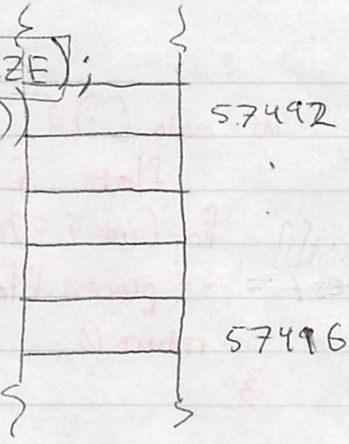
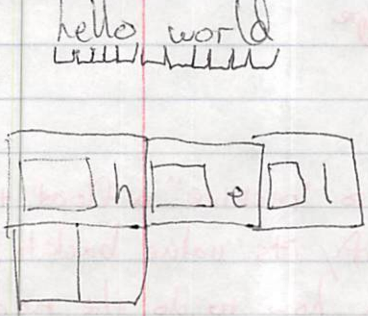
2016- Aug, Sept

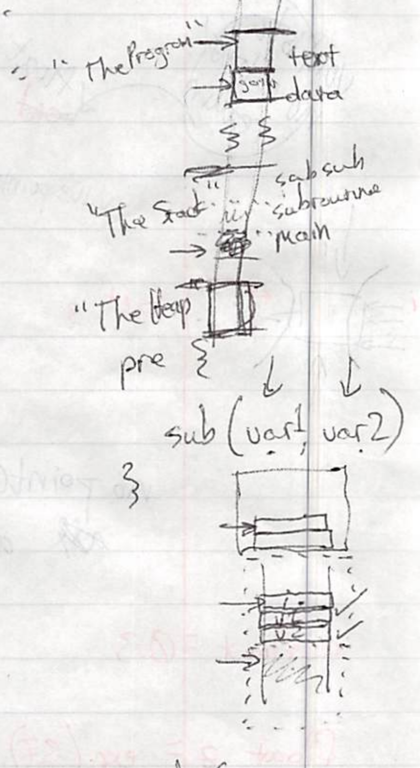
✓SCANNED

(01)

```
char myString = "gotta gotta get up..."
stack → int arr[ARR_SIZE];
heap → int *arr = malloc(sizeof(int)*ARR_SIZE);
arr[i] === *(arr + i*sizeof(int))
"new"
int *arr = new int[100];
delete arr;
free(arr);
```







```
int *arr = malloc/new

&arr === address on the stack

arr === address on the heap

*arr === the int I wanted to store

&(*arr) === arr

*arr === arr[0]

("built-in")
*(arr + i*sizeof(int)) === arr[i]
datatype• example
i.e. if {int *arr or int arr[SOME_SIZE]}
```

sub {

int i

for i = 0 i++

ssub

DON'T FORGET

*(arr) === arr[0] array is pointer to first element in array

int var;

*var = 10;

float new_val();

(void)

$$\begin{bmatrix} 5.7 \\ 2 \\ -4 \end{bmatrix}$$

return [out 1, out 2]

$g(x,y,z) \pm f(x,y,z) = h(x,y,z)$

stuff1, stuff2 = func()

```
int main () {
        float f_list [100]; test;   //load up f_list w/ int vals
    for (int i = 0; i < 100; i++)
f_list[i]
~~test~~ = process_floats (&(f_list[i])                    );   // how to get proc-floats
                                                                     to mod vals in f_l[]
    return 0;                                                                         ???
}
```

it doesn't matter @ memory address is int; it matters @ pointer is pointing @ value w/ "float" datatype

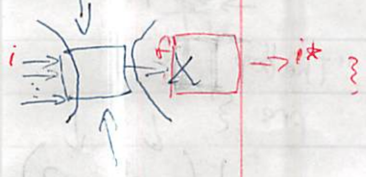
~~memory address~~ is ~~simple "int"~~

* ⇒ pointer

```
        float
~~void~~ process_floats (~~[illegible]~~ * a_float ______ ){ // how to "receive" a float to
                                                    modify its value back in main
   value pointed to by a_float = new_val()
        *a_float   =   new_val();                   // then, how to do the mod

   return
```

void b/c no output/return



}

no pointers* no references &
~~all~~ all values passed as copies.
maintain current flow

f_list[i] = process_floats(f_list[i]);

a_float = 0.3

float z = exp(3.7)

a_float = ~~*~~ exp(3.7)

```
int ~~a~~ a = 10 ;
int d = 5 ;
~~int~~ a = a + d ;
~~a = c~~;
```

assign (& a, add(a, d));

TLAP pg 62: finding the mode

- mode: value ⓣ appears most often.

survey responses: array of values btwn 1 & 10

→ sort answers

→ find mode (if multiple, only select one)

~ ~ ~

✓ define array → unsorted answers → 27 responses

✓ → sort array ✓ → create function ⓣ (tallies) counts each response type

→ Create function to determine mode.

