CSE 29
Lecture 19 Summary

March 10, 2026

📣 Logistical Things

- Your Assignment 4 Resubmission and Assignment 5 are due Friday, March 13th at 11:59pm

Handout Page 1

Review: `DoubleList`

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

typedef struct {
  double *data;
  int size;
} DoubleList;

void list_append(DoubleList *list, double val);
void list_print(DoubleList *list);

int main() {
  DoubleList temps = { NULL, 0 };

  list_append(&temps, 68.5);
  list_append(&temps, 72.0);
  list_append(&temps, 65.3);

  list_print(&temps);

  free(temps.data);
}
```

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

```c
void list_append(DoubleList *list, double val) {
  double *new_data = malloc((list->size + 1) * sizeof(double));
  memcpy(new_data, list->data, list->size * sizeof(double));
  list->data = new_data;
  list->data[list->size] = val;
  list->size++;
}
```

```
$ 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`

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




}
```

`list_append` — version 2, using `realloc`

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




}
```

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

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

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

Handout Page 2

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?


}
```

🧠 Review Questions (refer to handout page 1)

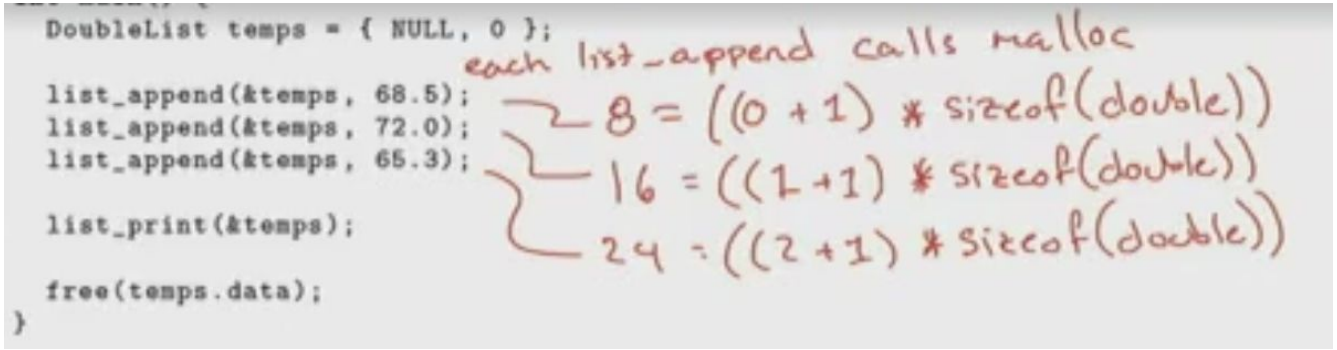
Q1: How many calls to malloc happen, with what sites, in the program labelled Review: DoubeList?

Q2: (skip)

Q3: Draw the picture showing the “24 bytes in 2 blocks lost”

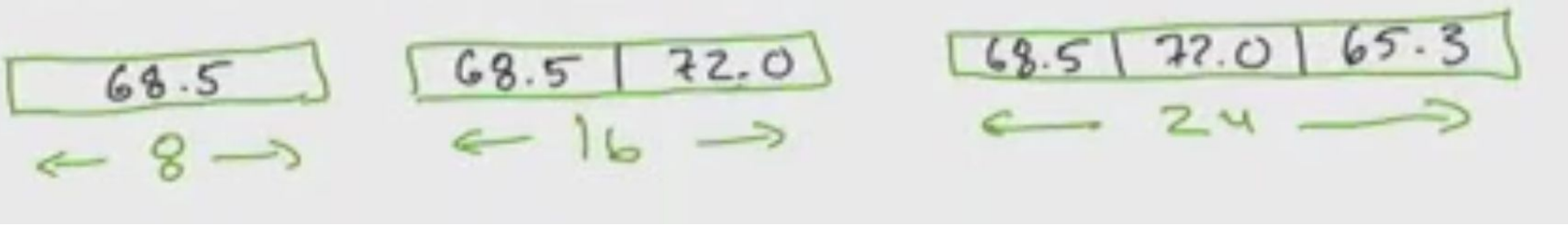
🧠 Review Question 1 Answer

1. Each call to `list_append` calls `malloc`, therefore there are 3 calls to `malloc`.

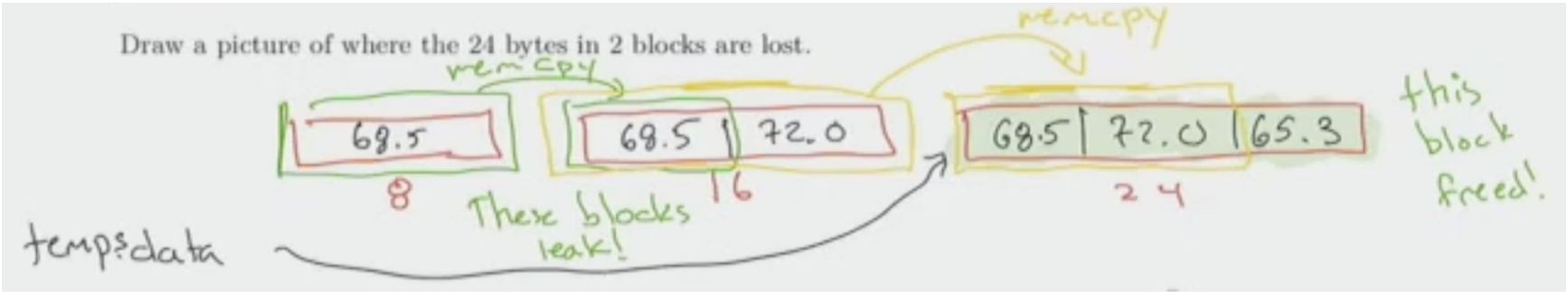


🧠 Review Question 3 Answer

After each `list_append` call:



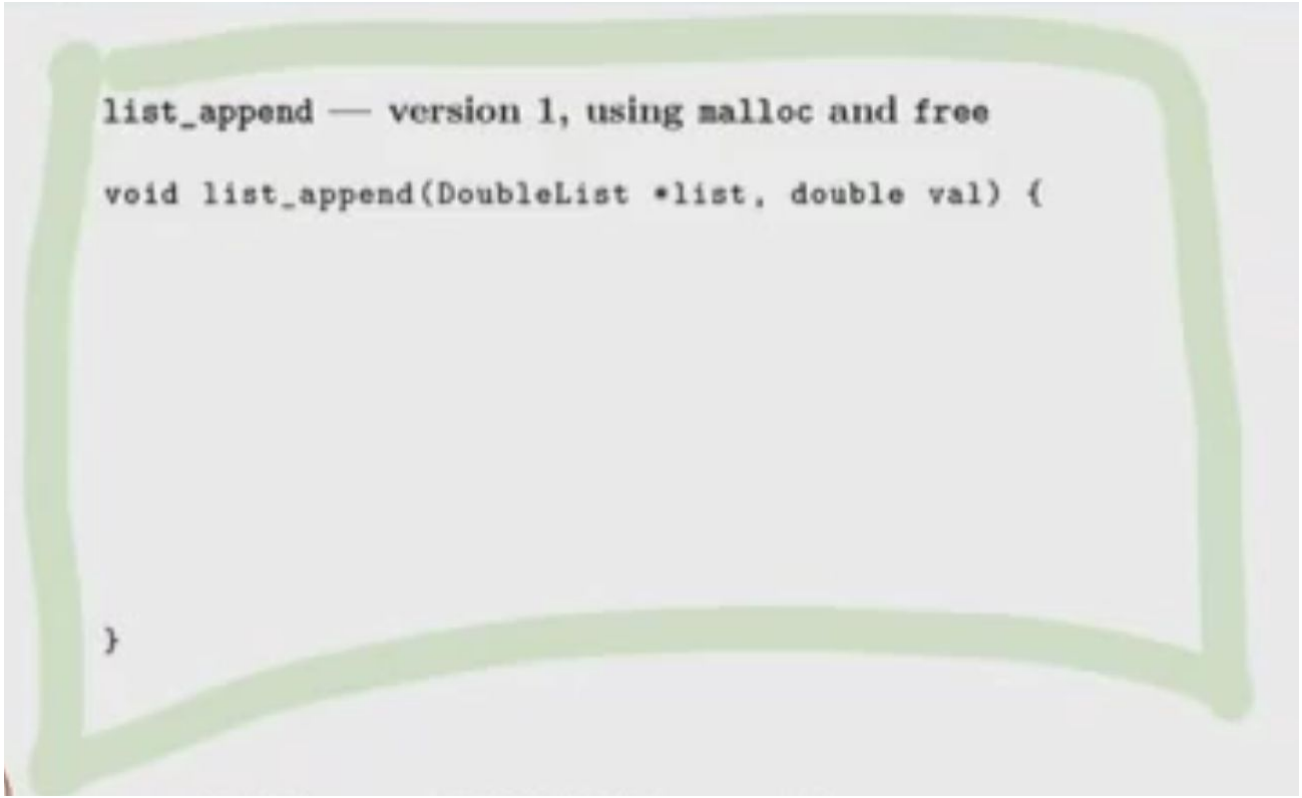
Only `temps.data` was freed at the end, but the previous `malloc`ed data was lost, because there was no longer a reference to it.



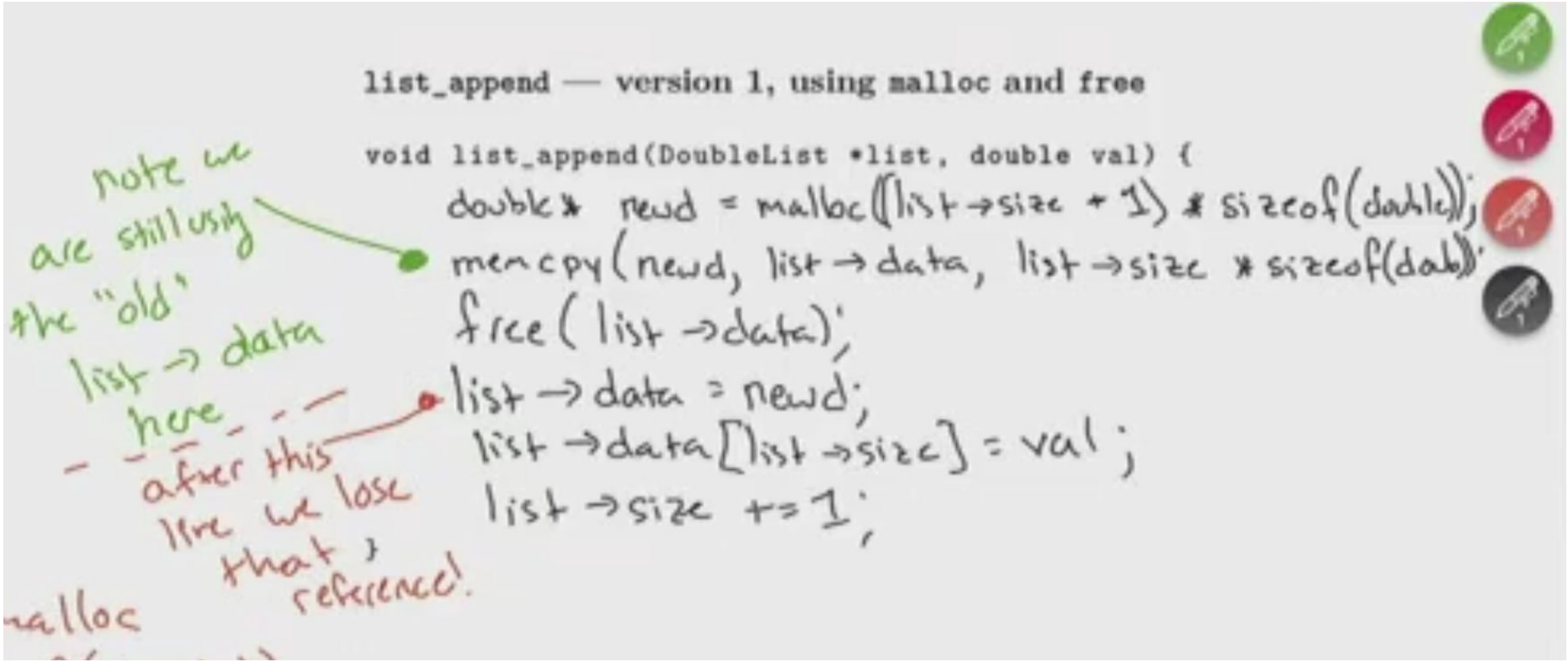
Questions

- Shouldn't `valgrind` be reported a size of 48 bytes lost because of the block headers?
  - If I run this on my mac, ieng6, or somewhere else, the implementation of `malloc` may differ
  - Your implementation of `vmalloc` in PA4 is different from other implementations of `malloc`
  - The other implementations have different metadata, which may or may not be stored next to the block, as your PA4 implementation did
  - **However**, for every implementation, `valgrind` will report bytes in terms of what the user allocated or free'd. If we `malloc` 9 bytes, for example, `valgrind` will report 9 bytes being lost
- Why does `valgrind` report 4 mallocs and 2 frees?
  - In my function `list_print` which is not shown, it calls `printf` which does an internal allocation and then frees it
- What do lines 10 and 11 do?
  - I declared the functions before I defined them, so that other functions could use them earlier in the file before my definitions (what a header file does!!)

Try fixing `list_append` using `malloc` and `free`!



Answer – only need to add one additional line!



realloc

`void *realloc(void *ptr, size_t size)`:

The realloc() function tries to change the size of the allocation pointed to by ptr to size, and returns ptr.

If there is not enough room to enlarge the memory allocation pointed to by ptr, realloc() creates a new allocation, copies as much of the old data pointed to by ptr as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory.

If ptr is NULL, realloc() is identical to a call to malloc() for size bytes.

Fixing `list_append` using `realloc`!

list_append — version 2, using realloc

