COMP1511 - Programming Fundamentals

Week 7 - Lecture 11

What did we cover last time?

Characters and Strings

- Using letters as variables
- Using arrays of letters
- Some useful library functions

Command Line Arguments

- Reading strings from the command line

What are we covering today?

Memory

- How functions work in memory
- Direct use of memory in C

Structs

- Making custom variables
- Collections of variables that aren't all the same type

Functions and Memory - a recap

What actually gets passed to a function?

- Everything gets passed **"by value"**
- Variables are copied by the function
- The function will then work with their own versions of the variables

What happens to variables passed to functions?

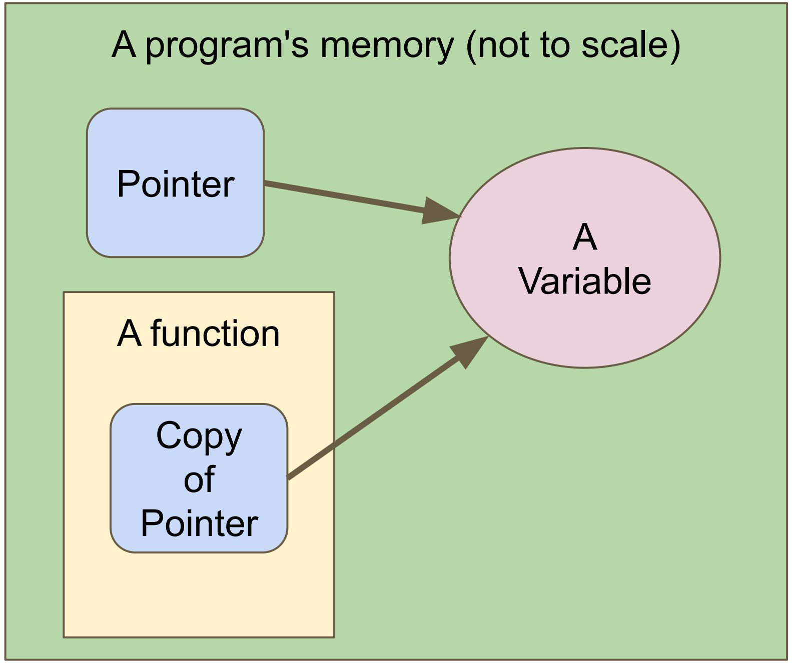
```
int main (void) {
    int x = 5;
    doubler(x);
    printf("x is %d.\n", x,);
    // "x is 5"
    // this is because the doubler function takes the value 5 from x
    // and copies it into the variable "number" which is a new variable
    // that only lasts as long as the doubler function runs
}

void doubler(int number) {
    number = number * 2;
}
```

Functions and Pointers

What happens to pointers that are passed to functions?

- Everything gets passed "by value"
- But the value of a pointer is a memory address!
- The memory address will be copied into the function
- This means **both** pointers are accessing the same variable!



