Structures

- A collection of related variables, bundled together into one variable

```
struct card {
    int rank;
    char suit;
};
```

fields a.k.a members of the structure

Structures

Variables in a `struct` are *fields*

```
struct cc_receipt {
    float amount;
    char cc_number[16];
};
```

Two fields: `float` named `amount`, and `char[16]` named `cc_number`

Structures

We're programming a checkers game.
We want a `struct` describing everything about a game piece

```
struct checkers_piece {
    // ???
};
```

Structures

```
struct checkers_piece {
    int x; // horizontal offset
    int y; // vertical offset
    int black; // 0 = white, non-0 = black
};
```

Structures

```c
// struct_eg1.c:
#include <stdio.h>

struct date {
    int year;
    int month;
    int day;
};

int main() {
    struct date today; // like 3 variables in 1!
    today.year = 2019; // use . to refer to fields
    today.month = 2;
    today.day = 25;
    printf("Today's date: %d/%d/%d\n",
           today.month, today.day, today.year);
    return 0;
}
```

use dot to access each field

```
$ gcc -c struct_eg1.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_eg1 struct_eg1.o
$ ./struct_eg1
Today's date: 2/25/2019
```

Structures

The `struct name { ... };` syntax defines a new struct data type

```
struct date {
    int year;
    int month;
    int day;
};
```

This syntax declares a *variable* that *has* that type

```
struct date today;
```

Structures

`struct` variable can be initialized in similar way to an array:

```
struct date {
    int year;
    int month;
    int day;
};

...

struct date today = {2019, 2, 25};
```

Structures

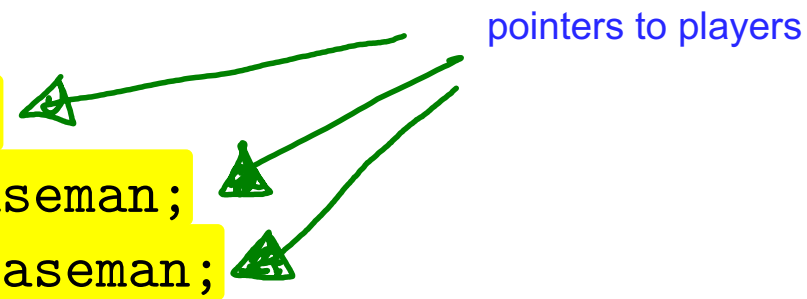
`struct` fields can be other `structs`

```
struct date {
    int year;
    int month;
    int day;
};

struct cc_transaction {
    // struct within a struct is fine!
    struct date purchase_date;
    float amount;
    char cc_number[16];
};
```

Structures

`struct` fields can be pointers

```
struct player {
    int home_runs;
    int strikeouts;
    int walks;
};

struct team {
    struct player *catcher;
    struct player *first_baseman;
    struct player *second_baseman;
    ...
};
```



Structures

`sizeof(struct player)` returns total size of all fields

```
struct date {
    int year;
    int month;
    int day;
};
```

What is `sizeof(struct date)`?

Structures

```c
// struct_sizeof.c:
#include <stdio.h>

struct date {
    int year;  // 4 bytes
    int month; // 4 bytes
    int day;   // 4 bytes
};

int main() {
    printf("%d\n", (int)sizeof(struct date));
}
```

sizeof returns long unsigned (i.e., %lu), so we cast it to int, or alternatively use %lu with printf instead of %d

```
$ gcc -c struct_sizeof.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_sizeof struct_sizeof.o
$ ./struct_sizeof
12
```

Structures

A `struct` can be a function parameter and/or return type

```
struct date next_day(struct date d) {
    if ((++d.day) > 30) { // assume 30-day months
        d.day = 1;
        if ((++d.month) > 12) {
            d.month = 1;
            d.year++;
        }
    }
    return d;
}
```

Structures

What if it were a `void` function, without `return d` at the end?

```
void next_day(struct date d) {
    if ((++d.day) > 30) { // assume 30-day months
        d.day = 1;
        if ((++d.month) > 12) {
            d.month = 1;
            d.year++;
        }
    }
}
```

Structures

`structs` are passed by value. So `next_day` on previous slide has no effect. We need to return a new `struct`, or take a pointer to a `struct` and dereference/modify it