```
void list_append(DoubleList *list, double val) {
    double* newd = realloc(list->data,
                           ((list->size+1) * sizeof(double));
    list->data = newd;
    list->data[list->size] = val;
    list->size += 1;
}
```

realloc does malloc + memcpy + free

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)`

Questions

- Can malloc blocks be non-contiguous?
  - No
- Can you put the `free` before the `memcpy` in `list_append` (v1)?
  - Probably, but when you free the block the payload may be modified, so the data won't be guaranteed to be the same
- How does `realloc` know how much data to copy (the size of original ptr)?
  - `realloc` utilizes the same metadata `malloc` uses

Filtering a DoubleList

Try describing these functions with one sentence

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 }
```

Joe's Descriptions

```
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 }
```

in English!

Takes a DoubleList, returns a new DL with all elements less than 70

```
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 }
```

Takes a DoubleList returns a new DL with all elements between 0 and 1 (inclusive)

```
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 }
```

Takes a DoubleList returns a new DL with all elements that are exact integers.

Notice that the code is all the same, except for the predicate (what we are searching for)

It would be inefficient to have different functions for all filters (or create one when a new predicate arises)

How to abstract this based on the predicate?

Factoring out the pattern: list_filter

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

"a pointer to/addr of a function takes double returns int

```
DoubleList list_filter(DoubleList *list, Predicate pred) {
    DoubleList result = {NULL, 0};
    for(int i=0; i<list->size; i+=1) {
        if( (*pred) (list->data[i])) {
            list_append(&result, list->data[i]);
        }
    }
