') return false;
148
149     // case: highest level outer list
150     if(source[0] == '-')
151     {
152         if(source[1] != ' ') return false;
153
154         // since whole file is newline delimited, you dont have to worry about
155         // "- \n" as an argument
156         return true;
157     }
158
159     // case: nested list
160     if(source[0] == '\t')
161     {
162         tab_level = 1;
163         for(int i = tab_level; i < len; i++)
164         {
165             if(source[i] == '\t') tab_level++;
166             else break;
167         }
168         if(source[tab_level] == '-') return true;
169         else return false;
170     }
171 }
172
173 bool is_deflist(char *source)
174 {
175     int len = strlen(source);
176
177     printf("len of \"%s\" is %d\n", source, len);
178
179     // case: end of deflist [true]
180     if(source[0] == '_' && len < 3) return true;
181
182     // case: insufficient length to be deflist beginning
183     if(len < 3) return false;
184
185     if(source[0] != '_') return false;
186
187
188     if(source[1] != ' ') return false;
189
190     // case: start of deflist (if all other cases fail, it is a deflist)
191     return true;
192 }
193
194 // GENERAL NOTE: you must check in this order to avoid jumping to a conclusion:
195 // 1. check if italic bold

```

```
196 //                2. check if bold
197 //                3. check if italic
198 //                otherwise, if you check for italic first, for example, then you will
199 //                determine the line bold to be italic, though it is bold.
200
201 bool is_italic_initiated(char *source)
202 {
203     int len = strlen(source);
204     if(len < 2) return false;
205
206     if(source[0] == '*' && source[1] != '*') return true;
207
208     return false;
209 }
210
211 bool is_italic_terminated(char *source)
212 {
213     int len = strlen(source);
214     if(len < 2) return false;
215
216     if(source[len-1] == '*' && source[len-2] != '*') return true;
217
218     return false;
219 }
220
221 bool is_bold_initiated(char *source)
222 {
223     int len = strlen(source);
224
225     // text that demarcates the beginning or end of bold formatting must be at least 3
226     // characters long, because it must include 2 asterisks that precede or follow
227     // at least one non-asterisk character
228     if(len < 3) return false;
229
230     if(source[0] == '*' && source[1] == '*') return true;
231
232     return false;
233 }
234
235 bool is_bold_terminated(char *source)
236 {
237     int len = strlen(source);
238
239     if(len < 3) return false;
240
241     if(source[len-2] == '*' && source[len-1] == '*') return true;
242
243     return false;
244 }
245
246 bool is_italic_bold_initiated(char *source)
247 {
248     int len = strlen(source);
249     if(len < 4) return false;
250
251     if(source[0] == '*' && source[1] == '*' && source[2] == '*') return true;
252
253     return false;
254 }
255
256 bool is_italic_bold_terminated(char *source)
257 {
258     int len = strlen(source);
259     if(len < 4) return false;
260
```

```

261     if(source[len-1] == '*' && source[len-2] == '*' && source[len-3] == '*') return true;
262
263     return false;
264 }
265
266 char* string_substring(char *source, int inclusive_start, int exclusive_end)
267 {
268     int required_chars = (exclusive_end - inclusive_start) + 1;
269     char *result = malloc(sizeof(char) * required_chars);
270     int j = 0;
271     for(int i = inclusive_start; i < exclusive_end; i++)
272     {
273         result[j] = source[i];
274         j++;
275     }
276     result[j] = '\0';
277     return result;
278 }
279
280 bool flip_boolean(bool b)
281 {
282     if(b) return false;
283     else return true;
284 }
285
286 int main(int argc, char *argv[])
287 {
288     if(argc < 2) return 1;
289
290     FILE *source = fopen(argv[1], "r");
291     if(source == NULL)
292     {
293         printf("Could not open file \"%s\". Does it exist?\n", argv[1]);
294         return 1;
295     }
296
297     // get all contents of the file
298     int len = 0;
299     int capacity = 100;
300     char *contents = malloc(sizeof(char) * capacity);
301     char c;
302     while(true)
303     {
304         c = fgetc(source);
305         if(c == EOF) break;
306
307         contents[len] = c;
308         len = len + 1;
309         if(len >= capacity)
310         {
311             capacity *= 2;
312             contents = realloc(contents, sizeof(char) * capacity);
313         }
314     }
315     contents[len] = '\0';
316     len = len + 1;
317
318
319
320     // tokenize the file (split by new lines)
321     char **string_tokens = split_string(contents, "\n");
322     free(contents);
323     int TEMP_INT = 0; // remove later
324     printf("File contents (delimited by newlines):\n\n");
325     while(string_tokens[TEMP_INT] != NULL)

```

