CSE 29
Lecture 11 Summary

February 10, 2026

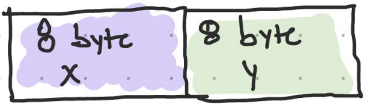
📣 Logistical Things

- Grades for Assignment 2 are released
  - If you would like to submit a regrade request, please do so on the grade summary Gradescope assignment following the instructions on the Piazza post @193
- Assignment 3 and Assignment 2 Resubmit are due this Thursday!
  - If you’re having trouble, please come to office hours!! We’re here to help :)

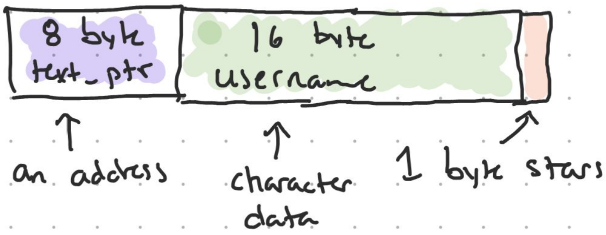
Hints/extra info in speaker notes

🧠 Struct Definitions for Review Questions

```
struct Point {
    double x;
    double y;
}
```



```
struct YelpReview {
    char* text_ptr;
    char username[16];
    uint8_t stars;
}
```

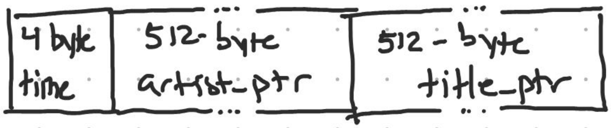


doubles are 8 bytes
pointers are 8 bytes
username is declared as being 16 characters long, making it 16 bytes.
Regardless of how long the actually username is
uint8_t is 1 byte (8 bits)
uint16_t is 2 bytes (16 bits)

🧠 Review Question 1

Q1: Fill in the struct definition

```
struct Song {


}
```



🧠 Review Question 2 & 3

Q2: Draw the picture!

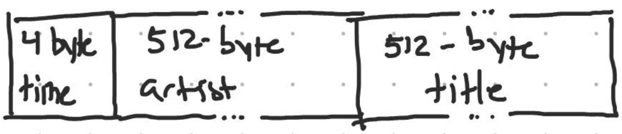
```
struct LineSegment {
    Point start;
    Point end;
}
```

Q3: What is sizeof for each struct above?

🤓 Review Question 1 Answer

Q1: Fill in the struct definition for a Spotify song

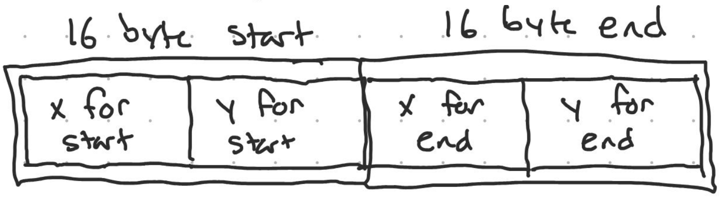
```
struct Song {
    uint32_t time;
    char artist[512];
    char title[512];
}
```

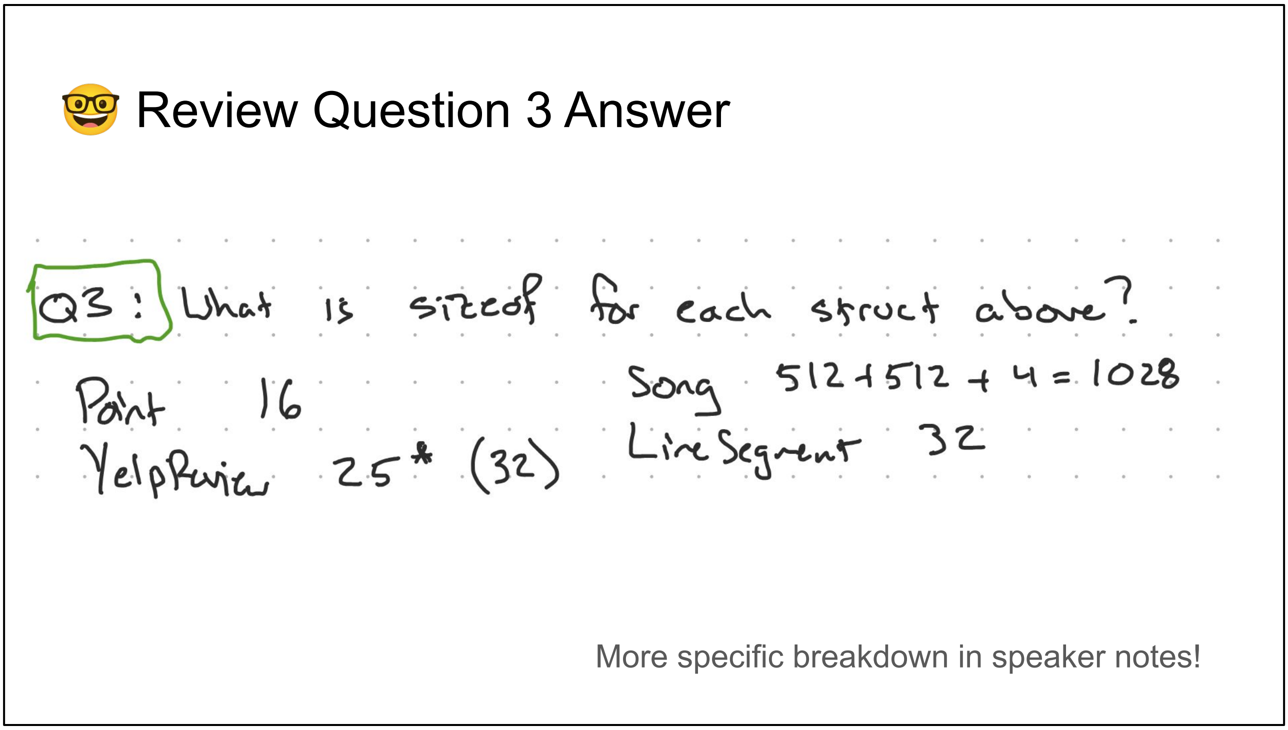


🤓 Review Question 2 Answer

Q2: Draw the picture!

```
struct LineSegment {

    struct Point start;
    struct Point end;
}
```





🤓 Review Question 3 Answer

Q3: What is sizeof for each struct above?

