This program will take the first few daily assignments and combine their behaviors to make an opaque object wrapper that allows for all the behaviors we want in a bit collection data structure called BIT_FLAGS.

```
#ifndef BIT_FLAGS_H
#define BIT_FLAGS_H
#include "status.h"

typedef void* BIT_FLAGS;

//Intentionally leaving out a default init function to force user to at least guess at the size needed.
//If one WERE to be used it would have the following prototype:
//BIT_FLAGS bit_flags_init_default(void);

//Precondition: number of bits is a positive integer.
//Postcondition: Returns the handle to a valid Bit_flags object that has the ability to store up to
//   number_of_bits bits but currently all flags are set at zero. Returns NULL on failure.  The container
//   is assumed to hold size=number_of_bits after the init function runs.
BIT_FLAGS bit_flags_init_number_of_bits(int number_of_bits);

//Precondition: flag_position is a non-negative integer and hBit_flags is a handle to a valid Bit_flags object.
//Postcondition: The flag at the flag_position index is set to 1.  Function will attempt to resize the
//   internal representation if the flag_position is too large instead of failing for out of bounds.  Returns
//   SUCCESS if the operation is successful and FAILURE if the operation fails a needed resize.  This
//   operation is considered to be expensive if flag_position is constantly going out of bounds by a small
//   amount because the resize always attempts to minimize the amount of space required to store the bits.
//   All new flags created in a resize operation (except the one being set) will be set as zero.
Status bit_flags_set_flag(BIT_FLAGS hBit_flags, int flag_position);

//Precondition: flag_position is a non-negative integer and hBit_flags is a handle to a valid Bit_flags object.
//Postcondition: The flag at the flag_position index is set to 0.  Function will attempt to resize the
//   internal representation if the flag_position is too large instead of failing for out of bounds.  Returns
//   SUCCESS if the operation is successful and FAILURE if the operation fails a needed resize.  This
//   operation is considered to be expensive if flag_position is constantly going out of bounds by a small
//   amount because the resize always attempts to minimize the amount of space required to store the bits.
//   All new flags created in a resize operation will be set as zero.
Status bit_flags_unset_flag(BIT_FLAGS hBit_flags, int flag_position);

//Precondition: flag_position is a non-negative integer and hBit_flags is a handle to a valid Bit_flags object.
//Postcondition: returns the value of the flag at index flag_position if it is in bounds or -1 otherwise.
int bit_flags_check_flag(BIT_FLAGS hBit_flags, int flag_position);

//Precondition: hBit_flags is a handle to a valid Bit_flags object.
//Postcondition: returns the number of bits currently held by the data structure.
int bit_flags_get_size(BIT_FLAGS hBit_flags);

//Precondition: hBit_flags is a handle to a valid Bit_flags object.
//Postcondition: returns the number of bits the object CAN hold.
int bit_flags_get_capacity(BIT_FLAGS hBit_flags);

//Precondition: phBit_flags is the address of a handle to a valid Bit_flags object.
//Postcondition: The memory for the object referred to by the handle is free'd and the handle is set to NULL.
void bit_flags_destroy(BIT_FLAGS* phBit_flags);

#endif
```

Think of flag_position as being the index of an array of bits that you will have access to set and unset at will. This data structure will make the user believe that you have such an array when you are really handling it with an array of well used integers instead.

Turn in only your bit_flags.h and bit_flags.c files for grading. The graders will use their own main program to test your data structure but you should test it as well as you can yourself with your own driver. The graders will also assume you are using the status.h we created in class. Please note that you may not change the prototype of any of these functions as the functions must work with our driver you can only use these.

In the comments section of your code think about your data structure and what might be missing. Come up with one proposed function that you could add to the interface (you do not have to write it) and explain why you think it should be added (what it would do and how it would be useful).

Here are a few scenarios of expected behavior to help you debug your program:

If I create a BIT_FlAGS object and ask for 10 bits in it then the size will be 10 and the expected capacity will probably be 32 (assuming you have 32 bit integers on your machine). You should not assume the size of integers on the machine though so we can get different results if we run on a machine that uses 64 bit integers.

If I take the object above and try to set or unset the flag at position 100 then this is out of bounds so the object will have to resize to accommodate the additional data. The resulting size would be 101 and the capacity would be 128 assuming 32 bit integers. Why is the size 101 and not 100? It is because the flag position is like an index into an array and the array starts at 0 so the one at position 100 is actually the one hundred first bit.

At the top of your code you should have a comment section that has the following format:

```
/**********************************************************
       Author: <your name>
       Date: <Today's date>
       Effort: <Time you spent on this project>
       Purpose: <Purpose of this assignment in your own words>
       Interface proposal: <answer the last question in the specification>
**********************************************************/
```