Functions and Pointers

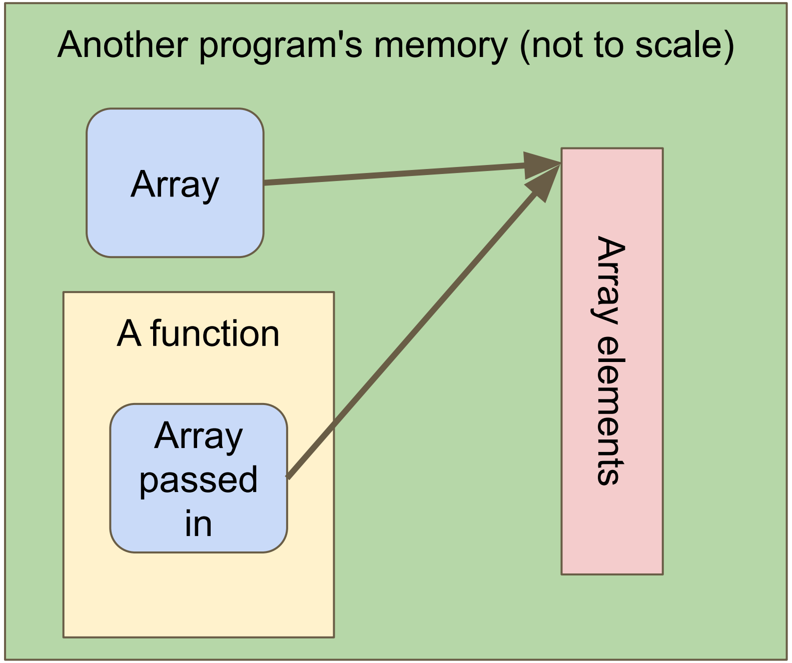
```
int main (void) {
    int x = 5;
    int *pointer_x = &x;
    double_pointer(pointer_x);
    printf("x is %d.\n", x);
    // "x is 10"
    // This is because double_pointer gets given access to x via its
    // copied pointer . . . since it changes what's at the other end of
    // that pointer, it affects x
}

// Double the value of the variable the pointer is aiming at
void double_pointer(int *num_pointer) {
    *num_Pointer = *num_pointer * 2;
}
```

Arrays are represented as memory addresses

Arrays and pointers are very similar

- An array is a variable
- It's not actually a variable containing all the elements
- When we use the array variable (no `[]`), it's actually the memory address of the start of the elements
- Arrays and pointers are nearly identical when passed to functions



Functions and Arrays

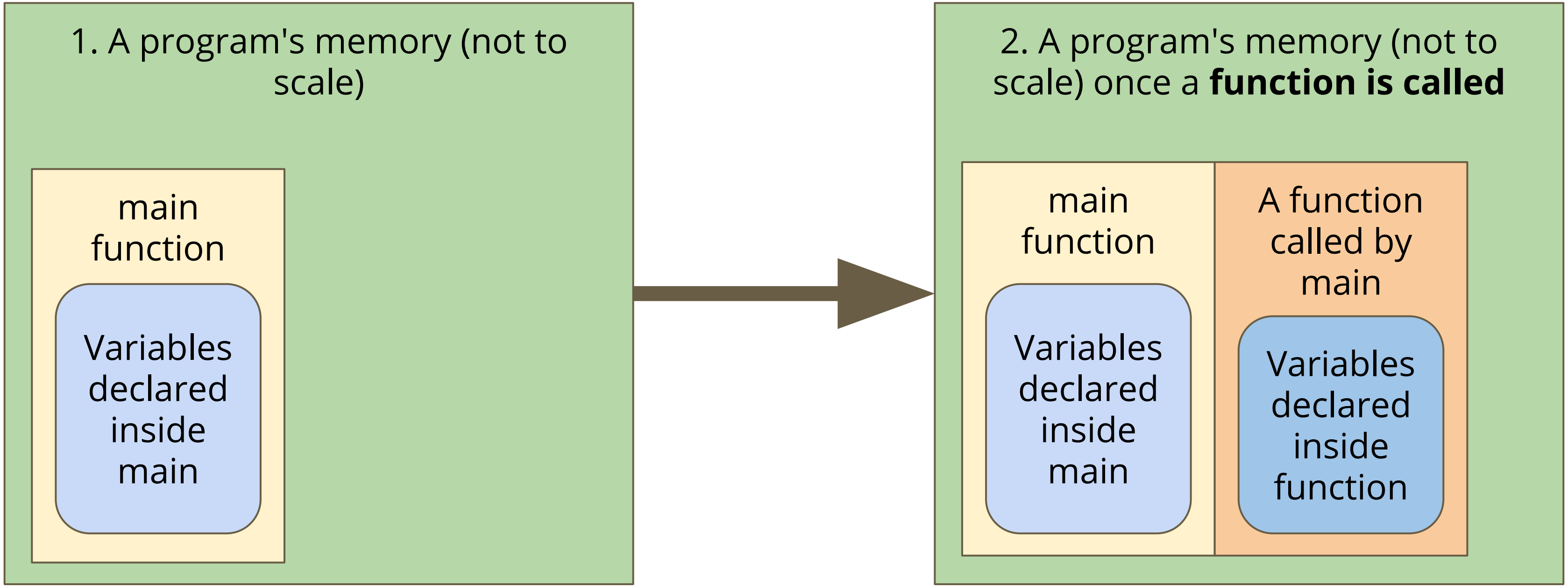
```
int main (void) {
    int numbers[3] = {1,2,3};
    double_all(3, numbers);
    printf("Array is: ");
    int i = 0;
    while(i < 3) {
        printf("%d ", numbers[i]);
        i++;
    }
    printf("\n");
    // "Array is 2 4 6"
    // Since passing an array to a function will pass the address
    // of the array, any changes made in the function will be made
    // to the original array
}
```