Point 16

YelpReview 25* (32)

Song 512+512 + 4 = 1028

LineSegment 32

More specific breakdown in speaker notes!

Q3:

sizeof(struct Point) = 16; // 8 bytes for x and 8 bytes for y

sizeof(struct YelpReview) = 25; // 8 bytes for text_ptr, 16 bytes for username and 1 for stars

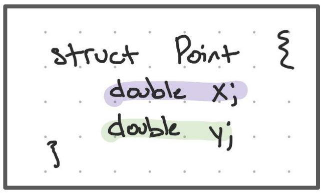
sizeof(struct Song) = 1028; // 4 bytes for time, 512 bytes for artist, and 512 bytes for title

sizeof(struct LineSegment) = 32; // 16 bytes for start and 16 bytes for end

structs!

What is a struct?

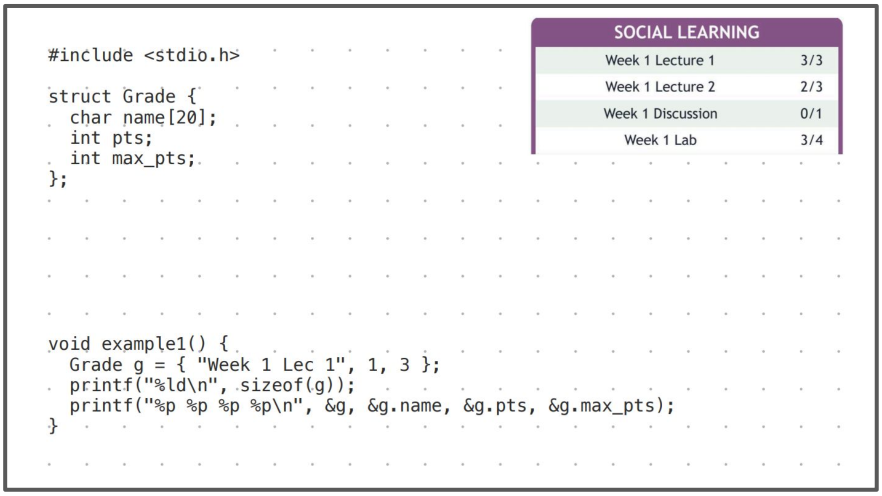
- **A struct describes a multi-value memory layout**
  - If we are thinking in terms of Java, a struct is like an object
  - However, C is not an Object Oriented Programming language, so this is not an object
- Structs have a name and members/fields
- Structs are defined like this



```
struct Point {
    double x;
    double y;
}
```

- The struct definition tells the compiler the “shape” of memory that is needed for the struct
- Any empty space between values is generally initialized to all 0s

Let’s look at the top of the handout

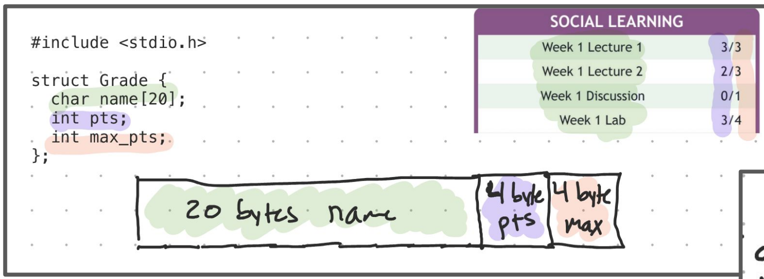


```
#include <stdio.h>

struct Grade {
  char name[20];
  int pts;
  int max_pts;
};




void example1() {
  Grade g = { "Week 1 Lec 1", 1, 3 };
  printf("%ld\n", sizeof(g));
  printf("%p %p %p %p\n", &g, &g.name, &g.pts, &g.max_pts);
}
```

SOCIAL LEARNING	
Week 1 Lecture 1	3/3
Week 1 Lecture 2	2/3
Week 1 Discussion	0/1
Week 1 Lab	3/4

How many bytes does struct Grade use and how does it look in memory?

How struct Grade looks in memory

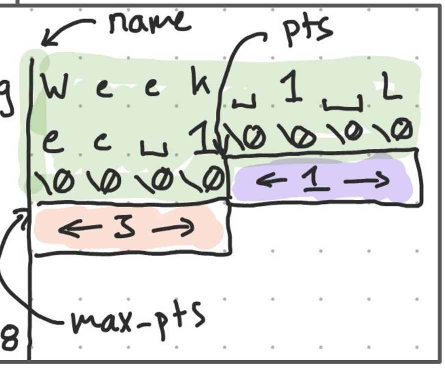


```
#include <stdio.h>

struct Grade {
  char name[20];
  int pts;
  int max_pts;
};
```

SOCIAL LEARNING	
Week 1 Lecture 1	3/3
Week 1 Lecture 2	2/3
Week 1 Discussion	0/1
Week 1 Lab	3/4

Amount of bytes

In memory



What would example1() print?

Answers on next slide

```
void example1() {
  Grade g = { "Week 1 Lec 1", 1, 3 };
  printf("%ld\n", sizeof(g));
  printf("%p %p %p %p\n", &g, &g.name, &g.pts, &g.max_pts);
}
```