```

326     {
327         printf("line %d: %s\n", TEMP_INT, string_tokens[TEMP_INT]);
328         TEMP_INT++;
329     }
330
331
332
333     // HTML translation
334     int html_len = 0;
335     int html_capacity = 100;
336     char *html = malloc(sizeof(char) * html_capacity);
337     concatenate(&html, &html_len, &html_capacity, "<!DOCTYPE html>\n<html>\n"); // top of
document
338     concatenate(&html, &html_len, &html_capacity, "<head>\n\t <style>\n\t\ttbody
{\n\t\t\tfont-family: Minion Pro Display;\n\t\t}\n.admonition {\nbackground-color:
#f7f7f7;\nmargin-bottom: 10px;\nposition: relative;\noverflow: hidden;\npadding-left: 12px;
/* Added padding to create space between the vertical stripe and the label */\n}\n/* This is
the vertical stripe */\n.admonition:before {\ncontent: \"\";\nposition: absolute;\ntop:
0;\nleft: 0;\nwidth: 6px;\nheight: 100%;\nbackground-color: #ffa500; /* Default vertical
stripe color */\n}\n\n/* Add more custom classes and styles for different variants if
desired */\n\n.admonition h4 {\nmargin: 0;\npadding: 10px 8px;\nfont-size: 18px;\ncolor:
black; /* Adjust the color as needed */\nbackground-color: #f4e7d4; /* Adjust the default
header background color as needed */\nborder-radius: 0 4px 4px 0; /* Added border-radius to
only round the right side */\nmargin-left: -6px;\nmargin-right:
-8px;\n}\n\n.admonition.example h4 {\nbackground-color: #f2edff;\n}\n\n/* format for
admonition variants */\n.admonition.example:before {\nbackground-color:
#7C4DFF;\n}\n\n.admonition.note h4 {\nbackground-color:
#ecf3ff;\n}\n\n.admonition.note:before {\nbackground-color: #448aff;\n}\n\n.admonition.tip
h4 {\nbackground-color: #e5f8f6;\n}\n\n.admonition.tip:before {\nbackground-color:
#00bfa5;\n}\n\n.admonition.success h4 {\nbackground-color:
green;\n}\n\n.admonition.success:before {\nbackground-color:
green;\n}\n\n.admonition.warning h4 {\nbackground-color:
#ffff4e5;\n}\n\n.admonition.warning:before {\nbackground-color:
#ff9100;\n}\n\n.admonition.danger h4 {\nbackground-color:
#ffe7ec;\n}\n\n.admonition.danger:before {\nbackground-color: #ff1744;\n}\n\n.admonition p
{\nmargin-top: 10px;\n}\n</style></head>\n<body>\n");
339
340     // variables for HTML translation
341     int header_level;
342     int code_level;
343     int admonition_level;
344     char *substring;
345     char **split; // used to further split a line into whitespace-delimited tokens. Each token
is then analyzed for
346         // Lower order markers, like ** for bold. A "lower order marking" is one that
does not effect the whole
347         // Line, unlike # for headers and "| " for tables
348     char *admonition_type;
349     char *partially_converted_html;
350     char *joined;
351     bool building_code_block = false;
352     bool building_admonition = false;
353     bool building_unordered_list = false;
354     bool building_deflist = false;
355     bool italic_bold_initiated = false;
356     bool bold_initiated = false;
357     bool italic_initiated = false;
358     bool underline_initiated = false;
359     bool highlight_initiated = false;
360     bool MODIFY_FLAG = false; // used to keep track of whether or not a whitespace-delimited
token from char *split
361         // was modified to replace markdown with HTML (e.g. replacing **
for <b>)
362     int lower_bound;
363     int upper_bound;

```

```

364     int split_index;
365     int preprocessing_token_len;
366     int joined_len;
367     int joined_capacity;
368     int partially_converted_html_len;
369     int partially_converted_html_capacity;
370     int j; // used to iterate through the characters of split[split_index] when replacing
           asterisks with <b> and <i> tags

