CS2253 – Assignment #1
Graham Hill 3587614

Stack.c:

```
/**********************************************************
 *        Stack.c
 *
 *  Modified by Graham Hill
 *        Created by Jean-Philippe Legault
 *
 * Your task is to implement the section with the comment:
 * * TODO: finish implementing this
 **********************************************************/

// allows the usage of `scanf` and `printf`
#include <stdio.h>

// Has the macro definition for EXIT_SUCCESS
#include <stdlib.h>

// allows the usage of `bool`
#include <stdbool.h>

/**
 * function:
 *                is_whitespace
 *
 * expects:
 *                a single char
 *
 * returns:
 *        true when the char is a whitespace character
 *                false otherwise
 */
bool is_whitespace(char in)
{
        return (in == ' ' || in == '\t' || in == '\n' || in == '\r');
}

/**
 * function:
 *                print_stack
 *
 * expects:
 *                a pointer to the root of the stack
 *                a pointer to the current size of the stack
 *
 * Prints a visual representation of the current state of the stack
 */
```

```
void print_stack(int *stack, int *size)
{
	for(int i=0; i<(*size); i++)
	{
		printf("|_ %d _|\n", stack[i] );
	}
	printf("%d elements\n", (*size) );
}

/**
 * function:
 *		push
 *
 * expects:
 *		pointer to the stack
 *		pointer to the size
 *		the value to push
 *
 * returns:
 *	true when value has been pushed
 *		false otherwise
 *
 * The push function push a value to the passed in stack
 */
bool push(int *stack, int *size, int max_size, int to_push)
{
	/**
	 * TODO: finish implementing this
	 */
	if((*size) == max_size){
		return false;
	}
	else{
		stack[(*size)] = to_push;
		(*size)++;
		return true;
	}
}

/**
 * function:
 *		pop
 *
 * expects:
 *		pointer to the stack
 *		pointer to the size
 *		pointer to location to store the popped value
 *
 * returns:
```

```
 *	true when value has been popped
 *		false otherwise
 *
 * The pop function pops a value from the passed in stack and stores it at the to_return location.
 */
bool pop(int *stack, int *size, int *to_return)
{
	/**
	 * TODO: finish implementing this
	 */
	if((*size) ==  0){
		return false;
	}
	else{
		(*to_return) = stack[(*size)-1];
		(*size)--;
		return true;
	}
}

/**
 * function:
 *		peek
 *
 * expects:
 *		pointer to the stack
 *		pointer to the size
 *		pointer to location to store the popped value
 *
 * returns:
 *	true when value has been peeked
 *		false otherwise
 *
 * The peek function looks at the top value from the stack and stores it at the to_return location.
 */
bool peek(int *stack, int *size, int *to_return)
{
	/**
	 * TODO: finish implementing this
	 */
	if((*size) ==  0){
		return false;
	}
	else{
		(*to_return) = stack[(*size)-1];
		return true;
	}
}
```

```
/*************************************************************
 * function implementation
 */

/**
 * function:
 *              main
 *
 * expects:
 *              n/a
 *
 * returns:
 *              EXIT_SUCCESS when program ends.
 *
 * while we are not instructed to exit the program
 *
 *              We read in a char as an instruction :
 *                      'u' for push
 *                      'o' for pop
 *                      'e' for peek
 *                      'x' to exit the program
 *
 *              if the instruction is push ('u'),
 *                      we read in an integer (you may assume it is a valid integer)
 *                      push the read value onto the stack
 *                      if failed
 *                              printf("failed push\n")
 *                      else
 *                              print the value pushed
 *
 *              else if the instruction is pop ('o')
 *                      we execute the pop function
 *                      if failed
 *                              printf("failed pop\n")
 *                      else
 *                              print the value popped
 *
 *              else if the instruction is peek ('e')
 *                      we execute the peek function
 *                      if failed
 *                              printf("failed peek\n")
 *                      else
 *                              print the value peeked
 *
 *              else if the instruction is exit ('x')
 *                      we break out of the loop
 *
 *              else
 *                      printf("invalid instruction %c\n", input_instruction);
```

```
 */
int main( int argc, char **argv )
{
	// keep track of the max size and the current size of the stack
	int stack_max_size = 5;
	int stack_current_size = 0;

	// the stack is an array located on the main() function stack frame
	int stack[stack_max_size];

	// initialize our stack with 0 values
	for(int i=0; i < stack_max_size; i++)
	{
		stack[i] = 0;
	}

	// count the number of instructions (peek, pop, push) that successfully happened
	int successful_instructions = 0;
	bool stop_execution = false;

	while(!stop_execution)
	{
		// read the input instruction (a single character)
		char input_instruction = 0;
		scanf("%c", &input_instruction);

		// the character could be a whitespace so we need to skip those
		if( false == is_whitespace(input_instruction) )
		{
			if (input_instruction == 'u'){
				int valueToPush = 0;

				if(scanf("%d", &valueToPush) == 0)
				{
					return EXIT_FAILURE;
				}

				bool pushed = push(stack, &stack_current_size, stack_max_size,
valueToPush);

				if (!pushed)
				{
					printf("failed push\n");
				}
				else
				{
					printf("%d\n", valueToPush);
					successful_instructions++;
				}
```

```
                }
                else if(input_instruction == 'o')
                {
                        int valuePopped = 0;
                        bool popped = pop(stack,&stack_current_size, &valuePopped);

                        if(!popped)
                        {
                                printf("failed pop\n");
                        }
                        else
                        {
                                printf("%d\n", valuePopped);
                                successful_instructions++;
                        }
                }
                else if(input_instruction == 'e')
                {
                        int peekValue = 0;
                        bool peeked = peek(stack, &stack_current_size, &peekValue);

                        if (!peeked)
                        {
                                printf("failed peek\n");
                        }
                        else
                        {
                                printf("%d\n", peekValue);
                                successful_instructions++;
                        }
                }
                else if(input_instruction == 'x')
                {
                        stop_execution = true;
                }
                else
                {
                        printf("invalid instruction %c\n", input_instruction);
                }

        }
    }
    printf("Successfully executed %d instructions\n", successful_instructions);
    print_stack(stack, &stack_current_size);

    return EXIT_SUCCESS;
}
```