Let's just say &g = 0xff…20

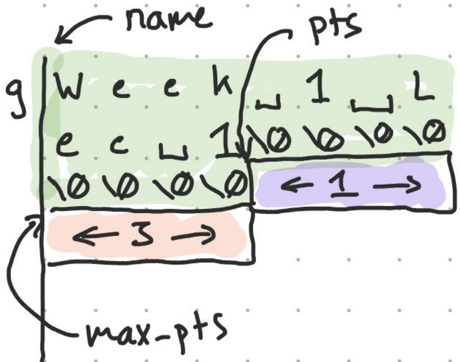
sizeof(g) = ________

&g = 0xff...20

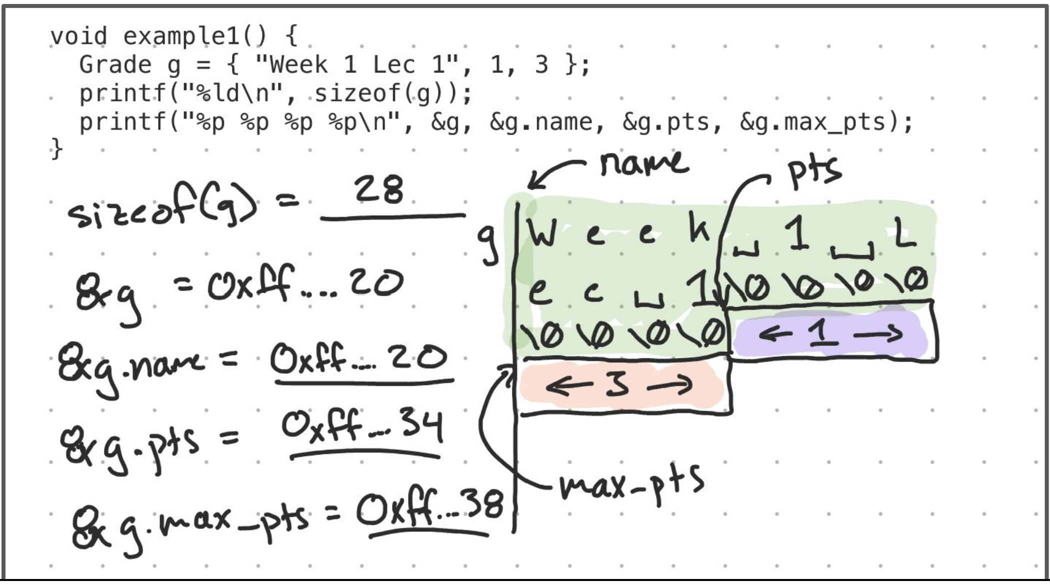
&g.name = ________

&g.pts = ________

&g.max_pts = ________



What would `example1()` print?



```
void example1() {
  Grade g = { "Week 1 Lec 1", 1, 3 };
  printf("%ld\n", sizeof(g));
  printf("%p %p %p %p\n", &g, &g.name, &g.pts, &g.max_pts);
}
```

sizeof(g) = 28

&g = 0xff... 20

&g.name = 0xff... 20

&g.pts = 0xff... 34

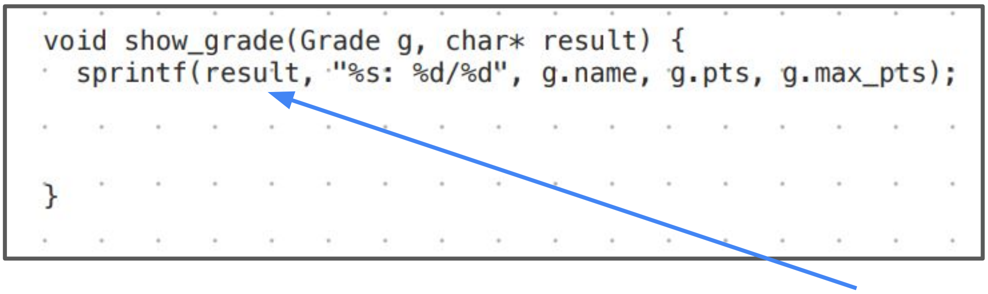
&g.max_pts = 0xff... 38

🤓 Side Note: Creating an **alias** for our struct

If we have the line `typedef struct Grade Grade;`

- This lets us write `Grade` instead of `struct Grade` for the variable type
- Instead of writing
  - `struct Grade g = {“Week 1 Lec 1”, 1, 3}`
- We can instead write
  - `Grade g = {“Week 1 Lec 1”, 1, 3}`
- The only thing this does is make it more convenient and let us not need the word “struct” in front of variables

Lets look at show_grade()



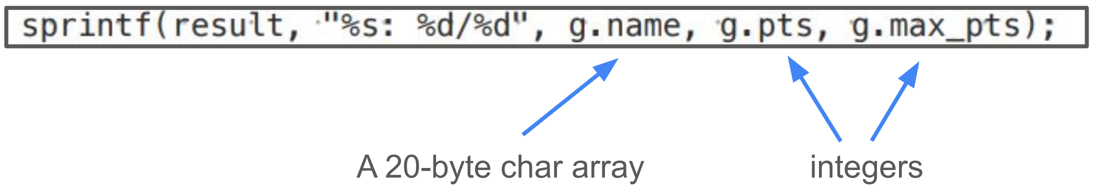
```
void show_grade(Grade g, char* result) {
  sprintf(result, "%s: %d/%d", g.name, g.pts, g.max_pts);


}
```

sprintf is like printf, but stores its printed output in the 1st arg (a char*) instead of stdout (printing to terminal)

- sprintf assumes that result is long enough and will write over arbitrary memory if we write longer

We can access fields by using the dot (.) [ex. g.name, g.pts]

Let’s look at the fields being accessed



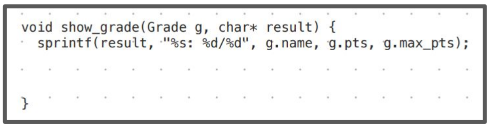
```
sprintf(result, "%s: %d/%d", g.name, g.pts, g.max_pts);
```

A 20-byte char array

integers

- Just like arrays, `char[]` in `Grade` “decays” into a `char*`
    - When we pass around an array, what is passed around is a pointer (`char*`) not an array
- The address gets passed around
- Just like when we pass an array defined in `main` to another function, what gets passed is the pointer of the array
- So therefore, `g.name` is a `char*`

Try it out: example2()

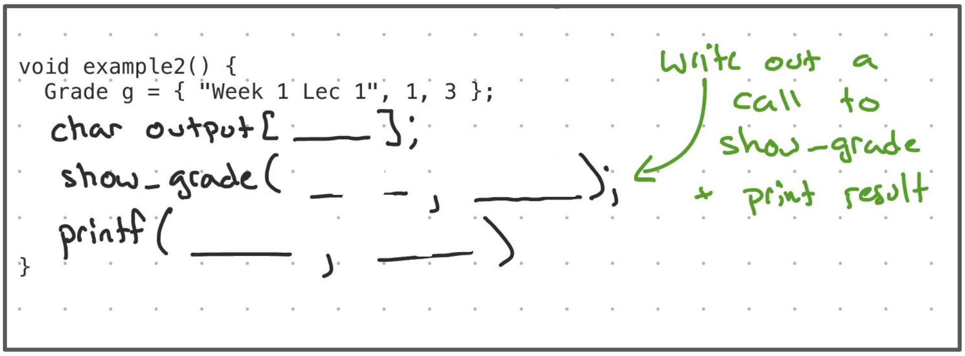


```
void show_grade(Grade g, char* result) {
  sprintf(result, "%s: %d/%d", g.name, g.pts, g.max_pts);


}
```

