Review: `DoubleList`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct {
6    double *data;
7    int size;
8  } DoubleList;
9
10 void list_append(DoubleList *list, double val);
11 void list_print(DoubleList *list);

13 int main() {
14   DoubleList temps = { NULL, 0 };
15
16   list_append(&temps, 68.5);
17   list_append(&temps, 72.0);
18   list_append(&temps, 65.3);
19
20   list_print(&temps);
21
22   free(temps.data);
23 }
```

`list_append` — version 0 (has a bug!)

```
25 void list_append(DoubleList *list, double val) {
26   double *new_data = malloc((list->size + 1) * sizeof(double));
27   memcpy(new_data, list->data, list->size * sizeof(double));
28   list->data = new_data;
29   list->data[list->size] = val;
30   list->size++;
31 }
```

```
$ valgrind ./list-leak
[68.5, 72.0, 65.3]
==2303169== HEAP SUMMARY:
==2303169==     in use at exit: 24 bytes in 2 blocks
==2303169==   total heap usage: 4 allocs, 2 frees, 1,072 bytes
    allocated
==2303169== LEAK SUMMARY:
==2303169==    definitely lost: 24 bytes in 2 blocks
```

Draw a picture of where the 24 bytes in 2 blocks are lost.

`list_append` — version 1, using `malloc` and `free`

```
void list_append(DoubleList *list, double val) {




}
```

`list_append` — version 2, using `realloc`

```
void list_append(DoubleList *list, double val) {




}
```

`realloc` wraps malloc + memcpy + free into one call:

```
void *realloc(void *ptr, size_t new_size);
```

- May extend in place or allocate + copy + free
- `realloc(NULL, n)` acts like `malloc(n)`

Filtering a `DoubleList`

```
16 int is_less_70(double val) { return val < 70.0; }
17 DoubleList filter_less_70(DoubleList *list) {
18   DoubleList result = { NULL, 0 };
19   for (int i = 0; i < list->size; i++) {
20     if (is_less_70(list->data[i])) {
21       list_append(&result, list->data[i]);
22     }
23   }
24   return result;
25 }

27 int is_integer(double val) { return val == (int)val; }
28 DoubleList filter_is_integer(DoubleList *list) {
29   DoubleList result = { NULL, 0 };
30   for (int i = 0; i < list->size; i++) {
31     if (is_integer(list->data[i])) {
32       list_append(&result, list->data[i]);
33     }
34   }
35   return result;
36 }

38 int is_unit(double val) { return val >= 0.0 && val <= 1.0; }
39 DoubleList filter_unit(DoubleList *list) {
40   DoubleList result = { NULL, 0 };
41   for (int i = 0; i < list->size; i++) {
42     if (is_unit(list->data[i])) {
43       list_append(&result, list->data[i]);
44     }
45   }
46   return result;
47 }
```

What's the same across all three functions?

What's different?

Factoring out the pattern: `list_filter`

```
typedef int (*Predicate)(double);

DoubleList list_filter(DoubleList *list, Predicate pred) {




}
```

Examples: hardcoded filters

```
void list_examples() {
  DoubleList d = { NULL, 0 };
  list_append(&d, 68.5);
  list_append(&d, 72.0);
  list_append(&d, 0.5);
  list_append(&d, 99.0);
  // How to call each filter function? What's the expected result
      for each?



}
```

Examples: using `list_filter`

```
void list_examples_pred() {
  DoubleList d = { NULL, 0 };
  list_append(&d, 68.5);
  list_append(&d, 72.0);
  list_append(&d, 0.5);
  list_append(&d, 99.0);
  // How to call list_filter for each? What's the expected result?



}
```