```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

bool is_substring(char *source, char *sub)
{
    int source_len = strlen(source);
    int sub_len = strlen(sub);
    int end = (source_len - sub_len) + 1;
    bool sub_found;

    // go through the source string, one char at a time
    for(int i = 0; i < end; i++)
    {
        sub_found = true;
        for(int j = 0; j < sub_len; j++)
        {
            if(sub[j] != source[i+j])
            {
                sub_found = false;
                break;
            }
        }
        if(sub_found) return true;
    }
    return false;
}

bool string_contains_char(char *source, char c)
{
    for(int i = 0; i < strlen(source); i++)
        if(source[i] == c) return true;

    return false;
}

int char_occurences(char *source, char c)
{
    int count = 0;
    for(int i = 0; i < strlen(source); i++)
    {
        if(source[i] == c) count++;
    }
    return count;
}

void insert_char(char **string_ptr, int *len, int *capacity, char c)
{
    if( (*len) >= (*capacity) )
    {
        (*capacity) *= 2;
        char *new_addr = realloc((*string_ptr), sizeof(char) * (*capacity));
        if(new_addr == NULL)
        {
            printf("Could not reallocate more memory to insert char \'%c\'\n", c);
            return;
        }
        (*string_ptr) = new_addr;
    }
    (*string_ptr)[(*len)] = c;
    (*len)++;
}

void concatenate(char **string_ptr, int *len, int *capacity, char *operand)
```

```c
{
    int limit = strlen(operand);
    for(int i = 0; i < limit; i++)
    {
        if( (*len) >= (*capacity) )
        {
            (*capacity) *= 2;
            char *new_addr = realloc((*string_ptr), sizeof(char) * (*capacity));
            if(new_addr == NULL)
            {
                printf("Could not allocate more memory to concatenate strings.\n");
                return;

            }
            (*string_ptr) = new_addr;
        }
        (*string_ptr)[(*len)] = operand[i];
        (*len)++;
    }
}

// 1. modified to use strdup to stop destruction of source
char** split_string(char *source, char *delimiter)
{
    char *source_copy = strdup(source);
    int len = 0;
    int capacity = 10;
    char **string_tokens = malloc(sizeof(char*) * capacity);

    char *token = strtok(source_copy, delimiter);
    while(token != NULL)
    {
        string_tokens[len] = strdup(token);
        len = len + 1;
        if(len >= capacity)
        {
            capacity *= 2;
            string_tokens = realloc(string_tokens, sizeof(char*) * capacity);
        }
        token = strtok(NULL, delimiter);
    }
    // final element is NULL so that you can traverse
    string_tokens[len] = NULL;

    free(source_copy);
    return string_tokens;
}

int determine_header_level(char *source)
{
    if(source[0] != '#') return 0;
    int i = 1;
    while(source[i] == '#') i++;
    return i;
}

int determine_code_level(char *source)
{
    if(source[0] != '`') return 0;
    int i = 1;
    while(source[i] == '`') i++;
    return i;
}

int determine_admoninition_level(char *source)
```

```c
{
    if(source[0] != '!') return 0;
    int i = 1;
    while(source[i] == '!') i++;
    return i;
}

bool is_unordered_list(char *source)
{
    int tab_level = 0;
    int len = strlen(source);

    // must, at minimum, start with "- a" (a can be any char)
    if(len < 3) return false;

    // must start with tab or '-'
    if(source[0] != '-' && source[0] != '\t') return false;

    // case: highest level outer list
    if(source[0] == '-')
    {
        if(source[1] != ' ') return false;

        // since whole file is newline delimited, you dont have to worry about
        // "- \n" as an argument
        return true;
    }

    // case: nested list
    if(source[0] == '\t')
    {
        tab_level = 1;
        for(int i = tab_level; i < len; i++)
        {
            if(source[i] == '\t') tab_level++;
            else break;
        }
        if(source[tab_level] == '-') return true;
        else return false;
    }
}

bool is_deflist(char *source)
{
    int len = strlen(source);

    printf("len of \"%s\" is %d\n", source, len);

    // case: end of deflist [true]
    if(source[0] == '_' && len < 3) return true;

    // case: insufficient length to be deflist beginning
    if(len < 3) return false;

    if(source[0] != '_') return false;


    if(source[1] != ' ') return false;

    // case: start of deflist (if all other cases fail, it is a deflist)
    return true;
}

// GENERAL NOTE: you must check in this order to avoid jumping to a conclusion:
//                 1. check if italic bold
//                 2. check if bold
//                 3. check if italic
//                 otherwise, if you check for italic first, for example, then you will
//                 determine the line **bold** to be italic, though it is bold.

