COMP1511 - Programming Fundamentals

Term 3, 2019 - Lecture 12

What did we cover yesterday?

Arguments in our main function

- How to read command line arguments

Professionalism

- Important skills for working with people

Pointers

- Memory addresses stored in variables
- These give us access to the memory where a variable is stored

What are we covering today?

Structs

- C has another way of collecting variables
- This time, it's able to store variables of different types

Pointers and Structs

- Pointers to structs
- A code example using pointers and structs

Recap - Pointers and Memory

What is a pointer?

- It's a variable that stores the address of another variable of a specific type
- We call them pointers because knowing something's address allows you to "point" at it

Why pointers?

- They allow us to pass around the address of a variable instead of the variable itself

Using Pointers

Pointers are like street addresses . . .

- We can create a pointer by declaring it with a ***** *(like writing down a street address)*
- If we have a variable *(like a house)* and we want to know its address, we use **&**

```
int i = 100;
// create a pointer called ip that points at
// the location of i
int *ip = &i;
```

Using Pointers

If we want to look at the variable that a pointer "points at"

- We use the ***** on a pointer to access the variable it points at
- Using the address analogy, this is like navigating to the house at that address and looking inside the house

```
int i = 100;
// create a pointer called ip that points at
// the location of i
int *ip = &i;
printf("The value of the variable at %p is %d", ip, *ip);
```

Pointers in Functions

We'll often use pointers as input to functions

- Pointers give a function access to a variable that's in memory
- They also allow us to affect multiple variables instead of only having one output

```
void swap_nums(int *num1, int *num2) {
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```

Pointers and Arrays

These are very similar

- Arrays are actually memory addresses along with a certain amount of memory set aside for their use
- Pointers are also memory addresses
- This gives both pointers and arrays access to memory

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;
    int health;
};
```

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

```
int main(void) {
    struct bender aang;
    strcpy(aang.name, "Aang");
    strcpy(aang.element, "Air");
    aang.power = 10;
    aang.health = 5;

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

Accessing Structs through pointers

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

```
struct bender *avatar = &aang;

// knowledge of pointers suggests using this
*avatar.power = 10;

// but there's another symbol that automatically
// dereferences the pointer and accesses a field
// inside the struct
avatar->power = 10;
```

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

Break Time

Breaking into new territory

- The first half of the course may be familiar to anyone who's looked at programming before
- It also had concepts that, while important, are not very complex

- The second half of the course will leverage what you've learnt
- And will add both complexity and some concepts that take a little bit more abstract thinking

Let's write some code

Element Benders are having a fight in a forest!

- A team of four benders against one very powerful enemy
- We'll create a struct that represents a bender
- We'll have four of them in a team
- And one who will fight them all
- We'll create some functions that pit the benders against each other
- We'll loop a series of attacks until either side has lost

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;
    int health;
};
```

Create the actual struct variables

The struct is defined, now we create the actual variables

- The team can be in an array

```
int main (void) {
    struct bender companions[TEAM_SIZE];
    strcpy(companions[0].name, "Avatar Aang");
    strcpy(companions[0].element, "Air");
    companions[0].power = 10;
    companions[0].health = 5;
    strcpy(companions[1].name, "Katara");
    strcpy(companions[1].element, "Water");
    companions[1].power = 7;
    companions[1].health = 7;
    // etc
```

The struct is a variable type

Each instance of the struct can have a different name and stats

- Which means we can use the same struct for different characters!
- It also means that any of our characters are now interchangeable

```
struct bender zuko;
strcpy(zuko.name, "Prince Zuko");
strcpy(zuko.element, "Fire");
zuko.power = 20;
zuko.health = 20;
```

Let's use a function for a single attack

We pass pointers to structs in the function

This allows the function to make changes to our characters

```
void attack(struct bender *attacker, struct bender *target) {
    printf("%s attacks %s for %d damage.\n",
        attacker->name, target->name, attacker->power
    );
    target->health -= attacker->power;
    if (target->health <= 0) {
        // target has run out of health
        printf("%s is knocked out.\n", target->name);
    }
}
```

Passing addresses into functions

- We're passing addresses of structs to the attack function
- We do this by declaring that the function takes pointers as input (*****)
- And when we call the function, we provide the addresses (**&**) of the variables
- This allows the function to know where it can access our data (including the ability to change it)

Calling the attack function

If we just want a duel between one bender and Zuko

```
int teamCount = 0;
attack(&zuko, &companions[teamCount]);
attack(&companions[teamCount], &zuko);
```

But if we want to be able to use pointers to each of them

```
int teamCount = 0;
struct bender *companion = &companions[teamCount];
struct bender *prince = &zuko;
attack(prince, companion);
attack(companion, prince);
```

Let's fight until one side loses

Let's loop and keep attacking until either side is knocked out

- We'll need a function that tells us whether either side has run out of health
- Then we'll need a loop that keeps the fight going, letting the companions step in for each other if one is knocked out

stillAlive()

```
int stillAlive(struct bender *solo, struct bender team[TEAM_SIZE]) {
    int sAlive = 1;
    int tAlive = 0;
    if (solo->health <= 0) {
        sAlive = 0;
    }
    int i = 0;
    while (i < TEAM_SIZE) {
        if (team[i].health > 0) {
            tAlive = 1;
        }
        i++;
    }
    return sAlive * tAlive;
}
```

The main loop

```
int teamCount = 0;
struct bender *companion = &companions[teamCount];
declareElement(companion);
struct bender *prince = &zuko;
while (stillAlive(prince, companions)) {
    if (companion->health <= 0) {
        // this companion is knocked out, move on
        benderCount++;
        companion = &companions[teamCount];
        declareElement(companion);
    } else {
        attack(prince, companion);
        attack(companion, prince);
    }
}
```

The declareElement function

A void function doesn't give any information back to the rest of the program but it still might have some useful side effects

```
// A simple function to declare a bender's name and their element
void declareElement(struct bender *fighter) {
    printf(
        "%s wields the element: %s\n",
        fighter->name,
        fighter->element
    );
}
```

We might want a bit more variation

Introducing rand() - A random number generator from C's Standard Library

- Calling `rand()` will return an int from a generated sequence
- The sequence appears random
- But if we run the program again, it will generate the same sequence!

- `srand()` allows us to give a seed to our random number generator
- We can use "seed" values to select different sequences to use
- If we try to run different seeds every time, we'll get different sequences

Seed the rand() with command line input

- We can take input from the command line that ran the program and use that as our seed value
- This lets us change the sequence each time

```
int main (int argc, char *argv[]) {
    if (argc > 1) {
        // if we received a command line argument,
        // use that as our random seed
        srand(strtol(argv[1], NULL, 10));
    }
```

Let's add some randomness to the attack

Using rand and % we can get an int that's between 0 and a number

- Now the damage is inconsistent, we won't always know the result

```
void attack(struct bender *attacker, struct bender *target) {
    int damage = rand() % attacker->power;
    printf("%s attacks %s for %d damage.\n",
        attacker->name, target->name, damage
    );
    target->health -= damage;
    if (target->health <= 0) {
        // target has run out of health
        printf("%s is knocked out.\n", target->name);
    }
}
```

So we have a complete element bender battle!

We're looping through the fight and we don't always know the outcome!

- We've declared our first struct
- We also used it just like a variable in an array
- We passed pointers to our structs into functions

What's next?

- Can you write better style than this?
- There are a few places where separating things into functions would be very effective at increasing readability!

What did we learn today

Structs

- We've used structs as elements of an array
- We've used structs as members of another struct

- We're now seeing more complex code using strings, libraries, functions, pointers and structs