Functions and Arrays continued

```
// Double all the elements of a given array
void double_all(int length, int numbers[]) {
    int i = 0;
    while(i < length) {
        numbers[i] = numbers[i] * 2;
        i++;
    }
}
```

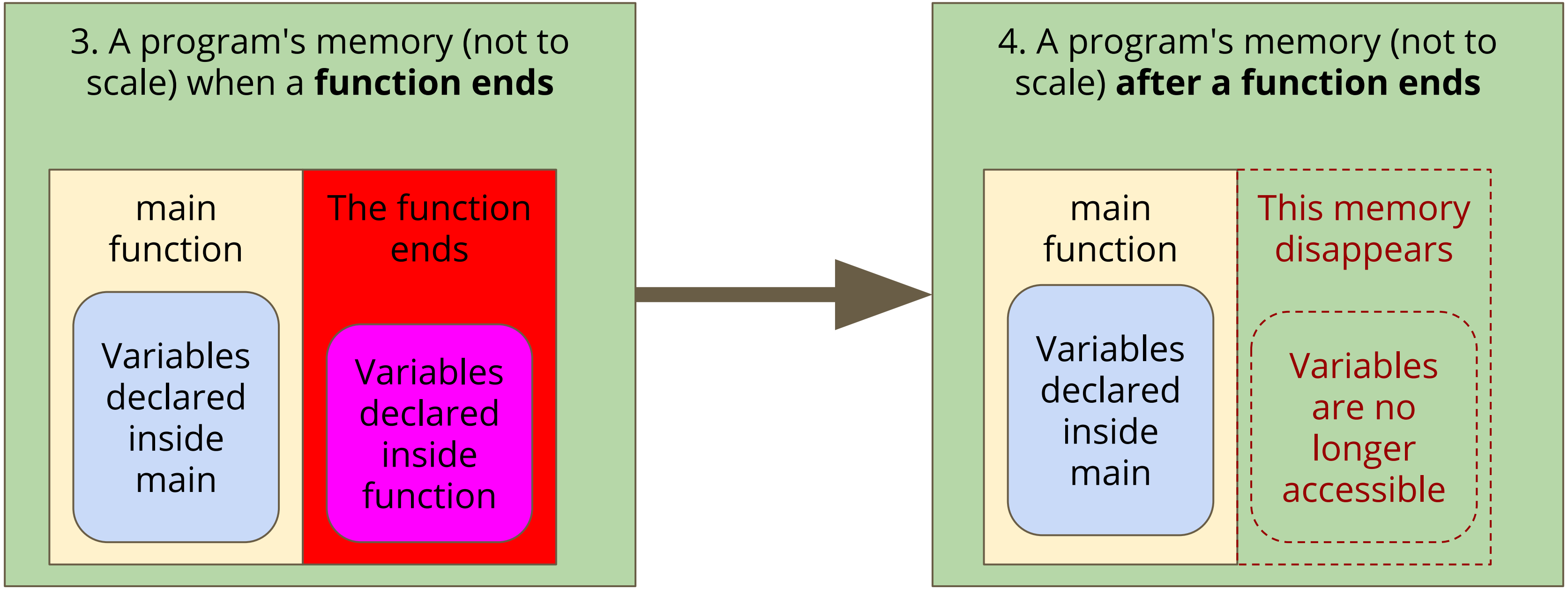
Memory in Functions

What happens to variables we create inside functions?