    return result;
}
```

Example of usage

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?
  DoubleList less70s = list_filter(&d, &is_less_70);
  DoubleList units = list_filter(&d, &is_unit);
}
```

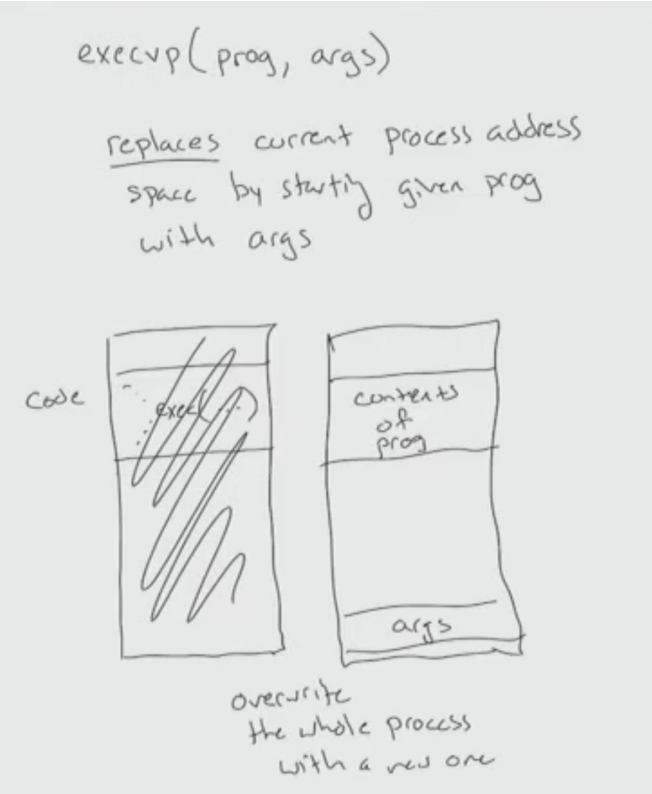
Questions

- Why do we need the typedef?
  - Defines a function pointer type that returns an integer and takes a double parameter called Predicate
  - Instead of `Predicate`, we could just type out `int (*predicate)(double)`, but it's more convenient to create a type
    - How `request_handler` works in PA5!
- Are function pointers the same as a lambda function?
  - Anonymous functions - declaring a function without a name
  - Can be used to passed in as a function parameter, but it does the same thing as declaring a function, just does not give a name
- Note: all programming languages have *some* way to pass a function as a parameter

CSE 29 Review

exec

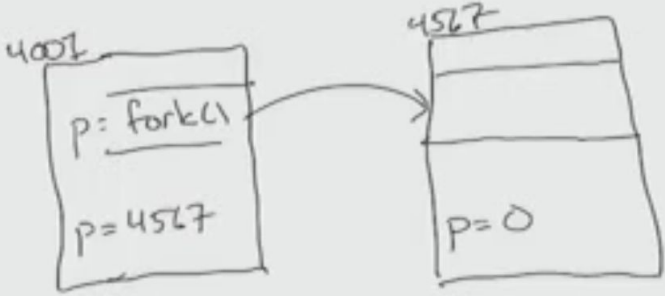
After execvp is called, the running process is overwritten with prog



fork The OS assigns PIDs to processes

int fork()

copies current process address space and runs both processes, returns 0 in child, pid in parent



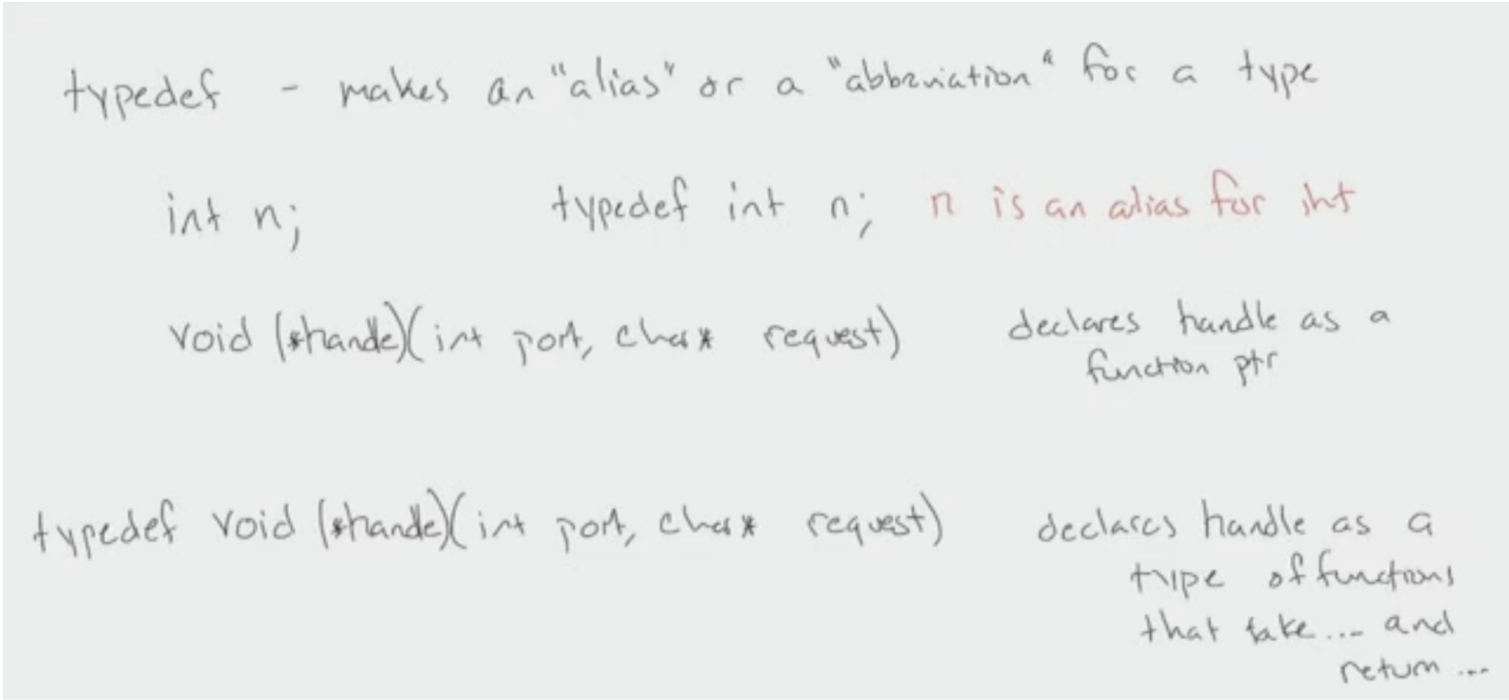
wait(int* status)

- returns when any child process finishes

waitpid(int pid, int* status)

- returns when pid finishes

typedef



Similar to declaring a variable, but instead declaring a type (a variable that represents a type value)

Runtime Environment

- A runtime environment is a program that is running alongside your program, helping manage your program
  - Reading and writing to files from your program is an example of something a runtime environment would manage

Joe's Notes

1. How many calls to malloc happen, with what sizes, in the program labelled Review: DoubleList?
2. (skip)
3. Draw the picture showing the "24 bytes in 2 blocks lost"

REFER TO HANDOUT!

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 }
```