In example2(),

- Call show_grade()
- Print out the result

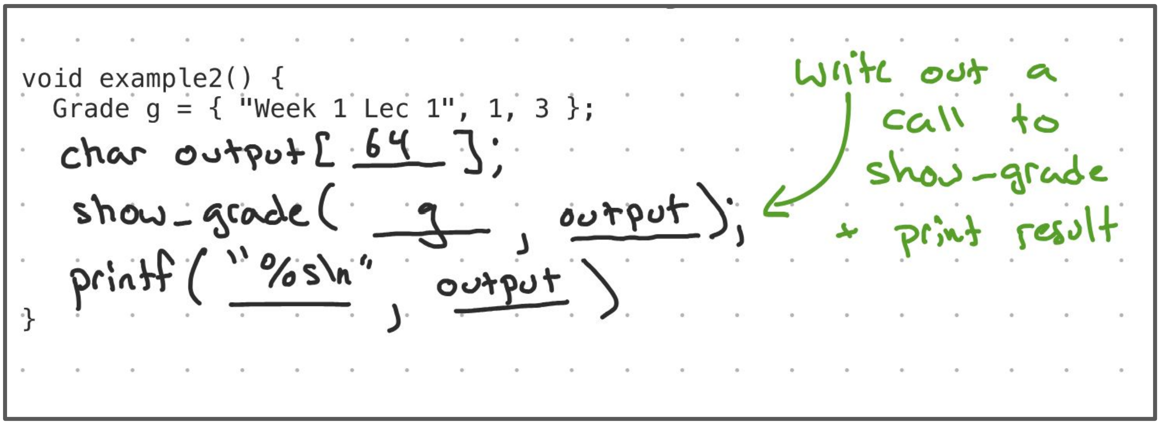
Try to fill in the blanks!



```
void example2() {
  Grade g = { "Week 1 Lec 1", 1, 3 };
   char output[ ____ ];
    show_grade( _ _ , _____ );
   printf( _____ , ____ )
}
```

write out a call to show_grade + print result

example2() Solution



```
void example2() {
  Grade g = { "Week 1 Lec 1", 1, 3 };
  char output[ 64 ];
  show_grade( g , output );
  printf( "%s\n" , output )
}
```

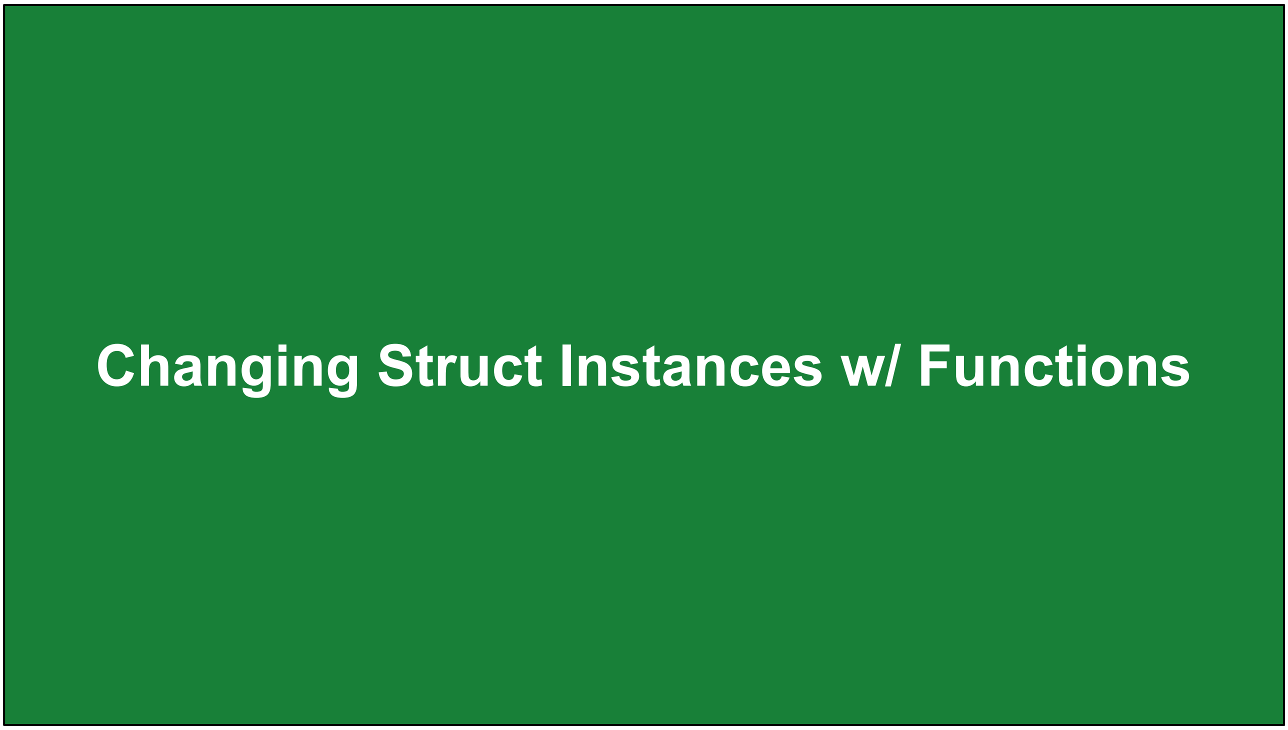
Write out a call to show_grade + print result

What if we did `output[32]` instead?