bool is_italic_initiated(char *source)
{
    int len = strlen(source);
    if(len < 2) return false;

    if(source[0] == '*' && source[1] != '*') return true;

    return false;
}

bool is_italic_terminated(char *source)
{
    int len = strlen(source);
    if(len < 2) return false;

    if(source[len-1] == '*' && source[len-2] != '*') return true;

    return false;
}

bool is_bold_initiated(char *source)
{
    int len = strlen(source);

    // text that demarcates the beginning or end of bold formatting must be at least 3
    // characters long, because it must include 2 asterisks that precede or follow
    // at least one non-asterisk character
    if(len < 3) return false;

    if(source[0] == '*' && source[1] == '*') return true;

    return false;
}

bool is_bold_terminated(char *source)
{
    int len = strlen(source);

    if(len < 3) return false;

    if(source[len-2] == '*' && source[len-1] == '*') return true;

    return false;
}

bool is_italic_bold_initiated(char *source)
{
    int len = strlen(source);
    if(len < 4) return false;

    if(source[0] == '*' && source[1] == '*' && source[2] == '*') return true;

    return false;
}

bool is_italic_bold_terminated(char *source)
{
    int len = strlen(source);
    if(len < 4) return false;

    if(source[len-1] == '*' && source[len-2] == '*' && source[len-3] == '*') return true;

    return false;
}

char* string_substring(char *source, int inclusive_start, int exclusive_end)
{
    int required_chars = (exclusive_end - inclusive_start) + 1;
    char *result = malloc(sizeof(char) * required_chars);
    int j = 0;
    for(int i = inclusive_start; i < exclusive_end; i++)
    {
        result[j] = source[i];
        j++;
    }
    result[j] = '\0';
    return result;
}

bool flip_boolean(bool b)
{
    if(b) return false;
    else return true;
}

int main(int argc, char *argv[])
{
    if(argc < 2) return 1;

    FILE *source = fopen(argv[1], "r");
    if(source == NULL)
    {
        printf("Could not open file \"%s\". Does it exist?\n", argv[1]);
        return 1;
    }

    // get all contents of the file
    int len = 0;
    int capacity = 100;
    char *contents = malloc(sizeof(char) * capacity);
    char c;
    while(true)
    {
        c = fgetc(source);
        if(c == EOF) break;

        contents[len] = c;
        len = len + 1;
        if(len >= capacity)
        {
            capacity *= 2;
            contents = realloc(contents, sizeof(char) * capacity);
        }
    }
    contents[len] = '\0';
    len = len + 1;



    // tokenize the file (split by new lines)
    char **string_tokens = split_string(contents, "\n");
    free(contents);
    int TEMP_INT = 0; // remove later
    printf("File contents (delimited by newlines):\n\n");
    while(string_tokens[TEMP_INT] != NULL)
    {
        printf("line %d: %s\n", TEMP_INT, string_tokens[TEMP_INT]);
        TEMP_INT++;
    }



    // HTML translation
    int html_len = 0;
    int html_capacity = 100;
    char *html = malloc(sizeof(char) * html_capacity);
    concatenate(&html, &html_len, &html_capacity, "<!DOCTYPE html>\n<html>\n"); // top of document
    concatenate(&html, &html_len, &html_capacity, "<head>\n\t <style>\n\t\tbody {\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: ##fff4e5;\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.admonition p {\nmargin-top: 10px;\n}\n</style></head>\n<body>\n");

    // variables for HTML translation
    int header_level;
    int code_level;
    int admonition_level;
    char *substring;
    char **split; // used to further split a line into whitespace-delimited tokens. Each token is then analyzed for
                  // lower order markers, like ** for bold. A "lower order marking" is one that does not effect the whole
                  // line, unlike # for headers and "| " for tables
    char *admonition_type;
    char *partially_converted_html;
    char *joined;
    bool building_code_block = false;
    bool building_admonition = false;
    bool building_unordered_list = false;
    bool building_deflist = false;
    bool italic_bold_initiated = false;
    bool bold_initiated = false;
    bool italic_initiated = false;
    bool underline_initiated = false;
    bool highlight_initiated = false;
    bool MODIFY_FLAG = false; // used to keep track of whether or not a whitespace-delimited token from char *split
                              // was modified to replace markdown with HTML (e.g. replacing ** for <b>)
    int lower_bound;
    int upper_bound;
    int split_index;
    int preprocessing_token_len;
    int joined_len;
    int joined_capacity;
    int partially_converted_html_len;
    int partially_converted_html_capacity;
    int j; // used to iterate through the characters of split[split_token] when replacing asterisks with <b> and <i> tags

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


        while(split[split_index] != NULL)
        {
            preprocessing_token_len = strlen(split[split_index]);

            // converted token
            partially_converted_html_len = 0;
            partially_converted_html_capacity = ((5 * preprocessing_token_len) / 8); // resize is inevitable, but minimize
                                                                                      // wasted space
            partially_converted_html = malloc(sizeof(char) * partially_converted_html_capacity);

