"MyVector" Lab

In this lab, you will create your own version of the `vector` data type. To simplify things, our vectors will only store integers, and they will only have the ability to get larger on the fly as we add items, not get smaller if we remove items (the way C++ vectors can). Furthermore, we will add bounds-checking to `myvector`s like Python.

Here's how a `myvector` will work (very similar to C++ vectors):

- The items in a `myvector` will be stored on the heap, in a dynamically-allocated array. We need to do this because all arrays declared on the stack (automatic variables) must have their sizes set in stone at compile-time, and we don't want that restriction.
- A `myvector` is comprised of three variables (in a `struct`):
  - `int * items`, a pointer to a C++ array of integers on the heap.
  - `int size`, the current number of items in the `myvector` (from the user's perspective)
  - `int capacity`, the current capacity of the items array (from the programmer's perspective)
- We need both `size` and `capacity` because as we add items to the `myvector`, we don't want to overflow the array too often. Therefore, we will allocate extra space in the array that we will use up as we put items into the `myvector`.
- When `size` equals `capacity`, this means the array is full and we can't add any more items. If the user asks to add another item, we will have to allocate a new block of memory for a new array (with some more extra space), copy the items in the old array into the new one, then de-allocate the old array.
- So we can see the re-allocation happen more often, a `myvector` will have an initial capacity of 3 and will grow by 3 items every time we increase the capacity, even though in the "real world," this is way too small an increment (10 is probably more common).

Here's the `struct` you should use:

```
struct myvector {
  int size, capacity;
  int * items;
};
```

Write the following functions along with a `main()` function to test them as you write them.

1. `myvector create_myvector()`: Create and return a new `myvector` with `size=0` and `capacity=3`. This means from the user's perspective, the `myvector` will be empty, but behind the scenes, there is space to add three items before we need more memory.

2. `void delete_myvector(myvector & vec)`: Deallocate the space for items in this `myvector`. After you call this function, your `myvector` will be unusable (because we returned the memory to the operating system).

3. `void append(myvector & vec, int value)`: Adds (like push_back) a new value to the `myvector`. This will increase `size` by one. For now, if there is no more space left (`size == capacity`), print an error message and don't change anything in the `struct` (we will work on the re-allocation later).

4. `int get(const myvector & vec, int pos)`: Return `items[pos]` in the `myvector`, assuming 0 <= pos < size. If this is not true, print an error message and return -1.

5. `void print(const myvector & vec)`: Print out the contents of a `myvector` in an easy-to-read format (e.g., all the integers on one line with spaces in between). Note that you should only print the items at positions [0] through [size-1], because while there might be more "valid" positions if `capacity` > `size`, we know (at the moment) those positions don't hold any meaningful values.

6. Edit your `append` function to support appending when the array is full. To do this, allocate a new array on the heap with enough space for the current capacity + 3. Then copy (you will need to use a `for` loop) each item from the old array into the new array. Then add the new value (that would have overflowed the original array) to the appropriate place in the new array. Then deallocate the old array. When done, `size` should be increased by 1, `capacity` should be increased by 3, and `items` should point to the new array. **Make this function print a message during the re-allocation so you can see when it happens.**

When you are all done, or even partially done, here's a main function to test things:

```
int main()
{
     myvector v1 = create_myvector();

     append(v1, 10);
     append(v1, 20);
     append(v1, 30);
     append(v1, 40);  // should trigger re-allocation
     print(v1);
     cout << get(v1, 2) << endl; // fine
     cout << get(v1, 4) << endl; // triggers error message

     append(v1, 2);
     append(v1, 4);
     append(v1, 8);  // re-alloc
     append(v1, 16);
     append(v1, 32);
     append(v1, 64); // re-alloc
     print(v1);

     cout << get(v1, 2) << endl; // fine
     cout << get(v1, 4) << endl; // fine b/c items[4] exists now
     cout << get(v1, 9) << endl; // fine
     cout << get(v1, 10) << endl; // error

     delete_myvector(v1);

     return 0;
}
```

If you finish early, try these:

- Add a `set` function to change an item in the `myvector`.
- Add a `remove_last` function to remove the last item in the `myvector`.
- Add a `remove` function to remove an item from **a specific position** in the `myvector`; e.g., `remove(v1, 2)` in the code above would remove the item at position 2. Any items to the right should slide over to not leave a hole in the `myvector`.
- Change the remove functions so that when size drops too far below capacity, the array is re-allocated to eliminate some of the unused space.