each list_append calls malloc

8 = ((0 + 1) * sizeof(double))

16 = ((1 + 1) * sizeof(double))

24 = ((2 + 1) * sizeof(double))

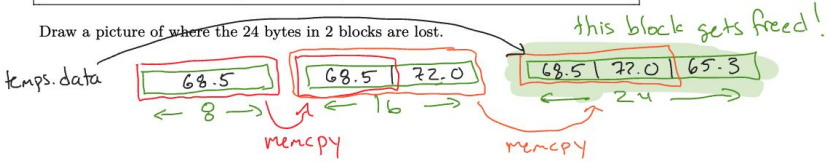
---

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) {
  double* newd = malloc((list->size + 1) * sizeof(double));
  memcpy(newd, list->data, list->size * sizeof(double));
  free(list->data);
  list->data = newd;
  list->data[list->size] = val;
  list->size += 1;
}
```

note we are still using the "old" list->data here - after this line we lose that reference!

list_append — version 2, using realloc

```
void list_append(DoubleList *list, double val) {
  double* newd = realloc(list->data,
                         ((list->size + 1) * sizeof(double)));
  list->data = newd;
  list->data[list->size] = val;
  list->size += 1;
}
```

realloc does malloc + memcpy + free

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 - describe what these 3 do in English!