            // used if any markdown was converted to HTML
            MODIFY_FLAG = false;

            // DEBUGGING
            printf("On sub-token \"%s\"\n", split[split_index]);

            // used to go through each individual character of split[split_index]
            j = 0;


            // case: underline (++underline++)
            // NOTE: this is a single if case, not followed by else-if cases, because there is no
            //       overlap between underline and other cases. The same cannot be said about italic, bold,
            //       and italic bold, because they use the same character to delimit (*, **, and ***)
            if(preprocessing_token_len > 2 && is_substring(split[split_index], "++"))
            {
                printf("DEBUG: found token \"%s\" to be part of an underline section.\n", split[split_index]);
                while(j < preprocessing_token_len - 1)
                {
                    if(split[split_index][j] == '+' && split[split_index][j+1] == '+')
                    {
                        MODIFY_FLAG = true;
                        underline_initiated = flip_boolean(underline_initiated);

                        if(underline_initiated) concatenate(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, "<u>");
                        else concatenate(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, "</u>");

                        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]);
                }
            }

```

```c

            // case: italic bold
            if(preprocessing_token_len > 3 && is_substring(split[split_index], "***"))
            {
                printf("DEBUG: found token \"%s\" to be part of an italic bold section.\n", split[split_index]);
                // the idea is to copy character by character until we reach the delimiter. If we reach the delimiter,
                // then do not copy the character. Instead, copy <b><i> or </i></b> depending on whether we have already
                // opened a <b><i> tag-pair. This is indicated by the italic_bold_initiated variable. Then, skip ahead
                // 3 characters so that we skip the first, second, and third asterisk.
                while(j < preprocessing_token_len - 2)
                {
                    if(split[split_index][j] == '*' && split[split_index][j+1] == '*' && split[split_index][j+2] == '*')
                    {
                        MODIFY_FLAG = true;
                        italic_bold_initiated= flip_boolean(italic_bold_initiated);

                        if(italic_bold_initiated) concatenate(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, "<b><i>");
                        else concatenate(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, "</i></b>");

                        j += 3;
                    }
                    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)
                {
                    while(j < preprocessing_token_len)
                    {
                        insert_char(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, split[split_index][j]);
                        j++;
                    }
                }
            }

            // case: bold
            else if(preprocessing_token_len > 2 && is_substring(split[split_index], "**"))
            {
                printf("DEBUG: found token \"%s\" to be part of a bold section.\n", split[split_index]);
                while(j < preprocessing_token_len - 1)
                {
                    if(split[split_index][j] == '*' && split[split_index][j+1] == '*')
                    {
                        MODIFY_FLAG = true;
                        bold_initiated = flip_boolean(bold_initiated);

                        if(bold_initiated) concatenate(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, "<b>");
                        else concatenate(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, "</b>");

                        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: italic
            else if(preprocessing_token_len > 1 && string_contains_char(split[split_index], '*'))
            {
                printf("DEBUG: found token \"%s\" to be part of an italic section.\n", split[split_index]);
                printf("DEBUG [ITALIC]: value of j is %d and value of preprocessing_token_len is %d\n", j, preprocessing_token_len);
                while(j < preprocessing_token_len)
                {
                    if(split[split_index][j] == '*')
                    {
                        MODIFY_FLAG = true;
                        italic_initiated = flip_boolean(italic_initiated);

                        if(italic_initiated) concatenate(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, "<i>");
                        else concatenate(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, "</i>");
                    }
                    else
                    {
                        insert_char(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, split[split_index][j]);
                    }
                    j++;
                }
                // unlike italic-bold and bold, there is no need to try to get any missing characters. There should not
                // be characters that were missing.