Memory in Functions

What happens to variables we create inside functions?



Keeping memory available

What if we want to create something in a function?

- We often want to run functions that create data
- We can't always pass it back as an output

```
// Make an array and return its address
int *create_array() {
    int numbers[10] = {};
    return numbers;
}
// This example will return a pointer to memory that we no longer have!
```

Memory Allocation

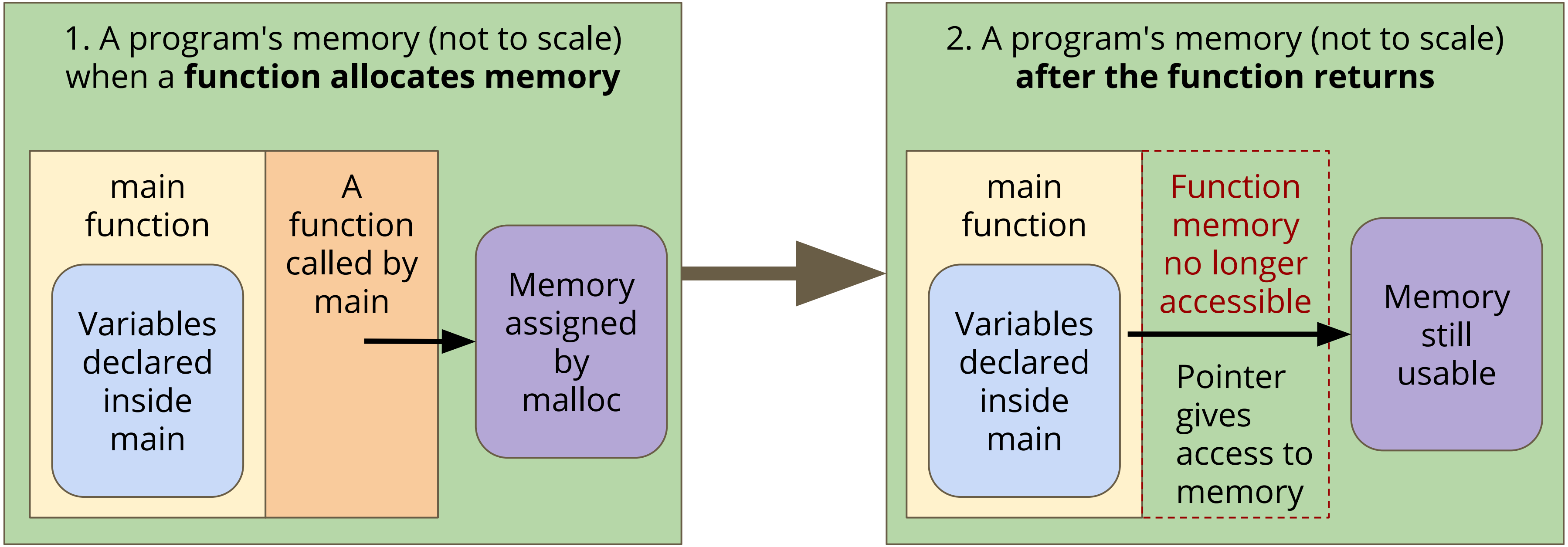
C has the ability to allocate memory

- A function called **`malloc(bytes)`** returns a pointer to memory
- Allows us to take control of a block of memory

- This won't automatically be cleaned up when a function ends
- To clean up the memory, we call **`free(pointer)`**
- **`free()`** will use the pointer to find our previous memory to clean it up