```
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 }
```

Takes a DoubleList, returns a new DL with all elements less than 70

```
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 }
```

Takes a DoubleList returns a new DL with all elements that are exact integers.

```
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 }
```

Takes a DoubleList returns a new DL with all elements between 0 and 1 (inclusive)

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) {
    DoubleList result = {NULL, 0};
    for(int i=0; i< list->size; i+=1) {
        if( (*pred) (list->data[i])) {
            list_append(&result, list->data[i]);
        }
    }
    return result;
}
```

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?
  DoubleList less70 = list_filter(&d, &is_less_70);
  DoubleList units = list_filter(&d, &is_unit);
}
```

Passing a function as a value

Filtering a DoubleList 1-sentence English description of each

```
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 }
```

Takes a DoubleList, returns a new DL containing all the elements less than 70

```
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 }
```

Takes a DoubleList returns a new DL containg all the elements that are exact integers

```
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 }
```

Takes a DoubleList returns a new DL containg all the elements that are between 0 and 1 (inclusive)

What's the same across all three functions?

What's different?

The function (predicate) that is called

Factoring out the pattern: list_filter

```
typedef int (*Predicate)(double);
DoubleList list_filter(DoubleList *list, Predicate pred) {
    DoubleList result = { NULL, 0 };
    for(int i=0; i<list->size; i+=1) {
        if( (*pred)(list->data[i]) ) {
            list_append(&result, list->data[i]);
        }
    }
    return result;
}
```

"a pointer to/addr of a function takes double returns int"

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?
  DoubleList lessTos = list_filter(&d, &is_less_70);
  DoubleList units = list_filter(&d, &is_unit);
}
```

Passing functions as values!
"Higher-order functions"

Review: DoubleList

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

typedef struct {
  double *data;
  int size;
} DoubleList;

void list_append(DoubleList *list, double val);
void list_print(DoubleList *list);

int main() {
  DoubleList temps = { NULL, 0 };

  list_append(&temps, 68.5);
  list_append(&temps, 72.0);
  list_append(&temps, 65.3);

  list_print(&temps);

  free(temps.data);
}
```

8 = ((0+1) * sizeof(double))
16 = ((1+1) * sizeof(double))
24 = ((2+1) * sizeof(double))

---

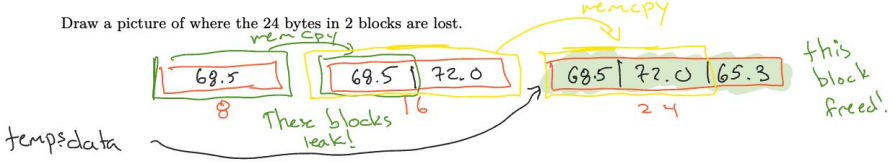
list_append — version 0 (has a bug!)

```c
void list_append(DoubleList *list, double val) {
  double *new_data = malloc((list->size + 1) * sizeof(double));
  memcpy(new_data, list->data, list->size * sizeof(double));
  list->data = new_data;
  list->data[list->size] = val;
  list->size++;
}
```

changes temp.data