- There is a possibility that we run into an error
- When we print out numbers, %d could cause more than 4 bytes of output
  - 11 characters could be printed out for a 4 byte signed int
  - Ex. -1000000000 (negative one billion is 11 characters)
- If `g.pts` and `g.max_pts` are both 11 characters and the `g.name` is 20 characters long, this would be 42 characters long
  - This would NOT fit in `output[32]`
- It is best if we chose `output[64]`
- `sprintf()` is related to what is printed, not what is stored
  - We have to be careful when deciding the size of our output

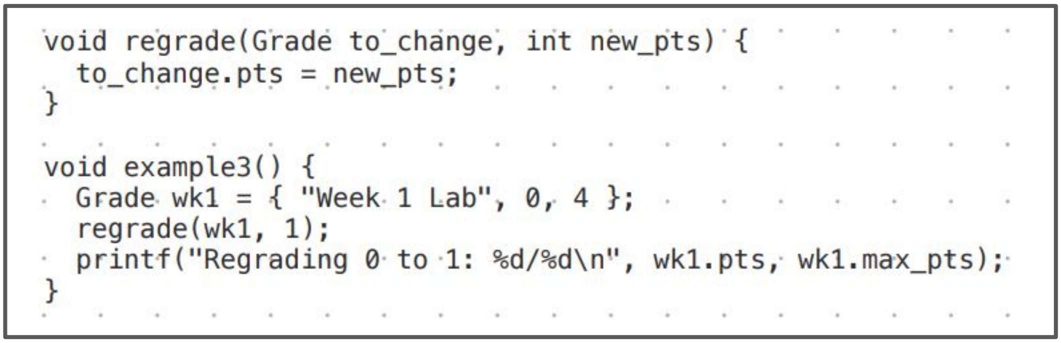
Questions

- Could you initialize output as a pointer instead? (What if it was written `char* output`)
  - Probably get a segfault
  - `char* output` would be initialized to `0` and then when something writes to it, we would be writing to 0x0 (NULL). This is memory we can't modify, so this would result in a segfault
  - No memory was given to it



Changing Struct Instances w/ Functions

Let’s say we want to regrade something



```
void regrade(Grade to_change, int new_pts) {
  to_change.pts = new_pts;
}

void example3() {
  Grade wk1 = { "Week 1 Lab", 0, 4 };
  regrade(wk1, 1);
  printf("Regrading 0 to 1: %d/%d\n", wk1.pts, wk1.max_pts);
}
```

What would print is “`Regrading 0 to 1: 0/4`”

- It didn’t work! Why not? (next slide)

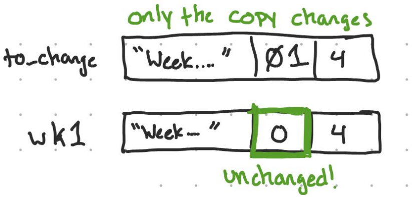
Why didn’t `regrade()` work?

It didn’t work because **struct instances DON’T act like arrays when used as function arguments**

struct instances are **copied** when passed as arguments

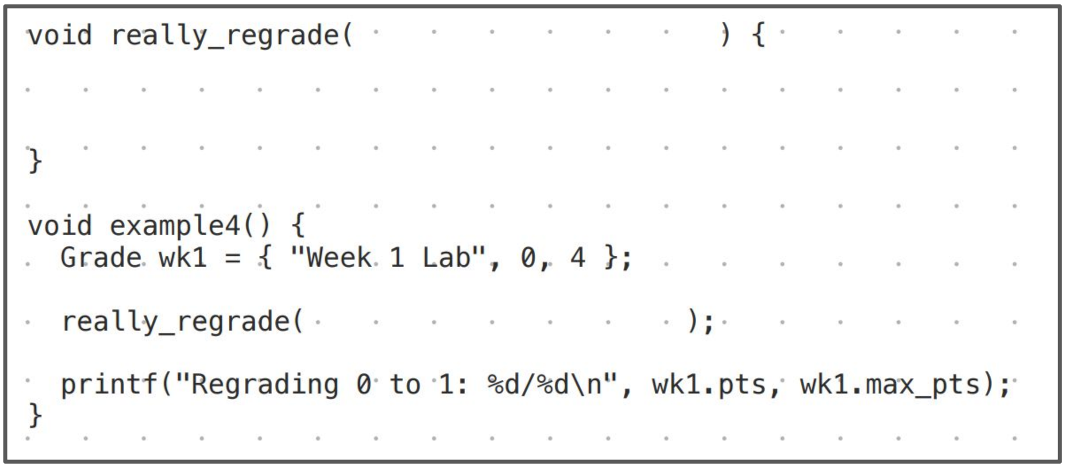
- This is *unlike* arrays
- This is more like pass-by-value and not pass-by-reference since its a copy

`regrade()` doesn’t work because it doesn’t actually the struct instance



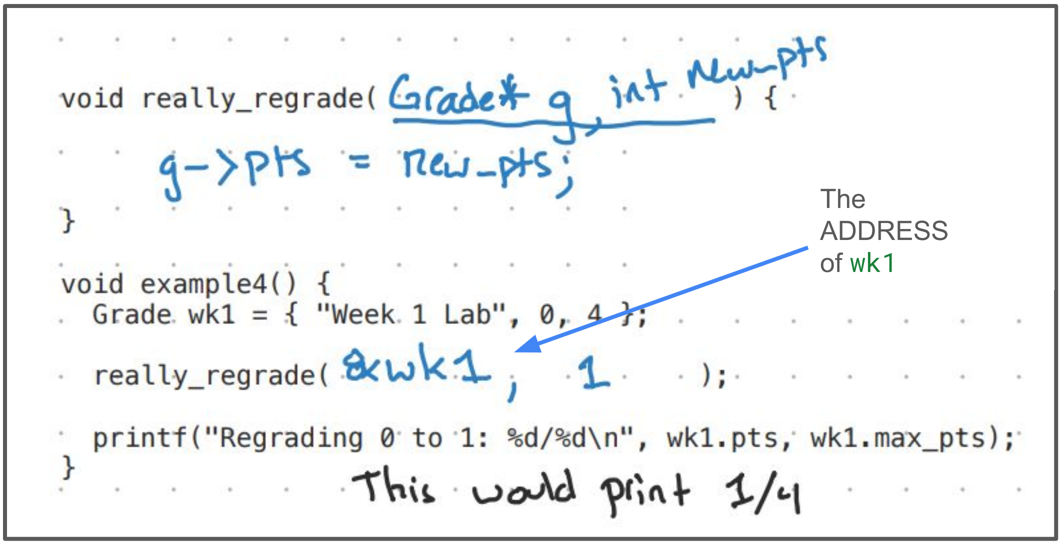
Let’s fix it with `really_regrade()`

Try writing the parameters and body of `really_regrade()`



```
void really_regrade(                  ) {


}

void example4() {
  Grade wk1 = { "Week 1 Lab", 0, 4 };

  really_regrade(                  );

  printf("Regrading 0 to 1: %d/%d\n", wk1.pts, wk1.max_pts);
}
```