371
372     // go through all the string_tokens, which are all just individual lines of the file
373     int i = 0;
374     while(string_tokens[i] != NULL)
375     {
376         printf("On line-token \"%s\"\n", string_tokens[i]);
377         // preprocessing -- replace lower order markings, such as *, **, and *** for italic,
           bold, and italic bold
378         // with equivalent HTML tags INLINE
379         split = split_string(string_tokens[i], " ");
380         split_index = 0;

381
382
383         while(split[split_index] != NULL)
384         {
385             preprocessing_token_len = strlen(split[split_index]);
386
387             // converted token
388             partially_converted_html_len = 0;
389             partially_converted_html_capacity = ((5 * preprocessing_token_len) / 8); // resize
           is inevitable, but minimize
390
391             space partially_converted_html = malloc(sizeof(char) * partially_converted_html_capacity);
392
393             // used if any markdown was converted to HTML
394             MODIFY_FLAG = false;
395
396             // DEBUGGING
397             printf("On sub-token \"%s\"\n", split[split_index]);
398
399             // used to go through each individual character of split[split_index]
400             j = 0;
401
402
403             // case: underline (++underline++)
404             // NOTE: this is a single if case, not followed by else-if cases, because there is no
405             // overlap between underline and other cases. The same cannot be said about
           italic, bold,
406             // and italic bold, because they use the same character to delimit (*, **, and
           ***)
407             if(preprocessing_token_len > 2 && is_substring(split[split_index], "+"))
408             {
409                 printf("DEBUG: found token \"%s\" to be part of an underline section.\n", split[
           split_index]);
410                 while(j < preprocessing_token_len - 1)
411                 {
412                     if(split[split_index][j] == '+' && split[split_index][j+1] == '+')
413                     {
414                         MODIFY_FLAG = true;
415                         underline_initiated = flip_boolean(underline_initiated);
416
417                         if(underline_initiated) concatenate(&partially_converted_html, &
           partially_converted_html_len, &partially_converted_html_capacity, "<u>");
418                         else concatenate(&partially_converted_html, &partially_converted_html_len
           , &partially_converted_html_capacity, "</u>");
419

```

```

420         j += 2;
421
422     }
423     else
424     {
425         insert_char(&partially_converted_html, &partially_converted_html_len, &
partially_converted_html_capacity, split[split_index][j]);
426         j++;
427     }
428 }
429 // get whatever characters were missed
430 if(j < preprocessing_token_len && split[split_index][preprocessing_token_len - 1]
!= '+')
431 {
432     insert_char(&partially_converted_html, &partially_converted_html_len, &
partially_converted_html_capacity, split[split_index][preprocessing_token_len
- 1]);
433 }
434 }
435
436
437 // case: highlight (==)
438 // NOTE: because "==" is commonly used in programming languages as the operator for
checking equality,
439 // it must also be the case that a code block is not currently being built for
the <mark> tag
440 // to be added.
441 if((is_substring(split[split_index], "==")) && (building_code_block != true) && (
preprocessing_token_len > 2))
442 {
443     printf("DEBUG: found token \"%s\" to be part of a highlight section.\n", split[
split_index]);
444     while(j < preprocessing_token_len - 1)
445     {
446         if(split[split_index][j] == '=' && split[split_index][j+1] == '=')
447         {
448             MODIFY_FLAG = true;
449             highlight_initiated = flip_boolean(highlight_initiated);
450
451             if(highlight_initiated) concatenate(&partially_converted_html, &
partially_converted_html_len, &partially_converted_html_capacity,
"<mark>");
452             else concatenate(&partially_converted_html, &partially_converted_html_len
, &partially_converted_html_capacity, "</mark>");
453
454             j += 2;
455         }
456     }
457     else
458     {
459         insert_char(&partially_converted_html, &partially_converted_html_len, &
partially_converted_html_capacity, split[split_index][j]);
460         j++;
461     }
462 }
463 // get whatever characters were missed
464 if(j < preprocessing_token_len && split[split_index][preprocessing_token_len - 1]
!= '=')
465 {
466     insert_char(&partially_converted_html, &partially_converted_html_len, &
partially_converted_html_capacity, split[split_index][preprocessing_token_len
- 1]);
467 }
468 }
469

```



