91.102 : Exercise 6

Read

- Section 5.1
- Section 5.3

in Esakov and Weiss and do the following tasks.

1 Preparation

1. Make a new directory, called ex6, for this exercise. Always make a new directory for every exercise as you may need previous work later in the course.
2. Make sure you are using the most current version of the Makefile and globals.h.
3. Copy your old versions of stack.h and stack.c from the appropriate previous exercise.

2 Header Files

There are some things that you must make a habit of doing for every header file. Therefore, these things are not included in the instructions for each header file but are simply mentioned here.

1. Include the necessary pre-processor directive to prevent multiple inclusions.

3 All Files

There are some things that you must make a habit of doing for all files. Those things are not included in the instructions for each file, but are simple mentioned here.

1. Include the necessary header files so that your file can compile on its own. You must type "make headers" at the prompt to check for this property with header files.

2. Always use angle brackets when including system headers. Always use double quotes when including your personal header files.

3. Always include system headers before personal header files. Always alphabetize your list of system includes and your list of personal includes.

4 Primitive and Application

Note that stack.h and stack.c constitute the interface and implementation of the primitive. stack_utils.h and stack_utils.c are also part of the interface and implementation of the primitive, but they contain derived functionality, i.e., procedures that can be defined solely in terms of the core procedures declared in stack.h.

Also, note that it is very important to add the test code in the main procedure in the order given and before the unconditional line

```
return EXIT_SUCCESS;
```

which should only be reached if every test has succeeded. The comments should help you understand why the order matters.

1. Change the prototypes of init_stack and destroy_stack so that init_stack's is

   ```
   extern status init_stack(stack * * const p_S,
                            size_t const size);
   ```

   and destroy_stack's is

   ```
   extern void destroy_stack(stack * * const p_S,
                             void (*p_func_f)());
   ```

2. Put the following type definition in a file called stack.c.

   ```
   struct stack {
     size_t MAXSTACKSIZE;
     generic_ptr * base;
     generic_ptr * top;
   };
   ```

   Make sure to compile and fix any errors before you move on.

3. Put the following procedure implementation in stack.c. Make sure this compiles before you move on.

   ```
   static size_t num_elements(stack * p_S)
   {
     return p_S->top - p_S->base;
   }
   ```

4. Put the following procedure implementation in array.c. Make sure this compiles before you move on.

```
status init_stack(stack * * const p_S,
                  size_t const size)
{
  stack * temp = (stack *) malloc(sizeof(stack));
  if (temp == NULL) {
    return ERROR;
  }

  generic_ptr * temp_array =
    (generic_ptr *) calloc(size, sizeof(generic_ptr));
  if (temp_array == NULL) {
    free(temp);
    return ERROR;
  }

  temp -> MAXSTACKSIZE = size;
  temp -> top = temp -> base = temp_array;

  *p_S = temp;

  return OK;
}
```

5. Start a file called main.c to exercise the code that you have placed in array.c. Place the following main procedure in main.c and make sure that you can compile and execute main. Make sure that main returns 0, indicating success.

```
int main(int argc, char * * argv)
{
  size_t const stack_size = 10;
  stack * p_S;

  /* a stack is initialized successfully */
  if (init_stack(&p_S, stack_size) == ERROR) {
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

6. Add the implementation of the empty_stack procedure to array.c like this. Make sure stack.c compiles before you move on.

```
bool empty_stack(stack * const p_S)
{
  return p_S -> top == p_S -> base;
}
```

7. Exercise the empty_stack procedure and check the correctness of init_stack

by adding the following lines of code to the main procedure. Make sure the code compiles, that you can run the executable, main, and that main returns 0.

```
/* a newly created stack is empty */
if (!empty_stack(p_S)) {
  return EXIT_FAILURE;
}
```

8. Add the implementation of the push procedure to stack.c like this. Make sure stack.c compiles before you move on.

```
status push(stack * const p_S,
            generic_ptr const data)
{
  if (num_elements(p_S) == p_S->MAXSTACKSIZE) {
    return ERROR;
  }
  *p_S -> top = data;
  p_S -> top++;
  return OK;
}
```

9. Exercise the push procedure by adding the following lines of code to the main procedure. Make sure the code compiles, that you can run the executable, main, and that main returns 0.