really_regrade() Solution



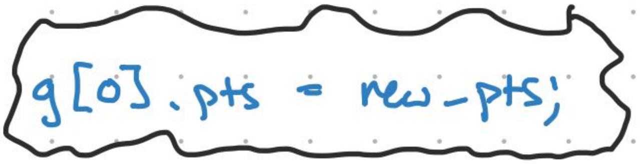
```
void really_regrade( Grade* g, int new_pts ) {
    g->pts = new_pts;
}

void example4() {
  Grade wk1 = { "Week 1 Lab", 0, 4 };

  really_regrade( &wk1, 1 );

  printf("Regrading 0 to 1: %d/%d\n", wk1.pts, wk1.max_pts);
}
```

This would print 1/4

The ADDRESS of wk1

Weird `really_regrade()` Solution

The body of `really_regrade()` could also be this line



```
g[0].pts = new_pts;
```

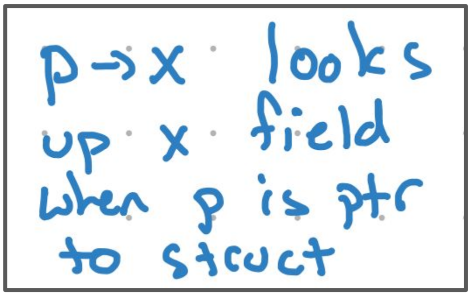
(g is a pointer to a variable of type Grade, therefore it can be accessed at the first index of g)

more to come on the `*` operator later!

We can even do `(*g).pts = new_pts;`

Arrow Operator (->)

With structs, we can use the arrow operator to **access the field of a struct through a pointer** (dereferencing)



p->x looks
up x field
when p is ptr
to struct

(p->x = v is called “dereferencing assignment”; v is a value)

Questions

- Would modifying the string in the original struct modify what’s in the copy?
  - Depends on if we initialized it as a `char[]` or `char*`
  - `char*` would just pass the pointer to the same memory address
  - `char[]` would copy the char array

The & Operator

& operator: How we can and can’t use it

- The & operator only works on left-hand side values (or LHS-values/L-values)
  - L-values: Expressions representing or evaluating to an assignable address
  - Left-hand side of the equals sign
  - There’s no way to ask for the address of `3+x`

Can Do

`&x`	`x` is a variable
`&a[n]`	`a` is an array or ptr
	`n` is an index
`&g.name`	`g` is a struct instance
	`name` is a field of g

Cannot Do

`&(3+x)`	
`&(f())`	`f()` is a function

Kind of like when we are working with variables

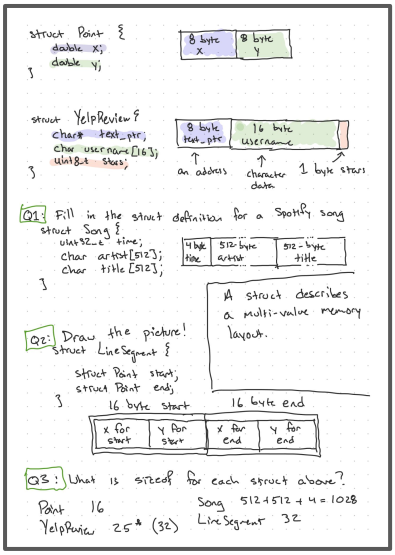
Can Do

```
x = v
a[n] = v
g.name = v
```

Cannot Do

```
3 + x = v
f() = v
```

Joe’s Notes (11am)



```
struct Point {
    double x;
    double y;
}
```

8 byte x | 8 byte y

```
struct YelpReview {
    char* text_ptr;
    char username[16];
    uint8_t stars;
}
```

8 byte text_ptr | 16 byte username |

an address — character data — 1 byte stars

Q1: Fill in the struct definition for a Spotify song

```
struct Song {
    uint32_t time;
    char artist[512];
    char title[512];
}
```

4 byte time | 512-byte artist | 512-byte title

A struct describes a multi-value memory layout.

Q2: Draw the picture!

```
struct LineSegment {
    struct Point start;
    struct Point end;
}
```

16 byte start | 16 byte end

x for start | y for start | x for end | y for end

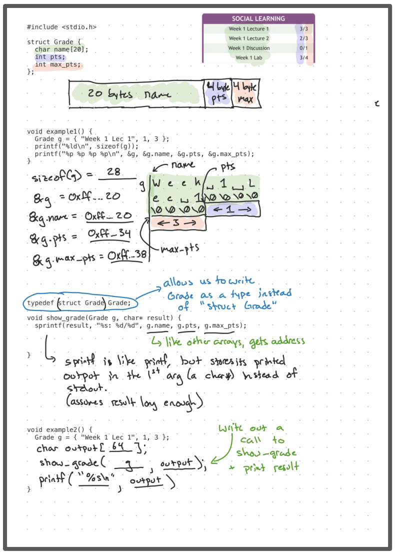
Q3: What is sizeof for each struct above?