Terminal Section:

```
ghill@ubuntu:~/Desktop/Lab 3/CS2263_Summer2019_A1$ make test
./Stack < Data/exit_test1.input > exit_test1.result
./TestPassed.sh exit_test1.result Data/exit_test1.expected

######   Passed   ###### exit_test1.result is equal to Data/exit_test1.expected

./Stack < Data/push_test1.input > push_test1.result
./TestPassed.sh push_test1.result Data/push_test1.expected

######   Passed   ###### push_test1.result is equal to Data/push_test1.expected

./Stack < Data/push_test2.input > push_test2.result
./TestPassed.sh push_test2.result Data/push_test2.expected

######   Passed   ###### push_test2.result is equal to Data/push_test2.expected

./Stack < Data/peek_test1.input > peek_test1.result
./TestPassed.sh peek_test1.result Data/peek_test1.expected

######   Passed   ###### peek_test1.result is equal to Data/peek_test1.expected

./Stack < Data/peek_test2.input > peek_test2.result
./TestPassed.sh peek_test2.result Data/peek_test2.expected

######   Passed   ###### peek_test2.result is equal to Data/peek_test2.expected

./Stack < Data/pop_test1.input > pop_test1.result
./TestPassed.sh pop_test1.result Data/pop_test1.expected

######   Passed   ###### pop_test1.result is equal to Data/pop_test1.expected

./Stack < Data/pop_test2.input > pop_test2.result
./TestPassed.sh pop_test2.result Data/pop_test2.expected

######   Passed   ###### pop_test2.result is equal to Data/pop_test2.expected

./Stack < Data/pop_test3.input > pop_test3.result
./TestPassed.sh pop_test3.result Data/pop_test3.expected

######   Passed   ###### pop_test3.result is equal to Data/pop_test3.expected

./Stack < Data/compound_test1.input > compound_test1.result
./TestPassed.sh compound_test1.result Data/compound_test1.expected

######   Passed   ###### compound_test1.result is equal to Data/compound_test1.expected

./Stack < Data/compound_test2.input > compound_test2.result
./TestPassed.sh compound_test2.result Data/compound_test2.expected

######   Passed   ###### compound_test2.result is equal to Data/compound_test2.expected

./Stack < Data/compound_test3.input > compound_test3.result
./TestPassed.sh compound_test3.result Data/compound_test3.expected

######   Passed   ###### compound_test3.result is equal to Data/compound_test3.expected
```

Custom Test:

```
./Stack < Data/customtest1.input > customtest1.result
./TestPassed.sh customtest1.result Data/customtest1.expected

######   Passed   ###### customtest1.result is equal to Data/customtest1.expected
```