
COMP1511 - Programming Fundamentals

— Week 6 - Optional Livestream —

Let's write some code

a person

Element Benders are having a fight in a forest!

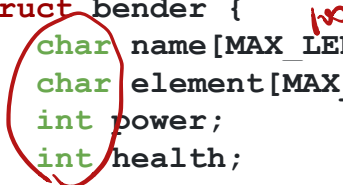
- A team of four benders against one very powerful enemy
- We'll create a struct that represents a bender
- We'll have four of them in a team
- And one who will fight them all
- We'll create some functions that pit the benders against each other
- We'll loop a series of attacks until either side has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

Handwritten notes:

- array* (with a red arrow pointing to the array declaration)
- char* (with a red arrow pointing to the first strcpy call)
- int* (with a red arrow pointing to the power and health assignments)
- in <string.h>* (with a red arrow pointing to the strcpy calls)
- 其他人 0~3* (with a red circle around the index 1 and a red arrow pointing to the array name)

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

Individual enemy

可互换的

```
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 ← attacker->name, target->name, attacker->power  
    );  
    target->health -= attacker->power; // health - 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; → 0-23  
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; → enemy  
    int tAlive = 0; → team  
    if (solo->health <= 0) {  
        sAlive = 0; → Return 0  
    }  
    int i = 0;  
    while (i < TEAM_SIZE) {  
        if (team[i].health > 0) { 如都 <= 0, tAlive = 0.  
            tAlive = 1; return 0.  
        }  
        i++;  
    }  
    return sAlive * tAlive;  
}
```

如 enemy-team 都有活着. return 1

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
        teamCount++; teamCount++;
        companion = &companions[teamCount];
        declareElement(companion);
    } else {
        attack(prince, companion);
        attack(companion, prince);
    }
}
```

(companions
[teamCount]
health)

array

loop until one side out of fight

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( 拥有
            "%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

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!