Point 16

YelpReview 25* (32)

Song 512+512 + 4 = 1028

LineSegment 32



SOCIAL LEARNING	
Week 1 Lecture 1	3/3
Week 1 Lecture 2	2/3
Week 1 Discussion	0/1
Week 1 Lab	3/4

```
#include <stdio.h>

struct Grade {
  char name[20];
  int pts;
  int max_pts;
};
```

20 bytes name | 4 byte pts | 4 byte max

```
void example1() {
  Grade g = { "Week 1 Lec 1", 1, 3 };
  printf("%ld\n", sizeof(g));
  printf("%p %p %p %p\n", &g, &g.name, &g.pts, &g.max_pts);
}
```

sizeof(g) = 28

&g = 0xff...20

&g.name = 0xff...20

&g.pts = 0xff...34

&g.max_pts = 0xff...38

name
pts
g
W e e k ␣ 1 ␣ L
e c ␣ 1 \0 \0 \0 \0
\0 \0 \0 \0 ← 1 →
← 3 →
max-pts

```
typedef struct Grade Grade;
```

allows us to write Grade as a type instead of "struct Grade"

```
void show_grade(Grade g, char* result) {
  sprintf(result, "%s: %d/%d", g.name, g.pts, g.max_pts);
}
```

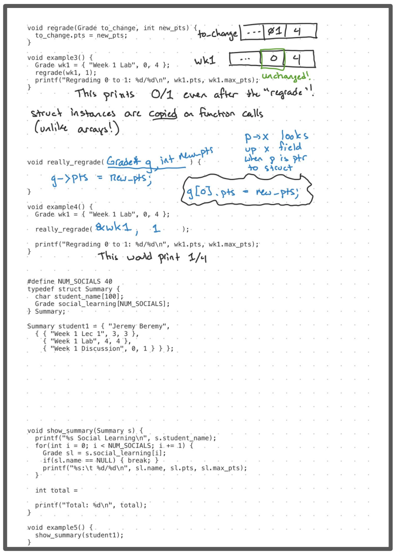
↳ like other arrays, gets address

sprintf is like printf, but stores its printed output in the 1st arg (a char*) instead of stdout.
(assumes result long enough)

```
void example2() {
  Grade g = { "Week 1 Lec 1", 1, 3 };
  char output[ 64 ];
  show_grade( g , output );
  printf( "%s\n", output )
}
```

Write out a call to show_grade + print result

Joe’s Notes (11am)



```
void regrade(Grade to_change, int new_pts) {
  to_change.pts = new_pts;
}

void example3() {
  Grade wk1 = { "Week 1 Lab", 0, 4 };
  regrade(wk1, 1);
  printf("Regrading 0 to 1: %d/%d\n", wk1.pts, wk1.max_pts);
}
```

to_change | ... | ~~0~~1 | 4

wk1 | ... | 0 | 4

unchanged!

This prints 0/1 even after the "regrade"!

struct instances are copied on function calls
(unlike arrays!)

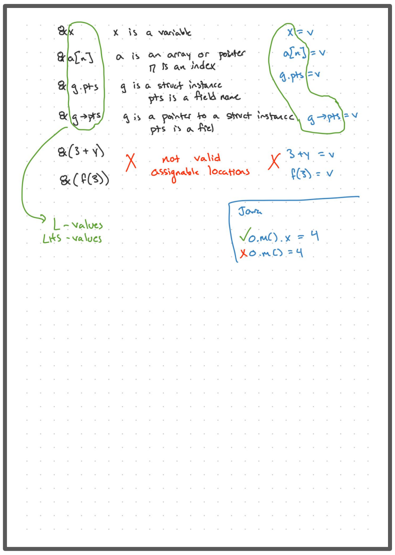
```
void really_regrade( Grade* g, int new_pts ) {
    g->pts = new_pts;
}

void example4() {
  Grade wk1 = { "Week 1 Lab", 0, 4 };

  really_regrade( &wk1, 1 );

  printf("Regrading 0 to 1: %d/%d\n", wk1.pts, wk1.max_pts);
}
```

p->x looks up x field when p is ptr to struct

g[0].pts = new_pts;

This would print 1/4

```
#define NUM_SOCIALS 40
typedef struct Summary {
  char student_name[100];
  Grade social_learning[NUM_SOCIALS];
} Summary;

Summary student1 = { "Jeremy Beremy",
  { { "Week 1 Lec 1", 3, 3 },
    { "Week 1 Lab", 4, 4 },
    { "Week 1 Discussion", 0, 1 } } };




void show_summary(Summary s) {
  printf("%s Social Learning\n", s.student_name);
  for(int i = 0; i < NUM_SOCIALS; i += 1) {
    Grade sl = s.social_learning[i];
    if(sl.name == NULL) { break; }
    printf("%s:\t %d/%d\n", sl.name, sl.pts, sl.max_pts);
  }

  int total = 

  printf("Total: %d\n", total);
}

void example5() {
  show_summary(student1);
}
```



&x x is a variable

&a[n] a is an array or pointer
n is an index

&g.pts g is a struct instance
pts is a field name

&g->pts g is a pointer to a struct instance
pts is a fiel

x = v

a[n] = v

g.pts = v

g->pts = v

&(3 + y)

&(f(3))

✗ not valid assignable locations

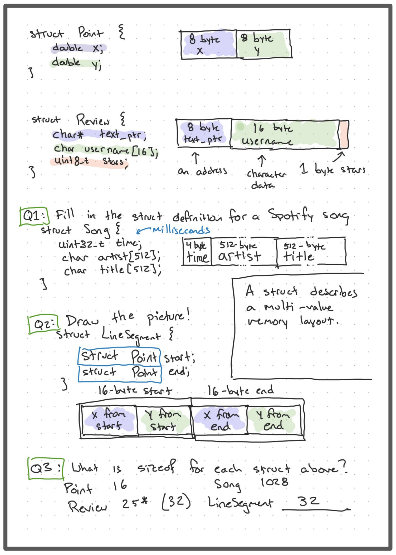
✗ 3+y = v
f(3) = v

L-values
LHS-values

Java

✓ o.m().x = 4
✗ o.m() = 4

Joe’s Notes (12:30pm)



```
struct Point {
    double x;
    double y;
}
```

8 byte x	8 byte y

```
struct Review {
    char* text_ptr;
    char username[16];
    uint8_t stars;
}
```

8 byte text_ptr	16 byte username	

an address

character data

1 byte stars

Q1: Fill in the struct definition for a Spotify song

```
struct Song {    ← milliseconds
    uint32_t time;
    char artist[512];
    char title[512];
}
```

4 byte time	512-byte artist	512-byte title

A struct describes a multi-value memory layout.