```
/* pushing stack_size elements onto a stack succeeds */
for(size_t i = 0; i != stack_size; ++i) {
  generic_ptr p = NULL;
  if (push(p_S, p) == ERROR) {
    return EXIT_FAILURE;
  }
}

/* pushing the stack_size + 1 th element fails */
{
  generic_ptr p;
  if (push(p_S, p) != ERROR) {
    return EXIT_FAILURE;
  }
}

/* a stack with elements pushed on is not empty */
if (empty_stack(p_S)) {
  return EXIT_FAILURE;
}
```

10. Add the implementation of the pop procedure to stack.c like this. Make sure stack.c compiles before you move on.

```
status pop(stack * const p_S,
           generic_ptr * const p_data)
{
  if (empty_stack(p_S)) {
    return ERROR;
  }
  p_S -> top--;
  *p_data = *p_S->top;
  return OK;
}
```

11. Exercise the pop procedure by adding the following lines of code to the main procedure. Make sure the code compiles, that you can run the executable, main, and that main returns 0.

```
/* stack_size pops from a full stack succeed */
for(size_t i = 0; i != stack_size; ++i) {
  generic_ptr p;
  if (pop(p_S, &p) == ERROR) {
    return EXIT_FAILURE;
  }
}

/* popping the stack_size + 1 th element fails */
{
  generic_ptr p;
  if (pop(p_S, &p) != ERROR) {
    return EXIT_FAILURE;
  }
}

/* a stack which has had its last element removed is empty */
if(!empty_stack(p_S)) {
  return EXIT_FAILURE;
}
```

12. Add the implementation of the top procedure to stack_utils.c like this. Make sure stack_utils.c compiles before you move on.

```
status top(stack * const p_S,
           generic_ptr * const p_data)
{
  generic_ptr temp;
  if (pop(p_S, &temp) == ERROR ||
      push(p_S, temp) == ERROR) {
    return ERROR;
  }
  *p_data = temp;
  return OK;
}
```

13. Exercise the top procedure by adding the following lines of code to the main procedure. Make sure the code compiles, that you can run the executable, main, and that main returns 0.

```
/* accessing the top element of an empty stack gives an error */
{
  generic_ptr p;
  if (top(p_S, &p) != ERROR) {
    return EXIT_FAILURE;
  }
}

for(size_t i = 0; i != stack_size; ++i) {
  generic_ptr p = NULL;
  if (push(p_S, p) == ERROR) {
    return EXIT_FAILURE;
  }
}

/* accessing the top element of a non-empty stack succeeds */
{
  generic_ptr p;
  if (top(p_S, &p) == ERROR) {
    return EXIT_FAILURE;
  }
}
```

14. Add the implementation of the destroy_stack procedure to stack.c like this. Make sure stack.c compiles before you move on

```
void destroy_stack(stack * * const p_S,
                   void (*p_func_f)())
{
  for (size_t i = 0; i != (*p_S)->MAXSTACKSIZE; ++i) {
    (*p_func_f)(&((*p_S)->base)[i]);
  }
  free((*p_S)->base);
  free(*p_S);
  *p_S = NULL;
}
```

15. Exercise the destroy_stack procedure by adding the following lines of code to main.c. Make sure the code compiles, that you can run the executable, main, and that main returns 0.

```
void destroy_generic_ptr(generic_ptr * p)
{
  free(*p);
  *p = NULL;
}
```

16. Then add the following lines of code to the main procedure. Make sure the code compiles, that you can run the executable, main, and that main returns 0.

```
destroy_stack(&p_S, destroy_generic_ptr);

/* the stack pointer is NULL after the stack is destroyed */
if (p_S != NULL) {
  return EXIT_FAILURE;
}
```

5 Final Steps

1. Type "make cppcheck" and fix any errors. Note that cppcheck may not be installed on your personal computer. If it is not, you will have to do the final step on the CS computers.

2. You are now done. Hand in using Bottlenose and fix any errors that Bottlenose exhibits.