```

470
471 // case: italic bold
472 if(preprocessing_token_len > 3 && is_substring(split[split_index], "***"))
473 {
474     printf("DEBUG: found token \"%s\" to be part of an italic bold section.\n", split
[split_index]);
475     // the idea is to copy character by character until we reach the delimiter. If
we reach the delimiter,
476     // then do not copy the character. Instead, copy <b><i> or </i></b> depending on
whether we have already
477     // opened a <b><i> tag-pair. This is indicated by the italic_bold_initiated
variable. Then, skip ahead
478     // 3 characters so that we skip the first, second, and third asterisk.
479     while(j < preprocessing_token_len - 2)
480     {
481         if(split[split_index][j] == '*' && split[split_index][j+1] == '*' && split[
split_index][j+2] == '*')
482         {
483             MODIFY_FLAG = true;
484             italic_bold_initiated= flip_boolean(italic_bold_initiated);
485
486             if(italic_bold_initiated) concatenate(&partially_converted_html, &
partially_converted_html_len, &partially_converted_html_capacity,
"<b><i>");
487             else concatenate(&partially_converted_html, &partially_converted_html_len
, &partially_converted_html_capacity, "</i></b>");
488
489             j += 3;
490         }
491         else
492         {
493             insert_char(&partially_converted_html, &partially_converted_html_len, &
partially_converted_html_capacity, split[split_index][j]);
494             j++;
495         }
496     }
497     // get whatever characters were missed
498     if(j < preprocessing_token_len)
499     {
500         while(j < preprocessing_token_len)
501         {
502             insert_char(&partially_converted_html, &partially_converted_html_len, &
partially_converted_html_capacity, split[split_index][j]);
503             j++;
504         }
505     }
506 }
507
508 // case: bold
509 else if(preprocessing_token_len > 2 && is_substring(split[split_index], "***"))
510 {
511     printf("DEBUG: found token \"%s\" to be part of a bold section.\n", split[
split_index]);
512     while(j < preprocessing_token_len - 1)
513     {
514         if(split[split_index][j] == '*' && split[split_index][j+1] == '*')
515         {
516             MODIFY_FLAG = true;
517             bold_initiated = flip_boolean(bold_initiated);
518
519             if(bold_initiated) concatenate(&partially_converted_html, &
partially_converted_html_len, &partially_converted_html_capacity, "<b>");
520             else concatenate(&partially_converted_html, &partially_converted_html_len
, &partially_converted_html_capacity, "</b>");
521

```

```

522         j += 2;
523
524     }
525     else
526     {
527         insert_char(&partially_converted_html, &partially_converted_html_len, &
                    partially_converted_html_capacity, split[split_index][j]);
528         j++;
529     }
530 }
531 // get whatever characters were missed
532 if(j < preprocessing_token_len && split[split_index][preprocessing_token_len - 1]
    != '*')
533 {
534     insert_char(&partially_converted_html, &partially_converted_html_len, &
                partially_converted_html_capacity, split[split_index][preprocessing_token_len
                    - 1]);
535 }
536 }
537
538 // case: italic
539 else if(preprocessing_token_len > 1 && string_contains_char(split[split_index], '*'))
540 {
541     printf("DEBUG: found token \"%s\" to be part of an italic section.\n", split[
        split_index]);
542     printf("DEBUG [ITALIC]: value of j is %d and value of preprocessing_token_len is
        %d\n", j, preprocessing_token_len);
543     while(j < preprocessing_token_len)
544     {
545         if(split[split_index][j] == '*')
546         {
547             MODIFY_FLAG = true;
548             italic_initiated = flip_boolean(italic_initiated);
549
550             if(italic_initiated) concatenate(&partially_converted_html, &
                partially_converted_html_len, &partially_converted_html_capacity, "<i>");
551             else concatenate(&partially_converted_html, &partially_converted_html_len
                , &partially_converted_html_capacity, "</i>");
552         }
553         else
554         {
555             insert_char(&partially_converted_html, &partially_converted_html_len, &
                partially_converted_html_capacity, split[split_index][j]);
556         }
557         j++;
558     }
559     // unlike italic-bold and bold, there is no need to try to get any missing
        characters. There should not
560     // be characters that were missing.
561 }
562
563 // do nothing
564 else
565 {
566 }
567
568 if(MODIFY_FLAG)
569 {
570     insert_char(&partially_converted_html, &partially_converted_html_len, &
                partially_converted_html_capacity, '\0');
571     free(split[split_index]);
572     split[split_index] = strdup(partially_converted_html);
573     free(partially_converted_html);
574 }
575

```