Structures

this is pass by pointer, so we can modify d

```
void next_day_in_place(struct date *d) {
    if ((++(*d).day) > 30) {
        (*d).day = 1;
        if ((++(*d).month) > 12) {
            (*d).month = 1;
            (*d).year++;
        }
    }
}
```

since d is a pointer, we need to dereference it

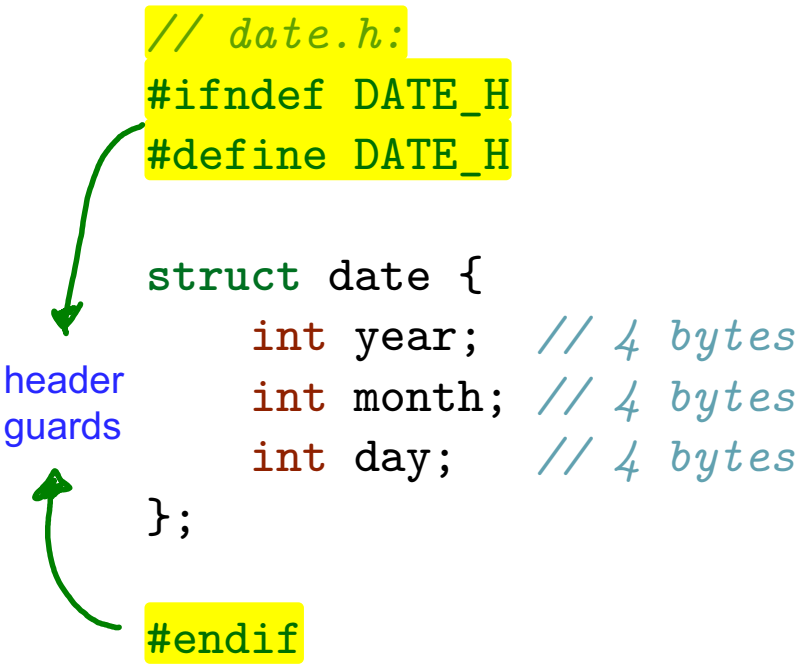
any changes made to the fields of d will persist even after we return from this function

Structures

`d->day` is a synonym for `(*d).day`

```
void next_day_in_place(struct date *d) {
    if ((++d->day) > 30) {
        d->day = 1;
        if ((++d->month) > 12) {
            d->month = 1;
            d->year++;
        }
    }
}
```

Structures




```
// date.h:
#ifndef DATE_H
#define DATE_H

struct date {
    int year;  // 4 bytes
    int month; // 4 bytes
    int day;   // 4 bytes
};

#endif
```

Structures

```c
// struct_next_day_1.c:
#include <stdio.h>
#include "date.h" // "struct date" defined here

struct date next_day(struct date d) {
    if (++d.day > 30) {
        d.day = 1;
        if (++d.month > 12) {
            d.month = 1;
            d.year++;
        }
    }
    return d;
}

int main() {
    struct date today = {2016, 2, 26};
    struct date tomorrow = next_day(today);
    printf("Tomorrow's date: %d/%d/%d\n",
           tomorrow.month, tomorrow.day, tomorrow.year);
}
```

```
$ gcc -c struct_next_day_1.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_next_day_1 struct_next_day_1.o
$ ./struct_next_day_1
Tomorrow's date: 2/27/2016
```

Structures

```c
// struct_next_day_2.c:
#include <stdio.h>
#include "date.h" // "struct date" defined here

void next_day_in_place(struct date *d) {
    if ((++d->day) > 30) {
        d->day = 1;
        if ((++d->month) > 12) {
            d->month = 1;
            d->year++;
        }
    }
}

int main() {
    struct date today = {2016, 12, 30};
    next_day_in_place(&today);
    printf("Tomorrow's date: %d/%d/%d\n",
           today.month, today.day, today.year);
    return 0;
}
```

```
$ gcc -c struct_next_day_2.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_next_day_2 struct_next_day_2.o
$ ./struct_next_day_2
Tomorrow's date: 1/1/2017
```

Structures

You can have an array of structs

```
struct album {
    const char *name;
    const char *artist;
    double length;
};

struct album music_collection[99999];
music_collection[0].name = "The Next Day";
music_collection[0].artist = "David Bowie";
music_collection[0].length = 41.9;
music_collection[1].name = "Hunky Dory";
...
```

each element of the array is a struct of type album. So, use index to access a particular elements and then use dot operator to access a specific field of that particular element

Structures

```c
// struct_array.c:
#include <stdio.h>

struct album {
    const char *name;
    const char *artist;
    double length;
};

int main() {
    struct album music_collection[99999];
    music_collection[0].name = "The Next Day";
    music_collection[0].artist = "David Bowie";
    music_collection[0].length = 41.9;
    music_collection[1].name = "Hunky Dory";
    return 0;
}
```

