

```

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_occurrences(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");
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.n.admonition.example h4 {\nbackground-color: #f2edff;\n}\n\n/* format for
admonition variants */\n.n.admonition.example:before {\nbackground-color:
#7C4dff;\n}\n\n.n.admonition.note h4 {\nbackground-color:
#ecf3ff;\n}\n\n.n.admonition.note:before {\nbackground-color: #448aff;\n}\n\n.n.admonition.tip
h4 {\nbackground-color: #e5f8f6;\n}\n\n.n.admonition.tip:before {\nbackground-color:
#00bfa5;\n}\n\n.n.admonition.success h4 {\nbackground-color:
green;\n}\n\n.n.admonition.success:before {\nbackground-color:
green;\n}\n\n.n.admonition.warning h4 {\nbackground-color:
##fff4e5;\n}\n\n.n.admonition.warning:before {\nbackground-color:
#ff9100;\n}\n\n.n.admonition.danger h4 {\nbackground-color:
#ffe7ec;\n}\n\n.n.admonition.danger:before {\nbackground-color: #ff1744;\n}\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
346 // is then analyzed for
347 // lower order markers, like ** for bold. A "lower order marking" is one that
348 // does not effect the whole
349 // line, unlike # for headers and "| " for tables
350 char *admonition_type;
351 char *partiallyConverted_html;
352 char *joined;
353 bool building_code_block = false;
354 bool building_admonition = false;
355 bool building_unordered_list = false;
356 bool building_deflist = false;
357 bool italic_bold_initiated = false;
358 bool bold_initiated = false;
359 bool italic_initiated = false;
360 bool underline_initiated = false;
361 bool highlight_initiated = false;
362 bool MODIFY_FLAG = false; // used to keep track of whether or not a whitespace-delimited
363 token from char *split
364 // was modified to replace markdown with HTML (e.g. replacing ** for <b>)
365 int lower_bound;
366 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_token] when replacing
371     asterisks with <b> and <i> tags
372
373     // go through all the string_tokens, which are all just individual lines of the file
374     int i = 0;
375     while(string_tokens[i] != NULL)
376     {
377         printf("On line-token \"%s\"\n", string_tokens[i]);
378         // preprocessing -- replace lower order markings, such as *, **, and *** for italic,
379         // bold, and italic bold
380         // with equivalent HTML tags INLINE
381         split = split_string(string_tokens[i], " ");
382         split_index = 0;
383
384         while(split[split_index] != NULL)
385         {
386             preprocessing_token_len = strlen(split[split_index]);
387
388             // converted token
389             partially_converted_html_len = 0;
390             partially_converted_html_capacity = ((5 * preprocessing_token_len) / 8); // resize
391             // is inevitable, but minimize
392             // wasted
393             space
394             partially_converted_html = malloc(sizeof(char) * partially_converted_html_capacity);
395
396             // used if any markdown was converted to HTML
397             MODIFY_FLAG = false;
398
399             // DEBUGGING
400             printf("On sub-token \"%s\"\n", split[split_index]);
401
402             // used to go through each individual character of split[split_index]
403             j = 0;
404
405             // case: underline (++underline++)
406             // NOTE: this is a single if case, not followed by else-if cases, because there is no
407             // overlap between underline and other cases. The same cannot be said about
408             italic, bold,
409             // and italic bold, because they use the same character to delimit (*, **, and
410             ***)
411             if(preprocessing_token_len > 2 && is_substring(split[split_index], "++"))
412             {
413                 printf("DEBUG: found token \"%s\" to be part of an underline section.\n", split[
414                     split_index]);
415                 while(j < preprocessing_token_len - 1)
416                 {
417                     if(split[split_index][j] == '+' && split[split_index][j+1] == '+')
418                     {
419                         MODIFY_FLAG = true;
420                         underline_initiated = flip_boolean(underline_initiated);
421
422                         if(underline_initiated) concatenate(&partially_converted_html, &
423                             partially_converted_html_len, &partially_converted_html_capacity, "<u>");
424                         else concatenate(&partially_converted_html, &partially_converted_html_len,
425                             &partially_converted_html_capacity, "</u>");
426
427                     }
428                 }
429             }
430         }
431     }