```

576         split_index++;
577     }
578
579     // setup to join the split line
580     split_index = 0;
581     joined_len = 0;
582     joined_capacity = 50;
583     joined = malloc(sizeof(char) * joined_capacity);
584
585     // join the split line -- the joined version will have all the lower-order HTML tags
586     // - make sure to free memory while at it
587     while(split[split_index] != NULL)
588     {
589         concatenate(&joined, &joined_len, &joined_capacity, split[split_index]);
590         free(split[split_index]);
591         insert_char(&joined, &joined_len, &joined_capacity, ' ');
592         split_index++;
593     }
594     joined[joined_len] = '\0';
595
596     // free memory and move joined split line into string_tokens[i]
597     free(split);
598     free(string_tokens[i]);
599     string_tokens[i] = strdup(joined);
600     free(joined);
601
602     printf("DEBUG: Found line \"%s\" to be %d for call to is_deflist\n", string_tokens[i],
603           is_deflist(string_tokens[i]));
604     // case: header
605     // ----- do not check for bold or italic -- no bold or italic permitted in
606     headers
607     if(string_tokens[i][0] == '#')
608     {
609         // determine level of header
610         header_level = determine_header_level(string_tokens[i]);
611         printf("Header level found to be %d for line \"%s\"\n", header_level, string_tokens[i]);
612         while(string_tokens[i][header_level] == '#')
613         {
614             header_level++;
615         }
616
617         // add header tag
618         concatenate(&html, &html_len, &html_capacity, "<h");
619         insert_char(&html, &html_len, &html_capacity, header_level + '0');
620         insert_char(&html, &html_len, &html_capacity, '>');
621
622         // add header contents -- this requires the removal of the # characters
623         substring = string_substring(string_tokens[i], header_level+1, strlen(string_tokens[i]));
624         concatenate(&html, &html_len, &html_capacity, substring);
625         free(substring);
626
627         // end the tag
628         concatenate(&html, &html_len, &html_capacity, "</h");
629         insert_char(&html, &html_len, &html_capacity, header_level + '0');
630         concatenate(&html, &html_len, &html_capacity, ">");
631         header_level = 0;
632     }
633
634     // case: code block
635     // ----- do not check for bold or italic -- no bold or italic permitted in
636     headers
637     else if(string_tokens[i][0] == '`')
638     {

```

```

636     code_level = determine_code_level(string_tokens[i]);
637     if(code_level == 3)
638     {
639         printf("Determined line \"%s\" to be a code block delimiter, but nothing was
        added to the HTML, because this feature is not implemented yet!\n", string_tokens
        [i]);
640         building_code_block = flip_boolean(building_code_block);
641     }
642     if(building_code_block) concatenate(&html, &html_len, &html_capacity, "<code>");
643     else concatenate(&html, &html_len, &html_capacity, "</code>");
644 }
645
646 // case: admonition
647 else if(string_tokens[i][0] == '!')
648 {
649     admonition_level = determine_admonition_level(string_tokens[i]);
650     if(admonition_level == 3)
651     {
652         printf("DEBUG: Determined line \"%s\" to be an admonition delimiter, but nothing
        was added to the HTML, because this feature is not implemented yet!\n",
        string_tokens[i]);
653         // upon reaching admonition beginning, invert value of building_admonition so as
        to indicate when
654         // we have stopped and started building an admonition. The start and stop of an
        admonition looks the same
655         // except that the start has more text after it.
656         building_admonition = flip_boolean(building_admonition);
657     }
658     printf("%s\n", string_tokens[i]);
659
660     // get the type of admonition -- remember to free this later
661     split = split_string(string_tokens[i], " ");
662     admonition_type = split[1];
663     printf("DEBUG: string_tokens[i] after split to get admonition type: \"%s\"\n",
        string_tokens[i]);
664
665     if(building_admonition)
666     {
667         // create div tag including the admonition class and the subclass (e.g.
        admonition note)
668         concatenate(&html, &html_len, &html_capacity, "<div class=\"admonition ");
669         concatenate(&html, &html_len, &html_capacity, admonition_type);
670         concatenate(&html, &html_len, &html_capacity, ">\n");
671
672         // add the admonition type in the form of h4 (e.g. <h4>example<h4>)
673         concatenate(&html, &html_len, &html_capacity, "<h4>");
674         concatenate(&html, &html_len, &html_capacity, admonition_type);
675
676         // add the rest of the title -- this is just everything after the "!!! example "
        or "!!! note " part
677         lower_bound = 4 + strlen(admonition_type); // The +4 skips over the "!!! " part
678                                                     // and the strlen(admonition_type)
679                                                     // skips over the "example part"
680                                                     // of the "!!! example ". finally
681                                                     // since index is 1 smaller, the space
682                                                     // after "example" is ignored
683         upper_bound = strlen(string_tokens[i]);
684         printf("DEBUG: found lower bound = %d and upper bound = %d for line \"%s\"\n",
        lower_bound, upper_bound, string_tokens[i]);
685         if(upper_bound > lower_bound)
686         {
687             // add the rest of the title
688             substring = string_substring(string_tokens[i], 4 + strlen(admonition_type),
        strlen(string_tokens[i])); // the rest of the title
689             insert_char(&html, &html_len, &html_capacity, ':'); // since default C (no

```