Structures

What is `sizeof(struct album)`?

```
struct album {
    const char *name;
    const char *artist;
    double length; // 8 bytes
};
```

Structures

```c
// struct_sizeof_album.c:
#include <stdio.h>

struct album {
    const char *name;
    const char *artist;
    double length; // 8 bytes
};

int main() {
    printf("sizeof(struct album) = %d\n", (int)sizeof(struct album));
    return 0;
}
```

Structures

```
$ gcc -c struct_sizeof_album.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_sizeof_album struct_sizeof_album.o
$ ./struct_sizeof_album
sizeof(struct album) = 24
```

size of a pointer is 8 bytes

24 bytes

- `const char *s` are just (8-byte) pointers
- Strings themselves not stored in the `struct`

Structures

You can have a struct with an array in it:

```
struct cc_receipt {
    float amount;
    char cc_number[16];
};
```

What is `sizeof(struct cc_receipt)`?

```
struct cc_receipt {
    float amount; // 4 bytes
    char cc_number[16];
};
```

Structures

```c
// sizeof_receipt.c:
#include <stdio.h>

struct cc_receipt {
    float amount; // 4 bytes
    char cc_number[16];
};

int main() {
    printf("sizeof(struct cc_receipt) = %d\n",
       (int)sizeof(struct cc_receipt));
    return 0;
}
```

Structures

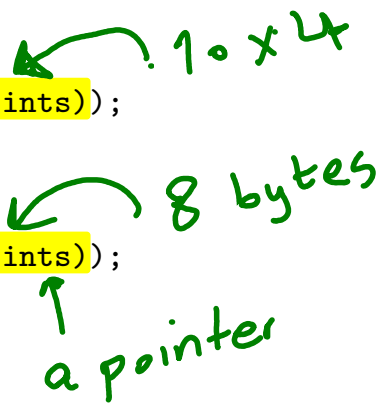
```
$ gcc -c sizeof_receipt.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o sizeof_receipt sizeof_receipt.o
$ ./sizeof_receipt
sizeof(struct cc_receipt) = 20
```

4 + 16

Answer: 20 bytes. `char cc_number[16]` is inside the struct, taking up 16 bytes.

Structures

```c
// struct_sizeof_receipt.c:
#include <stdio.h>

struct ten_ints {
    int ints[10];
};

void func1(struct ten_ints ints) {
    printf("func1 sizeof(ints)=%d\n", (int)sizeof(ints));
}

void func2(int *ints) {
    printf("func2 sizeof(ints)=%d\n", (int)sizeof(ints));
}

int main() {
    struct ten_ints ints;
    func1(ints);
    func2(ints.ints);
    return 0;
}
```




```
$ gcc -c struct_sizeof_receipt.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_sizeof_receipt struct_sizeof_receipt.o
$ ./struct_sizeof_receipt
func1 sizeof(ints)=40
func2 sizeof(ints)=8
```

Structures

- When a `struct` is passed to a function, everything inside is copied, including arrays
  This means an array wrapped in a `struct` is actually pass-by-value

arrays are not pass-by-value, but if the array is inside a struct (i.e., it is a struct field), then it becomes pass-by-value, because structs are pass-by-value

Clicker quiz!

What is the output of the following program?

```
#include <stdio.h>
struct Pokemon {
  char type;
  char name[12];
};
struct Pokemon makeElectric(struct Pokemon p) {
  p.type = 'E';
  return p;
}
int main(void) {
  struct Pokemon charmander = {
    'F', "Charmander"
  };
  makeElectric(charmander);
  printf("%s (%c)\n",
    charmander.name,
    charmander.type);
  return 0;
}
```

A. `Charmander (F)`
B. `Charmander (E)`
C. `Pikachu (E)`
D. Some other output
E. Code does not compile

Structures

Sometimes we get tired of writing `struct` over and over:

```
struct cc_receipt {
    float amount;
    char cc_number[16];
};

struct cc_receipt lunch_receipt;
struct cc_receipt dinner_receipt;
```

how can I get rid of writing
struct over and over again?

