Lecture 3 Pre-class Exercise

Q: Review the following C implementation of a vector. There are at least 7 bugs. You don't need to catch 'em all, but try to spot as many as you can.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// There are at least 7 bugs relating to memory on this snippet.
// Find them all!

// Vec is short for "vector", a common term for a resizable array.
// For simplicity, our vector type can only hold ints.
typedef struct {
    int* data;     // Pointer to our array on the heap
    int  length;   // How many elements are in our array
    int  capacity; // How many elements our array can hold
} Vec;

/* Return a pointer to a new, empty Vec object. */
Vec* vec_new() {
    Vec vec;
    vec.data = NULL;
    vec.length = 0;
    vec.capacity = 0;
    return &vec;
}

/* Push a new integer `n` onto `vec`. */
void vec_push(Vec* vec, int n) {
    if (vec->length == vec->capacity) {
        /* If previously-allocated space has been filled,
        allocate a new array for the vector contents
        with 2x the capacity, and copy over data. */
        int new_capacity = vec->capacity * 2;
        int* new_data = (int*) malloc(new_capacity);
```

```
        assert(new_data != NULL); /* Check that `malloc` succeeded. */

        for (int i = 0; i < vec->length; ++i) {
            new_data[i] = vec->data[i];
        }

        vec->data = new_data;
        vec->capacity = new_capacity;
    }

    vec->data[vec->length] = n;
    ++vec->length;
}

/* Free the vector and associated data. */
void vec_free(Vec* vec) {
    free(vec);
    free(vec->data);
}

/* Test program. */
void main() {
    Vec* vec = vec_new();
    vec_push(vec, 107);

    int* n = &vec->data[0];
    vec_push(vec, 110);
    printf("%d\n", *n);

    free(vec->data);
    vec_free(vec);
}
```

A:

1. **Returning a pointer to a local variable:** `vec_new()` was returning the address of a local variable `vec`. This memory is invalid once the function returns. **Fix:** Allocate `Vec` on the heap using `malloc`.
2. **Memory leak in `vec_push()` when `capacity` is 0:** When `capacity` is initially 0, multiplying by 2 still gives 0. **Fix:** Initialize `new_capacity` to 1 if `vec->capacity` is 0.
3. **Incorrect memory allocation size:** `malloc` in `vec_push` wasn't multiplying by `sizeof(int)`. **Fix:** Allocate `new_capacity * sizeof(int)` bytes.
4. **Memory leak on reallocation:** The old `vec->data` wasn't being freed when reallocating. **Fix:** Use `realloc` which handles the copying and freeing of the old memory block.
5. **Error handling for `malloc` and `realloc`:** The code didn't check if `malloc` or `realloc` failed. **Fix:** Add checks for `NULL` returns and handle errors appropriately.
6. **Double freeing in `vec_free()` and `main()`:** Both `vec_free()` and `main()` called `free(vec->data)`. **Fix:** Remove the redundant `free` in `main()`.
7. **Incorrect `free` in `vec_free()`:** The code was freeing the `Vec` struct *before* freeing the data it pointed to. This could lead to a crash if `free` tries to access the already-freed `vec->data`. **Fix:** Free `vec->data` first, then `vec`.
8. **`void main()`:** The correct signature for the main function is `int main()`. **Fix:** Change the return type to `int` and return 0 at the end.

Corrected Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct {
    int* data;
    int  length;
    int  capacity;
} Vec;

/* Return a pointer to a new, empty Vec object. */
Vec* vec_new() {
```

```c
    // Allocate memory on the heap for the Vec struct itself
    Vec* vec = (Vec*)malloc(sizeof(Vec));
    if (vec == NULL) {
        perror("Failed to allocate memory for Vec");
        exit(EXIT_FAILURE); // Handle allocation failure
    }
    vec->data = NULL;
    vec->length = 0;
    vec->capacity = 0;
    return vec;
}

/* Push a new integer `n` onto `vec`. */
void vec_push(Vec* vec, int n) {
    if (vec->length == vec->capacity) {
        // Handle the initial case where capacity is 0
        int new_capacity = vec->capacity == 0 ? 1 : vec->capacity * 2;
        int* new_data = (int*)realloc(vec->data, new_capacity *
sizeof(int));
        if (new_data == NULL) {
            perror("Failed to reallocate memory");
            exit(EXIT_FAILURE); // Handle reallocation failure
        }

        vec->data = new_data;
        vec->capacity = new_capacity;
    }

    vec->data[vec->length] = n;
    ++vec->length;
}

/* Free the vector and associated data. */
void vec_free(Vec* vec) {
    free(vec->data); // Free the data array first
    free(vec);       // Then free the struct itself
}
```

```c
int main() { // Use int main and return 0
    Vec* vec = vec_new();
    vec_push(vec, 107);

    int* n = &vec->data[0];
    vec_push(vec, 110);
    printf("%d\n", *n);  // This will now correctly print 107

    vec_free(vec); // Don't double free vec->data

    return 0;
}
```