```
$ 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

```c
void list_append(DoubleList *list, double val) {
  double* newd = malloc(....);
  memcpy(newd, list->data, list->size * sizeof(double));
  free(list->data);
  list->data = newd;
  list->data[list->size] = val;
  list->size += 1;
}
```

still using list->data

losing the reference to list->data

list_append — version 2, using realloc

```c
void list_append(DoubleList *list, double val) {
  double* newd = realloc(list->data,
                         ((list->size+1) * sizeof(double)));
  list->data = newd;
  list->data[list->size] = val;
  list->size += 1;
}
```

uses header/malloc metadata to decide how much to memcpy

realloc wraps malloc + memcpy + free into one call:

```c
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 1-sentence English description of each

```
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 }
```

Takes a DoubleList, returns a new DL containing all the elements less than 70

```
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 }
```

Takes a DoubleList returns a new DL containg all the elements that are exact integers

```
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 }
```

Takes a DoubleList returns a new DL containg all the elements that are between 0 and 1 (inclusive)

What's the same across all three functions?

What's different?

The function (predicate) that is called

Factoring out the pattern: list_filter

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

"a pointer to/addr of a function takes double returns int"

```
DoubleList list_filter(DoubleList *list, Predicate pred) {
    DoubleList result = { NULL, 0 };
    for (int i = 0; i < list->size; i += 1) {
        if ( (*pred) (list->data[i]) ) {
            list_append(&result, list->data[i]);
        }
    }
    return result;
}
```

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

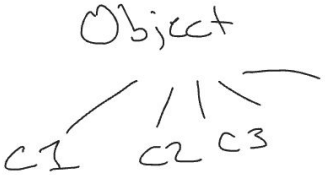
```
void list_examples_pred() {
  DoubleList d = { NULL, 0 };
  list_append(&d, 68.5);
  list_append(&d, 72.0);
  list_append(&d, 0.5);
  list_append(&d, 99.0);
  // How to call list_filter for each? What's the expected result?
  DoubleList less70s = list_filter(&d, &is_less_70);
  DoubleList units = list_filter(&d, &is_unit);
}
```

Passing functions as values!

"Higher-order functions"

"Dynamic Dispatch"

```
void doStrgStuff(Object[] objs) {
    ...
        obj[i].toStrg()
    ...
}
```



"lambda"

λ

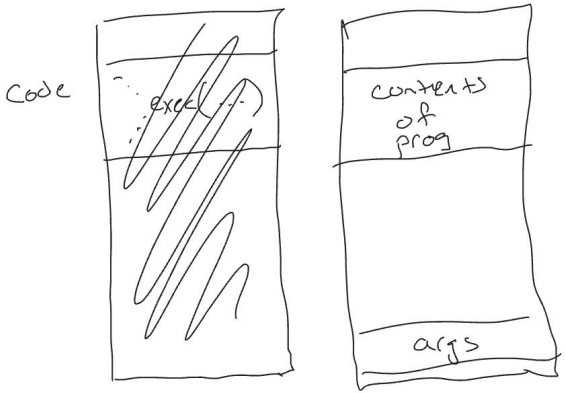
"anonymous functions"

```
def less70(n): return n < 70
filter(lst, less70)
```

```
filter(lst, lambda n: n < 70)
```

execvp(prog, args)

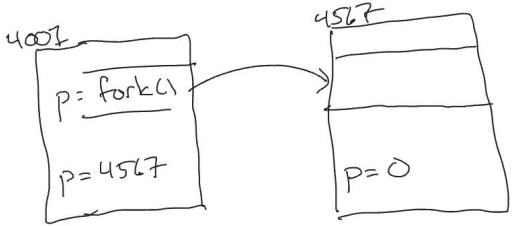
replaces current process address space by starting given prog with args



overwrite the whole process with a new one

int fork()

copies current process address space and runs both processes, returning 0 in child, pid in parent



wait(int* status)

- returns when any child process finishes

waitpid(int pid, int* status)

- returns when pid finishes

typedef - makes an "alias" or a "abbreviation" for a type

```
int n;
```

```
typedef int n;
```

n is an alias for int

```
void (*handle)(int port, char* request)
```

declares handle as a function ptr

```
typedef void (*handle)(int port, char* request)
```

declares handle as a type of functions that take ... and return ...

"Runtime environment"