
COMP1511 - Programming Fundamentals

— Term 2, 2019 - Lecture 12 —

What did we cover yesterday?

Arguments in our main function

- How to read command line arguments

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 asking what’s inside the house at that address

```
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 ninja {  
    char name[MAX_LENGTH];  
    char phrase[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 ninja leo;
    strcpy(leo.name, "Leonardo");
    strcpy(leo.phrase, "Leonardo Leads");
    leo.power = 8;
    leo.health = 8;

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

Accessing Structs through pointers

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

```
struct ninja *turtle = &leo;

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

// but there's another symbol that automatically
// dereferences the pointer and accesses a field
// inside the struct
turtle->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

Help Sessions

Assignment help is available at help sessions!

- There's at least one Help Session every day of the week
- Check the class webpage for times and locations
- Help Sessions are also good for getting general help on course content

Let's write some code

Ninjas are having a fight on a New York Rooftop

- A team of four ninjas against one very powerful enemy
- We'll create a struct that represents a ninja
- We'll have four of them in a team
- And one who will fight them all
- We'll create some functions that pit the ninjas 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 ninja {  
    char name[MAX_LENGTH];  
    char phrase[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 ninja turtles[TEAM_SIZE];
    strcpy(turtles[0].name, "Leonardo");
    strcpy(turtles[0].phrase, "Leonardo leads");
    turtles[0].power = 8;
    turtles[0].health = 8;
    strcpy(turtles[1].name, "Donatello");
    strcpy(turtles[1].phrase, "Donatello does machines");
    turtles[1].power = 6;
    turtles[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 ninja shredder;  
strcpy(shredder.name, "The Shredder");  
strcpy(shredder.phrase, "Tonight I dine on turtle soup");  
shredder.power = 20;  
shredder.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 ninja *attacker, struct ninja *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 turtle and the shredder

```
int turtleCount = 0;
attack(&shredder, &turtles[turtleCount]);
attack(&turtles[turtleCount], &shredder);
```

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

```
int turtleCount = 0;
struct ninja *turtle = &turtles[turtleCount];
struct ninja *shred = &shredder;
attack(shred, turtle);
attack(turtle, shred);
```

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 turtles step in for each other if one is knocked out

stillAlive()

```
int stillAlive(struct ninja *solo, struct ninja 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 turtleCount = 0;
struct ninja *turtle = &turtles[turtleCount];
printf("%s\n", turtle->phrase);
struct ninja *shred = &shredder;
while (stillAlive(shred, turtles)) {
    if (turtle->health <= 0) {
        // this turtle is knocked out, move on
        turtleCount++;
        turtle = &turtles[turtleCount];
        printf("%s\n", turtle->phrase);
    } else {
        attack(shred, turtle);
        attack(turtle, shred);
    }
}
```

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 ninja *attacker, struct ninja *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 rooftop 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