What malloc() does

Using malloc, we can assign some memory that is not tied to a function



Malloc() in code

We can assign a particular amount of memory for use

- The operator **`sizeof`** allows us to see how many bytes a variable needs
- We can use **`sizeof`** to allocate the correct amount of memory

```
// Allocate memory for a number and return a pointer to them
int *malloc_number() {
    int *int_pointer = malloc(sizeof (int));
    *int_pointer = 10;
    return int_pointer;
}
// This example will return a pointer to memory we can use
```

Cleaning up after ourselves

Allocated memory is never cleaned up automatically

- We need to remember to use **`free()`**
- Every pointer that is aimed at allocated memory must be freed!

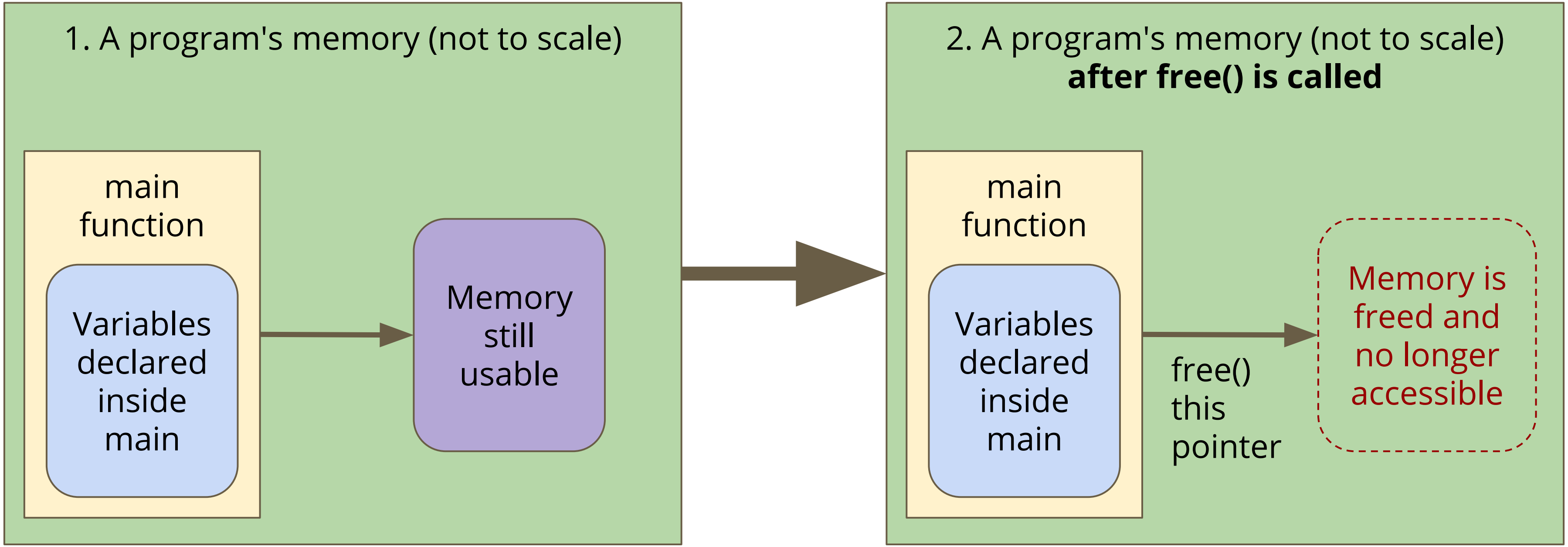
```
// Use an allocated variable via its pointer then free it
int main(void) {
    int *int_pointer = malloc_number();

    *int_pointer += 25;

    free(int_pointer);
    return 0;
}
```

Freeing up memory

Calling free will clean up the allocated memory that we're finished with



Using memory

Some things to think about with malloc() and free()