            }

            // do nothing
            else
            {
            }

            if(MODIFY_FLAG)
            {
                insert_char(&partially_converted_html, &partially_converted_html_len, &partially_converted_html_capacity, '\0');
                free(split[split_index]);
                split[split_index] = strdup(partially_converted_html);
                free(partially_converted_html);
            }

            split_index++;
        }

        // setup to join the split line
        split_index = 0;
        joined_len = 0;
        joined_capacity = 50;
        joined = malloc(sizeof(char) * joined_capacity);

        // join the split line -- the joined version will have all the lower-order HTML tags
        // - make sure to free memory while at it
        while(split[split_index] != NULL)
        {
            concatenate(&joined, &joined_len, &joined_capacity, split[split_index]);
            free(split[split_index]);
            insert_char(&joined, &joined_len, &joined_capacity, ' ');
            split_index++;
        }
        joined[joined_len] = '\0';

        // free memory and move joined split line into string_tokens[i]
        free(split);
        free(string_tokens[i]);
        string_tokens[i] = strdup(joined);
        free(joined);

        printf("DEBUG: Found line \"%s\" to be %d for call to is_deflist\n", string_tokens[i], is_deflist(string_tokens[i]));
        // case: header
        // ------------- do not check for bold or italic -- no bold or italic permitted in headers
        if(string_tokens[i][0] == '#')
        {
            // determine level of header
            header_level = determine_header_level(string_tokens[i]);
            printf("Header level found to be %d for line \"%s\"\n", header_level, string_tokens[i]);
            while(string_tokens[i][header_level] == '#')
            {
    header_level++;
            }

            // add header tag
            concatenate(&html, &html_len, &html_capacity, "<h");
            insert_char(&html, &html_len, &html_capacity, header_level + '0');
            insert_char(&html, &html_len, &html_capacity, '>');

            // add header contents -- this requires the removal of the # characters
            substring = string_substring(string_tokens[i], header_level+1, strlen(string_tokens[i]));
            concatenate(&html, &html_len, &html_capacity, substring);
            free(substring);

            // end the tag
            concatenate(&html, &html_len, &html_capacity, "</h");
            insert_char(&html, &html_len, &html_capacity, header_level + '0');
            concatenate(&html, &html_len, &html_capacity, ">");
            header_level = 0;
        }

        // case: code block
        // ------------- do not check for bold or italic -- no bold or italic permitted in headers
        else if(string_tokens[i][0] == '`')
        {
            code_level = determine_code_level(string_tokens[i]);
            if(code_level == 3)
            {
                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]);
                building_code_block = flip_boolean(building_code_block);
            }
            if(building_code_block) concatenate(&html, &html_len, &html_capacity, "<code>");
            else concatenate(&html, &html_len, &html_capacity, "</code>");
        }

        // case: admonition
        else if(string_tokens[i][0] == '!')