```

```

420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
    j += 2;
}
else
{
    insert_char(&partially_converted_html, &partially_converted_html_len, &
    partially_converted_html_capacity, split[split_index][j]);
    j++;
}
// get whatever characters were missed
if(j < preprocessing_token_len && split[split_index][preprocessing_token_len - 1]
!= '+')
{
    insert_char(&partially_converted_html, &partially_converted_html_len, &
    partially_converted_html_capacity, split[split_index][preprocessing_token_len
- 1]);
}
}

// case: highlight (==)
// NOTE: because "==" is commonly used in programming Languages as the operator for
// checking equality,
//       it must also be the case that a code block is not currently being built for
the <mark> tag
//       to be added.
if((is_substring(split[split_index], "==")) && (building_code_block != true) && (
preprocessing_token_len > 2))
{
    printf("DEBUG: found token \"%s\" to be part of a highlight section.\n", split[
    split_index]);
    while(j < preprocessing_token_len - 1)
    {
        if(split[split_index][j] == '=' && split[split_index][j+1] == '=')
        {
            MODIFY_FLAG = true;
            highlight_initiated = flip_boolean(highlight_initiated);

            if(highlight_initiated) concatenate(&partially_converted_html, &
            partially_converted_html_len, &partially_converted_html_capacity,
            "<mark>");
            else concatenate(&partially_converted_html, &partially_converted_html_len
            , &partially_converted_html_capacity, "</mark>");

            j += 2;
        }
        else
        {
            insert_char(&partially_converted_html, &partially_converted_html_len, &
            partially_converted_html_capacity, split[split_index][j]);
            j++;
        }
    }
    // get whatever characters were missed
    if(j < preprocessing_token_len && split[split_index][preprocessing_token_len - 1]
!= '+')
    {
        insert_char(&partially_converted_html, &partially_converted_html_len, &
        partially_converted_html_capacity, split[split_index][preprocessing_token_len
- 1]);
    }
}

```

```

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
476     // we reach the delimiter,
477     // then do not copy the character. Instead, copy <b><i> or </i></b> depending on
478     // whether we have already
479     // opened a <b><i> tag-pair. This is indicated by the italic_bold_initiated
480     // variable. Then, skip ahead
481     // 3 characters so that we skip the first, second, and third asterisk.
482     while(j < preprocessing_token_len - 2)
483     {
484         if(split[split_index][j] == '*' && split[split_index][j+1] == '*' && split[split_index][j+2] == '*')
485         {
486             MODIFY_FLAG = true;
487             italic_bold_initiated= flip_boolean(italic_bold_initiated);
488
489             if(italic_bold_initiated) concatenate(&partiallyConverted_html, &
490             partiallyConverted_html_len, &partiallyConverted_html_capacity,
491             "<b><i>");
492             else concatenate(&partiallyConverted_html, &partiallyConverted_html_len,
493             , &partiallyConverted_html_capacity, "</i></b>");
494
495             j += 3;
496         }
497         else
498         {
499             insert_char(&partiallyConverted_html, &partiallyConverted_html_len, &
500             partiallyConverted_html_capacity, split[split_index][j]);
501             j++;
502         }
503     }
504
505     // get whatever characters were missed
506     if(j < preprocessing_token_len)
507     {
508         while(j < preprocessing_token_len)
509         {
510             insert_char(&partiallyConverted_html, &partiallyConverted_html_len, &
511             partiallyConverted_html_capacity, split[split_index][j]);
512             j++;
513         }
514     }
515
516     // case: bold
517     else if(preprocessing_token_len > 2 && is_substring(split[split_index], "***"))
518     {
519         printf("DEBUG: found token \"%s\" to be part of a bold section.\n", split[split_index]);
520         while(j < preprocessing_token_len - 1)
521         {
522             if(split[split_index][j] == '*' && split[split_index][j+1] == '*')
523             {
524                 MODIFY_FLAG = true;
525                 bold_initiated = flip_boolean(bold_initiated);
526
527                 if(bold_initiated) concatenate(&partiallyConverted_html, &
528                 partiallyConverted_html_len, &partiallyConverted_html_capacity, "<b>");
529                 else concatenate(&partiallyConverted_html, &partiallyConverted_html_len,
530                 , &partiallyConverted_html_capacity, "</b>");
531             }
532         }
533     }