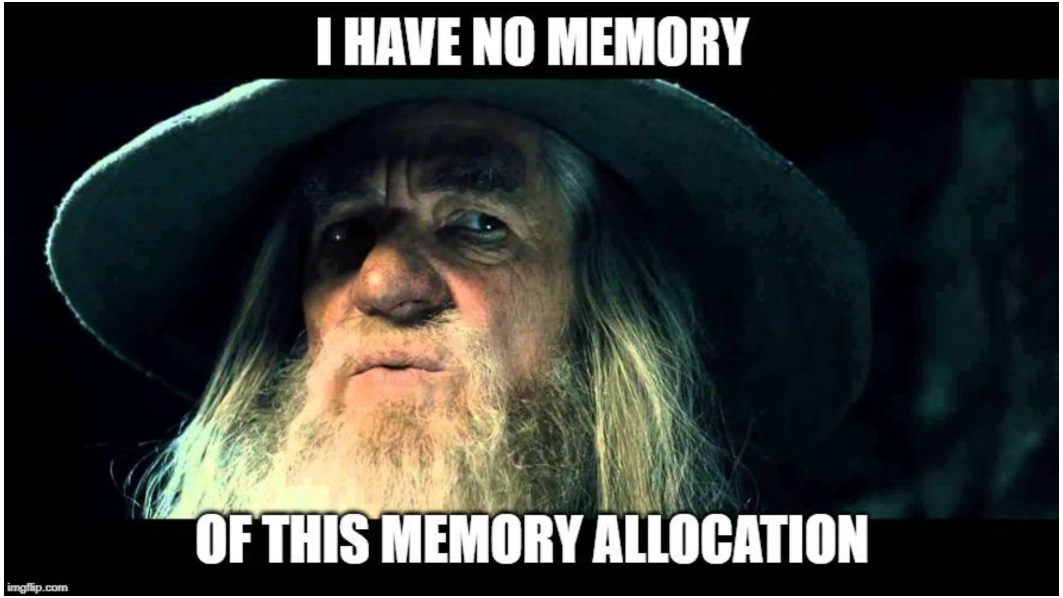
- You can use `sizeof()` to figure out how many bytes something needs
- We can `malloc()` any variables (but arrays are a bit complicated)
- In general, always use `sizeof()` with `malloc()`

- Anything allocated with `malloc()` must be `free()` after you've finished with it
- Otherwise we get what's known as memory leaks!
- `dcc --leak-check` can be used to tell you if you have any memory leaks

Break Time

Memory allocation is tricky

- It's easy to forget what you've allocated
- Then you might forget to free it!



Structs

A new way of collecting variables together

- Structs (short for structures) are a way to create custom variables
- Structs are variables that are made up of other variables
- They are not limited to a single type like arrays
- They are also able to name their variables
- Structs are like the bento box of variable collections

Before we can use a struct…

Structs are like creating our own variable type

- We need to declare this type before any of the functions that use it
- We declare what a struct is called and what the fields (variables) are

```
struct bender {
    char name[MAX_LENGTH];
    char element[MAX_LENGTH];
    int power;
};
```

Creating a struct variable and accessing its fields

Declaring and populating a struct variable

- Declaring a struct: "**struct** ***struct_name variable_name;***"
- Use the . to access any of the fields inside the struct by name

```
int main(void) {
    struct bender avatar;
    strcpy(avatar.name, "Aang");
    strcpy(avatar.element, "Air");
    avatar.power = 10;

    printf("%s's element is: %s.\n", avatar.name, avatar.element);
}
```

Accessing Structs through pointers

Pointers and structs go together so often that they have a shorthand!