Q2: Draw the picture!

```
struct LineSegment {
    struct Point start;
    struct Point end;
}
```

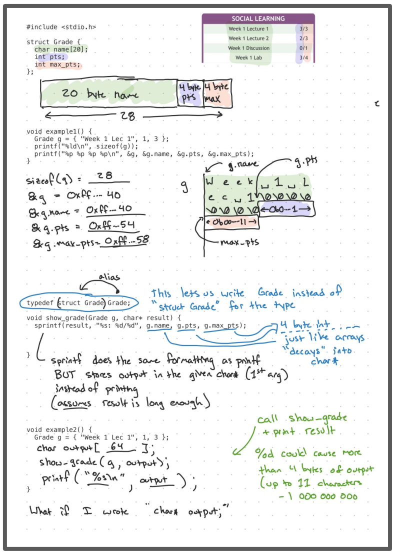
16-byte start | 16-byte end

x from start	y from start	x from end	y from end

Q3: What is sizeof for each struct above?

Point 16   Song 1028

Review 25* (32)   LineSegment 32



SOCIAL LEARNING	
Week 1 Lecture 1	3/3
Week 1 Lecture 2	2/3
Week 1 Discussion	0/1
Week 1 Lab	3/4

```
#include <stdio.h>

struct Grade {
  char name[20];
  int pts;
  int max_pts;
};
```

20 byte name	4 byte pts	4 byte max

← 28 →

```
void example1() {
  Grade g = { "Week 1 Lec 1", 1, 3 };
  printf("%ld\n", sizeof(g));
  printf("%p %p %p %p\n", &g, &g.name, &g.pts, &g.max_pts);
}
```

sizeof(g) = 28

&g = 0xff... 40

&g.name = 0xff...40

&g.pts = 0xff...54

&g.max_pts = 0xff...58

g

W	e	e	k	␣	1	␣	L
e	c	␣	1	\0	\0	\0	\0
\0	\0	\0	\0	←0b0...1→			
←0b00...11→							

g.name

g.pts

max_pts

alias

```
typedef struct Grade Grade;
```

This lets us write Grade instead of "struct Grade" for the type

```
void show_grade(Grade g, char* result) {
  sprintf(result, "%s: %d/%d", g.name, g.pts, g.max_pts);
}
```

4 byte int

just like arrays "decays" into char*

sprintf does the same formatting as printf BUT stores output in the given char* (1st arg) instead of printing (assumes result is long enough)

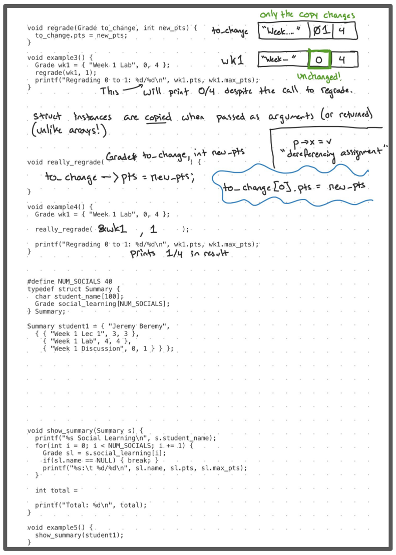
```
void example2() {
  Grade g = { "Week 1 Lec 1", 1, 3 };
  char output[ 64 ];
  show_grade(g, output);
  printf("%s\n", output);
}
```

call show_grade + print result

%d could cause more than 4 bytes of output (up to 11 characters -1 000 000 000

What if I wrote "char* output;"

Joe’s Notes (12:30pm)



```
void regrade(Grade to_change, int new_pts) {
  to_change.pts = new_pts;
}

void example3() {
  Grade wk1 = { "Week 1 Lab", 0, 4 };
  regrade(wk1, 1);
  printf("Regrading 0 to 1: %d/%d\n", wk1.pts, wk1.max_pts);
}
```

only the copy changes

to_change | "Week..." | ~~0~~ 1 | 4

wk1 | "Week–" | 0 | 4

unchanged!

This → will print 0/4 despite the call to regrade.

struct instances are copied when passed as arguments (or returned) (unlike arrays!)

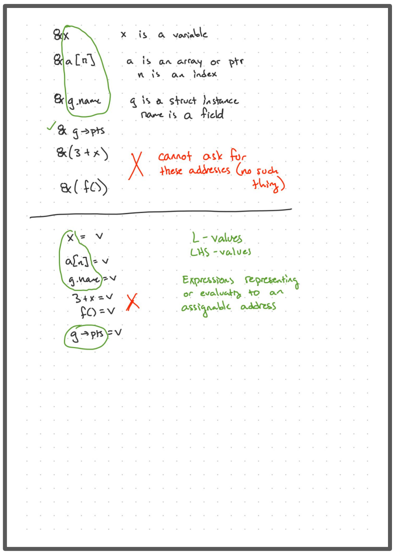
p→x = v
"dereferencing assignment"

```
void really_regrade( Grade* to_change, int new_pts ) {

  to_change -> pts = new_pts;

}

void example4() {
  Grade wk1 = { "Week 1 Lab", 0, 4 };

  really_regrade( &wk1 , 1 );

  printf("Regrading 0 to 1: %d/%d\n", wk1.pts, wk1.max_pts);
}
```

to_change[0].pts = new_pts

prints 1/4 in result

```
#define NUM_SOCIALS 40
typedef struct Summary {
  char student_name[100];
  Grade social_learning[NUM_SOCIALS];
} Summary;

Summary student1 = { "Jeremy Beremy",
  { { "Week 1 Lec 1", 3, 3 },
    { "Week 1 Lab", 4, 4 },
    { "Week 1 Discussion", 0, 1 } } };




void show_summary(Summary s) {
  printf("%s Social Learning\n", s.student_name);
  for(int i = 0; i < NUM_SOCIALS; i += 1) {
    Grade sl = s.social_learning[i];
    if(sl.name == NULL) { break; }
    printf("%s:\t %d/%d\n", sl.name, sl.pts, sl.max_pts);
  }

  int total =

  printf("Total: %d\n", total);
}

void example5() {
  show_summary(student1);
}
```



&x — x is a variable

&a[n] — a is an array or ptr
n is an index

&g.name — g is a struct instance
name is a field

✓ & g→pts

&(3 + x)

&(f())

✗ cannot ask for these addresses (no such thing)

x = v

a[n] = v

g.name = v

3 + x = v

f() = v ✗

g→pts = v

L - values
LHS - values

Expressions representing or evaluating to an assignable address