Structures

So we use this shorthand:

add this here

```
typedef struct {
    float amount;
    char cc_number[16];
} cc_receipt;

cc_receipt lunch_receipt;
cc_receipt dinner_receipt;
```

struct name

no need to write "struct" when declaring a new variable of that struct type

Now we can refer to the type simply as `cc_receipt` instead of `struct cc_receipt`

Structures

Say we have these definitions in `tennis.h`:

```
// tennis.h:
#ifndef TENNIS_H
#define TENNIS_H

typedef struct { // career statistics
    const char *name;
    int winners;
    int aces;
    int double_faults;
} player;

typedef struct {
    player *male;
    player *female;
} mixed_doubles_team;

#endif
```

Structures

Why might we do this:

```
typedef struct {
    player *male;
    player *female;
} mixed_doubles_team;
```

Instead of this?

```
typedef struct {
    player male;
    player female;
} mixed_doubles_team;
```

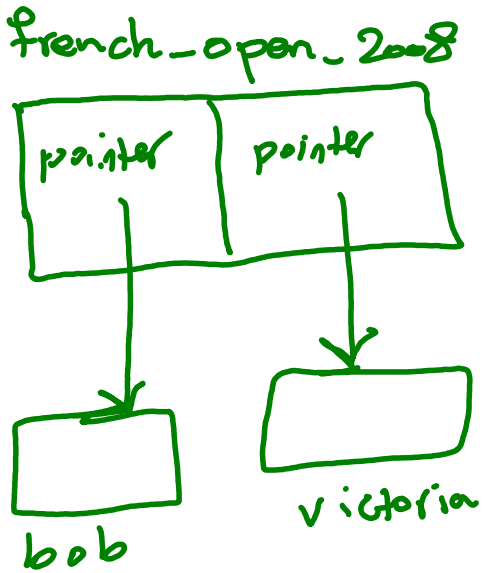
Take first case (using pointers)...

Structures

```
// tennis.h:
#ifndef TENNIS_H
#define TENNIS_H

typedef struct { // career statistics
    const char *name;
    int winners;
    int aces;
    int double_faults;
} player;

typedef struct {
    player *male;
    player *female;
} mixed_doubles_team;

#endif
```

Structures

```c
// tennis.c:
#include <stdio.h>
#include "tennis.h"

int main() {
    player bob_bryan;
    player victoria_azarenka;
    player samantha_stosur;

    mixed_doubles_team french_open_2008 =
        {&bob_bryan, &victoria_azarenka};

    mixed_doubles_team wimbledon_2008 =
        {&bob_bryan, &samantha_stosur};

    // updates "propagate" to team structs!
    bob_bryan.double_faults++;
    samantha_stosur.winners++;
    // ...
}
```



Structures

Size of `struct` is *at least* the sum of the sizes of its fields

It can be bigger if the compiler decides to add "padding"

```
struct plane {
    int passengers;
    double cargo_weight;
};
```

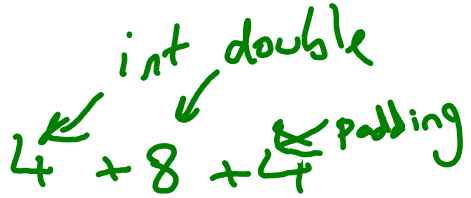
Structures

```c
// sizeof_plane.c:
#include <stdio.h>

struct plane {
    int passengers;       // int:    4 bytes
    double cargo_weight; // double: 8 bytes
};

int main() {
    printf("sizeof(struct plane) = %d\n", (int)sizeof(struct plane));
    return 0;
}
```

Structures

```
$ gcc -c sizeof_plane.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o sizeof_plane sizeof_plane.o
$ ./sizeof_plane
sizeof(struct plane) = 16
```

4 extra bytes used for "padding"



For obscure efficiency reasons, the compiler preferred to put 4 bytes of "spacer" between the `int` & `double`, driving total size to 16

Structures

Structures can be defined in a nested way:

```
typedef struct {
    struct {    // this struct type doesn't have a name;
        int r;  // it's just used once to declare a
        int b;  // field named color
        int g;
    } color;
    struct {    // again, no name
        int x;
        int y;
    } position;
} pixel;

pixel p;
p.color.r = 255;
p.position.x = 40;
p.position.y = 50;
```

Structures

```c
// nested_struct.c:
#include <stdio.h>

typedef struct {
    struct {    // this struct type doesn't have a name;
        int r;  // it's just used once to declare a
        int b;  // field named color
        int g;
    } color;
    struct {    // again, no name
        int x;
        int y;
    } position;
} pixel;

int main() {
    pixel p;
    p.color.r = p.color.g = p.color.b = 255;
    p.position.x = 40;
    p.position.y = 50;
    printf("[%d, %d, %d] at (%d, %d)\n",
           p.color.r, p.color.g, p.color.b,
           p.position.x, p.position.y);
    return 0;
}
```

Structures

```
$ gcc -c nested_struct.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o nested_struct nested_struct.o
$ ./nested_struct
[255, 255, 255] at (40, 50)
```