```

688         unicode -- I cannot figure out how to get it to work
689                                     // on my system with
690                                     unicode), write it like
691                                     "example: How to parse
692                                     markdown"
693                                     // instead of with an
694                                     emoji preceding the title
695
696         concatenate(&html, &html_len, &html_capacity, substring);
697
698         // free memory
699         free(substring);
700         free(split[0]);
701         split_index = 2;
702         while(split[split_index] != NULL)
703         {
704             free(split[split_index]);
705             split_index++;
706         }
707         free(split);
708     }
709     free(admonition_type);
710
711     // close the h4 label tag
712     concatenate(&html, &html_len, &html_capacity, "</h4>");
713 }
714 else concatenate(&html, &html_len, &html_capacity, "</div>");
715 }
716
717 // case: bulleted list
718 else if(is_unordered_list(string_tokens[i]))
719 {
720     building_unordered_list = flip_boolean(building_unordered_list);
721
722     // add the <ul> tag
723     if(building_unordered_list) concatenate(&html, &html_len, &html_capacity, "<ul>");
724     else concatenate(&html, &html_len, &html_capacity, "</ul>");
725 }
726
727 // case: deflist -- deflist is a little different here because it requires
728 //                 '_' to be the first character as its delimiter, and it
729 //                 must be followed by a space
730 else if(is_deflist(string_tokens[i]))
731 {
732     // create <dl> opening or closing tag
733     building_deflist = flip_boolean(building_deflist);
734     if(building_deflist)
735     {
736         // if the deflist has just begun, then you have to
737         // add the term as well. Deflists here are not really lists at all.
738         // Every deflist generated by this code is just a single term.
739         concatenate(&html, &html_len, &html_capacity, "<dl>");
740         concatenate(&html, &html_len, &html_capacity, "\n<dt>");
741         substring = string_substring(string_tokens[i], 2, strlen(string_tokens[i]));
742         concatenate(&html, &html_len, &html_capacity, substring);
743         free(substring);
744         concatenate(&html, &html_len, &html_capacity, "</dt>");
745     }
746     else concatenate(&html, &html_len, &html_capacity, "</dl>");
747 }
748
749 // case: paragraph
750 else
751 {
752     // whenever not building a list, just add a simple paragraph <p> tag

```

```

748     if( (!building_unordered_list) && (!building_deflist) )
749     {
750         concatenate(&html, &html_len, &html_capacity, "<p>");
751         concatenate(&html, &html_len, &html_capacity, string_tokens[i]);
752         concatenate(&html, &html_len, &html_capacity, "</p>");
753     }
754     else if(building_unordered_list)
755     {
756         concatenate(&html, &html_len, &html_capacity, "<li>");
757         concatenate(&html, &html_len, &html_capacity, string_tokens[i]);
758         concatenate(&html, &html_len, &html_capacity, "</li>");
759     }
760
761     else if(building_deflist)
762     {
763         concatenate(&html, &html_len, &html_capacity, "\t<dd>");
764         concatenate(&html, &html_len, &html_capacity, string_tokens[i]);
765         concatenate(&html, &html_len, &html_capacity, "</dd>");
766     }
767     // impossible case
768     else
769     {
770         printf("Now how did this happen?\n");
771     }
772 }
773
774 // go to next token
775 insert_char(&html, &html_len, &html_capacity, '\n');
776 i++;
777 } // end of while
778
779
780
781 // terminate and display HTML
782 concatenate(&html, &html_len, &html_capacity, "\n</body>\n</html>");
783 insert_char(&html, &html_len, &html_capacity, '\0');
784 printf("len %d capacity %d\n", html_len, html_capacity);
785 printf("%s", html);
786
787
788
789 // cleanup
790 for(int j = 0; j < i; j++)
791     free(string_tokens[j]);
792
793 free(string_tokens);
794 return 0;
795
796 }
797

```