-> is a new shorthand that avoids possible mistakes in dereferencing

```
struct bender *last_airbender = &avatar;

// knowledge of pointers suggests using this
(*last_airbender).power = 100;

// but there's another symbol that automatically
// dereferences the pointer and accesses a field
// inside the struct
last_airbender->power = 100;
```

Pointers and Structs

We often use pointers and structs together

- We use **->** to access fields when we have a pointer to a struct
- We often pass pointers to structs into functions

```
void display_person(struct bender *person) {
    printf("Name: %s\n", person->name);
    printf("Element: %s\n", person->element);
    printf("Power: %d\n", person->power);
}
```

Structs as Variables

Structs can be treated as variables

- Yes, this means arrays of structs are possible
- It also means structs can be some of the variables inside other structs
- In general, it means that once you've defined what a struct is, you use it like any other variable

Benders - an example of structs and malloc

We want to form a team of people with special elemental powers

- We'd like to have a struct that can represent an individual
- Then we'd like to build up a team
- We'll maintain an array of pointers
- And allocate memory for the team members

Create Structs for Characters

Create a struct to allow us to represent the characters

We'll borrow the one we created earlier

```
struct bender {
    char name[MAX_LENGTH];
    char element[MAX_LENGTH];
    int power;
};
```

Building up a team

We could actually do this with another struct!

We can make a struct that has an array of pointers to other structs

```
struct team {
    char name[MAX_LENGTH];
    int num_members;
    struct bender *team_members[TEAM_SIZE];
};
```

Creating a bender with a function

A function to allocate memory for a struct and give us a pointer to it

```
struct bender *create_bender(char *b_name, char *b_element, int b_power) {
    struct bender *new_bender = malloc(sizeof (struct bender));

    strcpy(new_bender->name, b_name);
    strcpy(new_bender->element, b_element);
    new_bender->power = b_power;

    return new_bender;
}
```

Setting up our structures in memory

We can use malloc in a very similar way to declaring a variable

```
// allocate the memory for one instance of benders
struct team *benders = malloc(sizeof (struct team));
strcpy(benders->name, "Avatar's team");

// Assigning the result of createBender to each element
// of benders's team_members array.
benders->team_members[0] = createBender("Aang", "Air", 10);
benders->num_members = 1;
benders->team_members[1] = createBender("Katara", "Water", 6);
benders->num_members++;
benders->team_members[2] = createBender("Sokka", "None", 2);
benders->num_members++;
```

Using structs without memory allocation

We can also use structs like regular variables

- Remember that accessing fields is different depending on whether you're using a pointer or not
- Accessing through a pointer: `->`
- Accessing a variable: `.`

```
// And an example of creating a struct without using
// memory allocation.
struct bender zuko;
strcpy(zuko.name, "Prince Zuko");
strcpy(zuko.element, "Fire");
zuko.power = 9;
```

Printing the contents

A function to print out the team. This only needs one pointer as input!

```
// print_team will print out the details of the team members
// to the terminal. It will not change the team.
void print_team(struct team *print_team) {
    printf("Team name is %s\n", print_team->name);
    int i = 0;
    while (i < print_team->num_members) {
        printf("Team member %s uses the element: %s\n",
                   print_team->team_members[i]->name,
                   print_team->team_members[i]->element
        );
        i++;
    }
}
```

What's left? There's memory left!

We still have allocated memory that we haven't given back!

- Every allocated piece of memory must be freed before the program ends
- This means we'd have to free all the members of the team
- And also the team itself
- **`dcc benders.c -o benders --leakcheck`**
- This command will create a version of the program that will check for memory leaks (unfreed memory allocations)

Some code for freeing memory

We can run a function that will clean up the memory for a team

```c
// free_team will free all the memory used for a team.
// It will first free all members, then the team itself
void free_team(struct team *f_team) {
    int i = 0;
    while (i < f_team->num_members) {
        free(f_team->team_members[i]);
        i++;
    }
    free(f_team);
}
```

What did we learn today?

Functions and Memory

- How functions have their own piece of memory
- How we lose access to anything in a function once it returns
- How we can specifically allocate memory

Structs

- Making our own custom variable types
- These can be collections of different types of variables