→ stop when finished, tell me what it is.

later, clean up code so ⓣ each segment lives in different files

simplified mode func

→ tally how many of each # there are.

```
for (i=0; i < A_LENGTH; i++) {
    ~~int j = 0; 1;~~
    ~~int runningTally = 1;~~
    int runningComparison = 1;
```

[ones or twos or threes ~~[illegible]~~]

```
mode ⇒ value ⓣ appears most often
  ~~int tallieVar = 0;~~
    ~~if ones > tallieVar & &&~~
       ~~tallieVar = ones;~~
    ~~} else if twos > tallieVar {~~
       ~~tallieVar = twos;~~
```

```
int tallyVar = 0;
if ~~ones > tall~~ ones > twos {
    tallyVar = ones;
} else {        //(if twos > ones)
    tallyVar = twos;
}
```

ONES = 0
TWOS = 0
THREES = 0
FOURS = 0

runningQuota = 0

```
    0 1 2 3 4 5 6
a= [1,1,2,3,4,4,4]   tally each #
  runningtally = 1
if a[i] == runningtally
   runningQuota ++;
else
     runningtally ++;
```

a[0] == rt? => 1 == 1? yes! => rQ++ => 1
a[1] == rt? => 1==1? yes => rQ++ => 2
a[2] == rt? => 2 == 1? no! => rt++ => 2

somehow I want a(2) to be compared again!

```
counter=1
if a[i] == counter
     ONES ++;
```

```
if a[i] == 1
     ones ++;
elsif a[i] == 2
     twos ++;
elsif a[i] == 3
     threes ++;
elsif a[i] == 4
     fours ++;
else
     cout << "unexpected error\n";
```

func
need pointers?
need return?
need new library?

error: invalid types 'int[int]' for array subscript

```
} else if array[i] == 2) {
```

something about array, subscript, int, or something similar

& &

over-simplistic mode w/ only 2 variabl.

```
int tallyVar = 0;
if ones > twos {
    tallyVar = ones;
} else      //(if ones ≤ twos)
    tallyVar = twos;
}
```

```
//use this tallyArray to help make
// mode code more universal:

tallyArray[ARRAY_LENGTH]
  function @ sets each
   element to zero!
//use tallyArray element & index to
// keep track of mode counting :)
```

the length of ~~arra~~ tallyArray is equal to the ~~#~~ answer # range (1 to 5, 1 to 10, etc), not the length of SurveyResponses (19).
> create second constant which is the length of survey responses

int*array → passing whole array → should point to address of starting point and then add the length ~~nees~~ needed & then do ~~thw~~ things to the value of @ location. Plus, arrays are effectively pointers by their very existence and function.

int location → why isn't this a pointer? I suppose it represents a single value (in this case, a variable @ means "5"), not an entire array. But surely there's more to it than that. "Length" ~~use~~ gets used in a different tier of the func than * array does → could @ have an impact? It seems unrelated to the core logic and if anything just an extension of how the functionality works or something...

if don't want to alter Parent, create unlinked Child (not pointer)

```
a=[1,3,2,2,3,3];
tA=[0,0,0];
```

```
for (i=0; i<length; i++) {
  if a[i] == i+1
      tallyA[i] ++;
        :
}
```

```
{
tally Index = 1
if a[i] == 1
   tA[0]++;          tA[1-1] ++;
elseif a[i]==2
   tA[1] ++;         tA [2-1]++;
else if a[i]==3
   tA[2] ++;         tA [3-1] ++;
else
   cout << "bork!";
}
```

```
~~tA.length~~
x = tA.length
y = x-(x-1)
  tA[y+1] ++;
```

as the index of array iterates, then check to see what the value equals and then increment the corresponding tallyarray. My for loop will iterate through each value of array, one by one in order. So I know that each value of array will be examined on its own in order.

Take the first element of array → a[0], which equals 1. I then need to increment tallyarray[0] by one → not b/c ~~tA[0]~~ the index of a=0 but b/c the value is 1. So, I guess another way to say it is @ tA[1-1]++. If a[0]=1, then tA[1-1]++. How do I represent that w/ variables instead of literals?! How do I make it more generic?

~~tA[a[i]]~~

utilize tA.length and maybe a.length? in making the variable tallies?

if a[i] < tA.length & > 0

```
(      element, index) {
       a[i]
          tA[e-1] ++;
          ~~tA[~~
```

```
a[i]
[ item = a[i];
  tA[item-1] ++;
```

```
tA[a[i]-1]++;
```

```
a. each_with_index {|item, index|
     a[i] == item
}    i == index
```

```
→     if (some check on a[i])
★★★      tA[?] ++;
→     else
→        cout << "bark!\n";
```

```
tA[x]++;
```

```
for (i=∅; i<L; i++)
   if a[i] checks out
      tA[_ . . .
```

what should 'x' be?

i : the index i.e. how far through 'a'

a : the array

a[i] : the element we are working on

L : max(i) + 1

tA : 'tally Array' - each element a running total

```
for(i=0; i<length; i++) {
~~tA[i]~~ modeCheck=0;
  if tA[i] > modeCheck
      modeCheck = tA[i];
  else
```