**multithreaded/p_stack.c**

```
/*
A dynamic stack holding pointers
- note: this current implementation only resizes stack to increase size, never to decrease size
*/

#include "p_stack.h"

/**
 * @brief initialize stack with an initial size (capacity)
 * @param p PSTACK*: a pointer to uninitialized memory
 * @param stack_size size_t: initial capacity of the stack to be initialized
 * @return 0 on success; 1 otherwise
 */
int init_pstack(PSTACK *p, size_t stack_size)
{
    if (p == NULL || stack_size == 0)
    {
        return 1;
    }

    p->size = stack_size;
    p->pos = -1;
    p->items = (void **)malloc(stack_size * sizeof(void *));

    memset(p->items, 0, stack_size * sizeof(void *));

    return 0;
}

/**
 * @brief push an item onto the stack; if the stack is full, resize the stack
 * @param p PSTACK*: (pointer to) the stack the function will push item onto
 * @param item void*: pointer to push onto stack
 * @return 0 on success; 1 otherwise
 */
int push_pstack(PSTACK *p, void *item)
{
    if (p == NULL)
    {
        return 1;
    }

    if (is_full_pstack(p))
    {
        resize_pstack(p);
    }

    ++(p->pos);
```

```c
    p->items[p->pos] = item;

    return 0;
}

/**
 * @brief pop from the stack
 * @param p PSTACK*: (pointer to) the stack the function will pop from
 * @param p_item void**: pointer that will be populated with popped element (which itself is a pointer)
 * @return 0 on success; 1 otherwise
 */
int pop_pstack(PSTACK *p, void **p_item)
{
    if (p == NULL || is_empty_pstack(p))
    {
        return 1;
    }

    *p_item = p->items[p->pos];
    p->items[p->pos] = NULL;
    (p->pos)--;
    return 0;
}

/**
 * @brief check if the stack is full
 * @param p PSTACK*: (pointer to) the stack to check
 * @return true if full; false otherwise
 */
bool is_full_pstack(PSTACK *p)
{
    if (p == NULL)
    {
        return 0;
    }
    return (p->pos == (p->size - 1));
}

/**
 * @brief check if the stack is empty
 * @param p PSTACK*: (pointer to) the stack to check
 * @return true if empty; false otherwise
 */
bool is_empty_pstack(PSTACK *p)
{
    if (p == NULL)
    {
        return 0;
    }
    return (p->pos == -1);
}

/**
 * @brief resize stack to have greater capacity; maintain existing elements
 * @param p PSTACK*: (pointer to) the stack to resize
 * @return 0 on success; 1 otherwise
 */
int resize_pstack(PSTACK *p)
{
    size_t old_size = p->size;
    void **old_items = p->items;
    p->size = (p->size) * PSTACK_RESIZE_FACTOR;
    p->items = (void **)malloc((p->size) * sizeof(void *));
    if (p->items == NULL)
    {
        return 1;
    }

    for (size_t i = 0; i < old_size; ++i)
    {
        p->items[i] = old_items[i];
    }
    for (size_t i = old_size; i < p->size; ++i)
    {
        p->items[i] = NULL;
    }

    free(old_items);
    old_items = NULL;

    return 0;
}

/**
 * @brief returns number of elements currently in the stack
 * @param p STACK*: (pointer to) the stack
 * @return number of elements in the stack
 */
size_t num_elements_pstack(PSTACK *p)
{
    return p->pos + 1;
}

/**
 * @brief deconstruct stack: free all allocated memory
 * @param p PSTACK*: (pointer to) the stack to deconstruct
 * @return 0 on success; 1 otherwise
 */
int cleanup_pstack(PSTACK *p)
```

```c
{
    if (p == NULL || p->items == NULL)
    {
        return 0;
    }
    for (size_t i = 0; i < p->size; ++i)
    {
        if (p->items[i] != NULL)
        {
            free(p->items[i]);
            p->items[i] = NULL;
        }
    }
    free(p->items);
    p->items = NULL;
    return 0;
}
```