Jonathan Chen & Adam Tecle

We used Professor Russell's malloc2_prevs.c in implementing our solution. We did this to avoid saturation and fragmentation. The errors we attempted to catch were when the user frees a variable that had not been allocated, freeing a variable that has already been freed. Our implementations fails in following case:

```
int *x = malloc(sizeof(int));
free(x);
char *t = malloc(4);
free(x);
```

Our program successfully frees x a second time when it should prompt an error. This is because, during malloc, x is set to some address space. If the next malloc is less than or equal to the size of x, then we are writing to the same addresses space x was previously in.

In order to keep track of things we have freed, we have a static array of pointers initialized to size 5000. Each time we successfully free we add the pointer to this array. To check if something has already been freed, we check for the pointer passed to my_free in the array. If found, we print an error statement and return from my_free. To check if something hasn't been allocated, we compare the address of the pointer passed to my_free into each address returned from malloc, starting at the root. If not found, we print a relevant error statement.