

multithreaded/stack.c

```
1  /*
2  A dynamic stack holding strings
3  - note: this current implementation only resizes stack to increase size, never to decrease size
4  */
5
6  #include "stack.h"
7
8  /**
9   * @brief initialize stack with an initial size (capacity)
10  * @param p STACK*: a pointer to uninitialized memory
11  * @param stack_size size_t: initial capacity of the stack to be initialized
12  * @return 0 on success; 1 otherwise
13  */
14  int init_stack(STACK *p, size_t stack_size)
15  {
16      if (p == NULL || stack_size == 0)
17      {
18          return 1;
19      }
20
21      p->size = stack_size;
22      p->pos = -1;
23      p->items = (char **)malloc(stack_size * sizeof(char *));
24
25      memset(p->items, 0, stack_size * sizeof(char *));
26
27      return 0;
28  }
29
30  /**
31  * @brief push an item onto the stack; if the stack is full, resize the stack
32  * @param p STACK*: (pointer to) the stack the function will push item onto
33  * @param item char*: string to push onto stack
34  * @return 0 on success; 1 otherwise
35  */
36  int push_stack(STACK *p, char *item)
37  {
38      if (p == NULL)
39      {
40          return 1;
41      }
42
43      if (is_full_stack(p))
44      {
45          resize_stack(p);
46      }
47
48      ++(p->pos);
```

```
49     // strlen(item) + 1 for end of string character
50     p->items[p->pos] = malloc((strlen(item) + 1) * sizeof(char));
51     memset(p->items[p->pos], 0, (strlen(item) + 1) * sizeof(char));
52     strncpy(p->items[p->pos], item, strlen(item));
53
54     return 0;
55 }
56
57 /**
58  * @brief pop from the stack
59  * @param p STACK*: (pointer to) the stack the function will pop from
60  * @param p_item char**: pointer that will be populated with popped element
61  * @return 0 on success; 1 otherwise
62  * @note the caller is responsible for deallocating memory assigned to p_item
63  */
64 int pop_stack(STACK *p, char **p_item)
65 {
66     if ((p == NULL) || is_empty_stack(p))
67     {
68         return 1;
69     }
70
71     *p_item = malloc(sizeof(char) * (strlen(p->items[p->pos]) + 1));
72     memset(*p_item, 0, sizeof(char) * (strlen(p->items[p->pos]) + 1));
73     strncpy(*p_item, p->items[p->pos], strlen(p->items[p->pos]));
74     free(p->items[p->pos]);
75     p->items[p->pos] = NULL;
76     (p->pos)--;
77     return 0;
78 }
79
80 /**
81  * @brief check if the stack is full
82  * @param p STACK*: (pointer to) the stack to check
83  * @return true if full; false otherwise
84  */
85 bool is_full_stack(STACK *p)
86 {
87     if (p == NULL)
88     {
89         return 0;
90     }
91     return (p->pos == (p->size - 1));
92 }
93
94 /**
95  * @brief check if the stack is empty
96  * @param p STACK*: (pointer to) the stack to check
97  * @return true if empty; false otherwise
98  */
```

```
99  bool is_empty_stack(STACK *p)
100  {
101      if (p == NULL)
102      {
103          return 0;
104      }
105      return (p->pos == -1);
106  }
107
108  /**
109   * @brief resize stack to have greater capacity; maintain existing elements
110   * @param p STACK*: (pointer to) the stack to resize
111   * @return 0 on success; 1 otherwise
112   */
113  int resize_stack(STACK *p)
114  {
115      size_t old_size = p->size;
116      char **old_items = p->items;
117      p->size = (p->size) * STACK_RESIZE_FACTOR;
118      p->items = (char **)malloc((p->size) * sizeof(char *));
119      if (p->items == NULL)
120      {
121          return 1;
122      }
123
124      for (size_t i = 0; i < old_size; ++i)
125      {
126          p->items[i] = old_items[i];
127      }
128      for (size_t i = old_size; i < p->size; ++i)
129      {
130          p->items[i] = NULL;
131      }
132
133      free(old_items);
134      old_items = NULL;
135
136      return 0;
137  }
138
139  /**
140   * @brief returns number of elements currently in the stack
141   * @param p STACK*: (pointer to) the stack
142   * @return number of elements in the stack
143   */
144  size_t num_elements_stack(STACK *p)
145  {
146      return p->pos + 1;
147  }
148
```

```
149 /**
150  * @brief deconstruct stack: free all allocated memory
151  * @param p STACK*: (pointer to) the stack to deconstruct
152  * @return 0 on success; 1 otherwise
153  */
154 int cleanup_stack(STACK *p)
155 {
156     if (p == NULL || p->items == NULL)
157     {
158         return 0;
159     }
160     for (size_t i = 0; i < p->size; ++i)
161     {
162         if (p->items[i] != NULL)
163         {
164             free(p->items[i]);
165             p->items[i] = NULL;
166         }
167     }
168     free(p->items);
169     p->items = NULL;
170     return 0;
171 }
```