Structs - a new way of collecting variables together

Structs - short for **structures** - a way to create a new custom type for variables. Like arrays structs are compound variables - they are made up of other variables. Structs are not limited to a single type like arrays. Structs have names for the variables they contain (arrays have a number). Think of structs as 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;
};
```

source code for simple_struct_example.c
Creating a struct variable and accessing its fields using .

```c
// create a struct variable
struct bender avatar;
// use . to access the field of a struct
avatar.power = 10;
strcpy(avatar.name, "Aang");
strcpy(avatar.element, "Air");
printf("%s's element is: %s.
", avatar.name, avatar.element);
```

source code for simple_struct_example.c
Accessing struct fields through a pointer using ->

```c
// create a struct variable
struct bender avatar;

// create a pointer to the struct variable
struct bender *andrew = &avatar;

// given a pointer to a struct
// use can use . and * to access
(*andrew).power = 100;

// but there is a better way
// -> dereferences the pointer and
// accesses a field inside the struct:
andrew->power = 100;
```

source code for simple_struct_pointer_example.c
Passing a struct pointer to a Function

```c
andrew->power = 100;
strcpy(andrew->name, "Andrew");
strcpy(andrew->element, "Code");
display_person(andrew);
return 0;
}

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

source code for simple_struct_pointer_example.c
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 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 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;
};
```

[source code for bender.c]
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 team_size;
    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

```c
// Allocate memory for a bender, populate it with
// the given information, then return a pointer
// to the allocated memory location

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;
}
```

source code for bender.c
Setting up our structures in memory

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

```c
// Allocate memory for a bender, populate it with
// the given information, then return a pointer
// to the allocated memory location

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;
}
```

Source code for `bender.c`
A function to print out the team. This only needs one pointer as input!

```c
// 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->team_size) {
        printf("Team member %s uses the element: %s\n",
                print_team->team_members[i]->name,
                print_team->team_members[i]->element);
        i++;
    }
}
```

(source code for bender.c)
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 all structs (from the team level)
void free_team(struct team *f_team) {
    // free all the allocations (bender structs) the team points at
    int i = 0;
    while (i < f_team->team_size) {
        free(f_team->team_members[i]);
        i++;
    }
    // then free the team struct itself
    free(f_team);
}
```

source code for bender.c
A function to print out the team. This only needs one pointer as input!

```c
int main(void) {
    struct team *companions = malloc(sizeof(struct team));
    strcpy(companions->name, "My Cabbages!");
    companions->team_size = 0;
    companions->team_members[companions->team_size] =
        create_bender("Aang", "Air", 10);
    companions->team_size++;
    companions->team_members[companions->team_size] =
        create_bender("Katara", "Water", 6);
    companions->team_size++;
    print_team(companions);
    free_team(companions);
}
```
- We have seen simple types e.g. **int, char, double**
  - variables of these types hold single values
- We have seen a compound type: **arrays**
  - array variables hold multiple values
  - arrays are homogenous - every array element is the same type
  - array element selected using integer index
  - array size can be determined at runtime
- Another compound type: **structs**
  - structs hold multiple values (fields)
  - struct are heterogeneous - fields can be different types
  - struct fields selected using name
  - struct fields fixed