        {
            admonition_level = determine_admoninition_level(string_tokens[i]);
            if(admonition_level == 3)
            {
                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]);
                // upon reaching admonition beginning, invert value of building_admonition so as to indicate when
                // we have stopped and started building an admonition. The start and stop of an admonition looks the same
                // except that the start has more text after it.
                building_admonition = flip_boolean(building_admonition);
            }
            printf("%s\n", string_tokens[i]);

            // get the type of admonition -- remember to free this later
            split = split_string(string_tokens[i], " ");
            admonition_type = split[1];
            printf("DEBUG: string_tokens[i] after split to get admonition type: \"%s\"\n", string_tokens[i]);

            if(building_admonition)
            {
                // create div tag including the admonition class and the subclass (e.g. admonition note)
                concatenate(&html, &html_len, &html_capacity, "<div class=\"admonition ");
                concatenate(&html, &html_len, &html_capacity, admonition_type);
                concatenate(&html, &html_len, &html_capacity, "\">\n");

                // add the admonition type in the form of h4 (e.g. <h4>example<h4>)
                concatenate(&html, &html_len, &html_capacity, "<h4>");
                concatenate(&html, &html_len, &html_capacity, admonition_type);

                // add the rest of the title -- this is just everything after the "!!! example " or "!!! note " part
                lower_bound = 4 + strlen(admonition_type); // The +4 skips over the "!!! " part
                                                            // and the strlen(admonition_type) skips over the "example part"
                                                            // of the "!!! example ". finally since index is 1 smaller, the space
                                                            // after "example" is ignored
                upper_bound = strlen(string_tokens[i]);
                printf("DEBUG: found lower bound = %d and upper bound = %d for line \"%s\"\n", lower_bound, upper_bound, string_tokens[i]);
                if(upper_bound > lower_bound)
                {
                    // add the rest of the title
                    substring = string_substring(string_tokens[i], 4 + strlen(admonition_type), strlen(string_tokens[i])); // the rest of the title
                    insert_char(&html, &html_len, &html_capacity, ':'); // since default C (no
                    unicode -- I cannot figure out how to get it to work
                                                                         // on my system with unicode), write it like "example: How to parse markdown"
                                                                         // instead of with an emoji preceding the title
                    concatenate(&html, &html_len, &html_capacity, substring);

                    // 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);

                // close the h4 label tag
                concatenate(&html, &html_len, &html_capacity, "</h4>");
            }
            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(&html, &html_len, &html_capacity, "<dl>");
                concatenate(&html, &html_len, &html_capacity, "\n\t<dt>");
                substring = string_substring(string_tokens[i], 2, strlen(string_tokens[i]));
                concatenate(&html, &html_len, &html_capacity, substring);
                free(substring);
                concatenate(&html, &html_len, &html_capacity, "</dt>");
            }
            else concatenate(&html, &html_len, &html_capacity, "</dl>");
        }

        // case: paragraph
        else
        {
            // whenever not building a list, just add a simple paragraph <p> tag
            if( (!building_unordered_list) && (!building_deflist) )
            {
                concatenate(&html, &html_len, &html_capacity, "<p>");
                concatenate(&html, &html_len, &html_capacity, string_tokens[i]);
                concatenate(&html, &html_len, &html_capacity, "</p>");
            }
            else if(building_unordered_list)
            {
                concatenate(&html, &html_len, &html_capacity, "<li>");
                concatenate(&html, &html_len, &html_capacity, string_tokens[i]);
                concatenate(&html, &html_len, &html_capacity, "</li>");
            }

            else if(building_deflist)
            {
                concatenate(&html, &html_len, &html_capacity, "\t<dd>");
                concatenate(&html, &html_len, &html_capacity, string_tokens[i]);
                concatenate(&html, &html_len, &html_capacity, "</dd>");
            }
            // impossible case
            else
            {
                printf("Now how did this happen?\n");
            }
        }

        // go to next token
        insert_char(&html, &html_len, &html_capacity, '\n');
        i++;
    } // end of while



    // terminate and display HTML
    concatenate(&html, &html_len, &html_capacity, "\n</body>\n</html>");
    insert_char(&html, &html_len, &html_capacity, '\0');
    printf("len %d capacity %d\n", html_len, html_capacity);
    printf("%s", html);



    // cleanup
    for(int j = 0; j < i; j++)
        free(string_tokens[j]);

    free(string_tokens);
    return 0;

}
```