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 \textit{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?

```c
int main (void) {
    int x = 5;
    doubler(x);
    printf("x is %d.
", 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!

A program's memory (not to scale)

- Pointer
- A function
- Copy of Pointer
- A Variable
# Functions and Pointers

```c
int main (void) {  
    int x = 5;
    int *pointerX = &x;
    doublePointer(pointerX);
    printf("x is %d\n", x);
    // "x is 10"
    // This is because doublePointer 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 doublePointer(int *numPointer) {
    *numPointer = *numPointer * 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
void main() {
    int myNums[3] = {1,2,3};
    doubleAll(3, myNums);
    printf("Array is: ");
    int i = 0;
    while (i < 3) {
        printf("%d ", myNums[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
}
// Double all the elements of a given array
void doubleAll(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?

1. A program's memory (not to scale)
   - main function
     - Variables declared inside main

2. A program's memory (not to scale) once a function is called
   - main function
     - Variables declared inside main
   - A function called by main
     - Variables declared inside function
Memory in Functions

What happens to variables we create inside functions?

3. A program's memory (not to scale) when a **function ends**

- **main function**
  - Variables declared inside main
- **The function ends**
  - Variables declared inside function

4. A program's memory (not to scale) **after a function ends**

- **main function**
  - Variables declared inside main
- **This memory disappears**
  - Variables are no longer accessible
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

```c
// Make an array and return its address
int *createArray() {
    int numbers[10] = {0};
    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

1. A program's memory (not to scale) when a function allocates memory
   - main function
   - A function called by main
   - Variables declared inside main
   - Memory assigned by `malloc`

2. A program's memory (not to scale) after the function returns
   - main function
   - Function memory no longer accessible
   - Variables declared inside main
   - Pointer gives access to memory
   - Memory still usable
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

```c
// Allocate memory for a number and return a pointer to them
int *mallocNumber() {
    int *intPointer = malloc(sizeof (int));
    *intPointer = 10;
    return intPointer;
}
// 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!

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

    *iPointer += 25;

    free(iPointer);
    return 0;
}
```
Freeing up memory

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

1. A program's memory (not to scale)
   - main function
   - Variables declared inside main
   - Memory still usable

2. A program's memory (not to scale) after free() is called
   - main function
   - Variables declared inside main
   - Memory is freed and no longer accessible
   - free() this pointer
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 arrays as well as variables
- 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

```c
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 structname variablename;"
- Use the . to access any of the fields inside the struct by name

```c
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

```c
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

```c
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

We'll do some live coding with structs later!
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

```c
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

```c
struct team {
    char name[MAX_LENGTH];
    int numMembers;
    struct bender *teamMembers[TEAM_SIZE];
};
```
Creating a bender with a function

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

```c
struct bender *createBender(
    char *benderName,
    char *benderElement,
    int benderPower
) {
    struct bender *newBender = malloc(sizeof (struct bender));
    
    strcpy(newBender->name, benderName);
    strcpy(newBender->element, benderElement);
    newBender->power = benderPower;
    
    return newBender;
}
```
Setting up our structures in memory

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

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

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

```c
// 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;
```
// printTeam will print out the details of the team members
// to the terminal. It will not change the team.
void printTeam(struct team *printTeam) {
    printf("Team name is %s\n", printTeam->name);
    int i = 0;
    while (i < printTeam->numMembers) {
        printf("Team member %s uses the element: %s\n", 
                printTeam->teamMembers[i]->name, 
                printTeam->teamMembers[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 --leak-check`
- 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
// freeTeam will free all the memory used for a team.
// It will first free all members, then the team itself
void freeTeam(struct team *fTeam) {
    int i = 0;
    while (i < fTeam->numMembers) {
        free(fTeam->teamMembers[i]);
        i++;
    }
    free(fTeam);
}
```
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