```

```

522           j += 2;
523
524       }
525       else
526       {
527           insert_char(&partially_converted_html, &partially_converted_html_len, &
528           partially_converted_html_capacity, split[split_index][j]);
529           j++;
530       }
531   // get whatever characters were missed
532   if(j < preprocessing_token_len && split[split_index][preprocessing_token_len - 1]
533     != '*')
534   {
535       insert_char(&partially_converted_html, &partially_converted_html_len, &
536       partially_converted_html_capacity, split[split_index][preprocessing_token_len
537         - 1]);
538   }
539
540   // case: italic
541   else if(preprocessing_token_len > 1 && string_contains_char(split[split_index], '*'))
542   {
543       printf("DEBUG: found token \"%s\" to be part of an italic section.\n", split[
544         split_index]);
545       printf("DEBUG [ITALIC]: value of j is %d and value of preprocessing_token_len is
546         %d\n", j, preprocessing_token_len);
547       while(j < preprocessing_token_len)
548       {
549           if(split[split_index][j] == '*')
550           {
551               MODIFY_FLAG = true;
552               italic_initiated = flip_boolean(italic_initiated);
553
554               if(italic_initiated) concatenate(&partially_converted_html, &
555                 partially_converted_html_len, &partially_converted_html_capacity, "<i>");
556               else concatenate(&partially_converted_html, &partially_converted_html_len
557                 , &partially_converted_html_capacity, "</i>");
558           }
559           else
560           {
561               insert_char(&partially_converted_html, &partially_converted_html_len, &
562                 partially_converted_html_capacity, split[split_index][j]);
563           }
564           j++;
565       }
566   // unlike italic-bold and bold, there is no need to try to get any missing
567   // characters. There should not
568   // be characters that were missing.
569   }
570
571   // do nothing
572   else
573   {
574       if(MODIFY_FLAG)
575       {
576           insert_char(&partially_converted_html, &partially_converted_html_len, &
577             partially_converted_html_capacity, '\0');
578           free(split[split_index]);
579           split[split_index] = strdup(partially_converted_html);
580           free(partially_converted_html);
581       }

```

```

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     );
613     while(string_tokens[i][header_level] == '#')
614     {
615         header_level++;
616     }
617
618     // add header tag
619     concatenate(&html, &html_len, &html_capacity, "<h");
620     insert_char(&html, &html_len, &html_capacity, header_level + '0');
621     insert_char(&html, &html_len, &html_capacity, '>');
622
623     // add header contents -- this requires the removal of the # characters
624     substring = string_substring(string_tokens[i], header_level+1, strlen(string_tokens[i]
625     ));
626     concatenate(&html, &html_len, &html_capacity, substring);
627     free(substring);
628
629     // end the tag
630     concatenate(&html, &html_len, &html_capacity, "</h");
631     insert_char(&html, &html_len, &html_capacity, header_level + '0');
632     concatenate(&html, &html_len, &html_capacity, ">");
633     header_level = 0;
634 }
635
636 // case: code block
637 // ----- do not check for bold or italic -- no bold or italic permitted in
638 // headers
639 else if(string_tokens[i][0] == '`')
640 {

```

```

636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
999

```

688

*unicode -- I cannot figure out how to get it to work*

689

*// on my system with  
unicode), write it like  
"example: How to parse  
markdown"  
// instead of with an  
emoji preceding the title*

690

concatenate(&amp;html, &amp;html\_len, &amp;html\_capacity, substring);

691

*// free memory*

free(substring);

free(split[0]);

split\_index = 2;

while(split[split\_index] != NULL)

{

free(split[split\_index]);

split\_index++;

}

free(split);

}

free(admonition\_type);

705

*// close the h4 Label tag*

concatenate(&amp;html, &amp;html\_len, &amp;html\_capacity, "&lt;/h4&gt;");

}

*else concatenate(&html, &html\_len, &html\_capacity, "</div>");*

}

*// case: bulleted list**else if(is\_unordered\_list(string\_tokens[i]))*

{

building\_unordered\_list = flip\_boolean(building\_unordered\_list);

*// add the <ul> tag*if(building\_unordered\_list) concatenate(&html, &html\_len, &html\_capacity, "<ul>");  
else concatenate(&html, &html\_len, &html\_capacity, "</ul>");

}

*// case: deflist -- deflist is a little different here because it requires**'\_' to be the first character as its delimiter, and it**must be followed by a space**else if(is\_deflist(string\_tokens[i]))*

{

*// create <dl> opening or closing tag*

building\_deflist = flip\_boolean(building\_deflist);

if(building\_deflist)

{

*// if the deflist has just begun, then you have to**// add the term as well. Deflists here are not really lists at all.**// Every deflist generated by this code is just a single term.*

concatenate(&amp;html, &amp;html\_len, &amp;html\_capacity, "&lt;dl&gt;");

concatenate(&amp;html, &amp;html\_len, &amp;html\_capacity, "\n\t&lt;dt&gt;");

substring = string\_substring(string\_tokens[i], 2, strlen(string\_tokens[i]));

concatenate(&amp;html, &amp;html\_len, &amp;html\_capacity, substring);

free(substring);

concatenate(&amp;html, &amp;html\_len, &amp;html\_capacity, "&lt;/dt&gt;");

}

else concatenate(&amp;html, &amp;html\_len, &amp;html\_capacity, "&lt;/dl&gt;");

}

*// case: paragraph**else*

{

*// whenever not building a list, just add a simple paragraph <p> tag*

743

744

745

746

747

```

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

```