COMP1511 - Programming Fundamentals

Term 1, 2019 - Lecture 14
Stream B

What did we cover on Tuesday?

Computer Memory

- Referencing and dereferencing

Arguments in our main function

- How to read command line arguments

Structs

- Packaging variables together

What are we covering today?

Professionalism

- What it means to be a programmer long term
- How to deal with insurmountable problems

Structs

- More complex use of 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);
```

What does it mean to be a programmer?

Marc’s four pillars of being a professional:

1. Communication
2. Teamwork
3. Resilience
4. Technical Skills

Communication

Making sure everyone understands what you’re doing

- Problem solving in teams involves shared understanding
- In order to solve human problems, we must understand what people need and how we can help them
- The more we communicate with computers the more risk we have of treating people like machines
- The ability to explain our code is important to keep us on track
- It’s especially important to be able to explain your code to non-programmers

Teamwork

Code is very rarely created alone

- Teamwork involves sharing and compromise
- Can you work with other people’s ideas?
- Can you follow someone else’s style and structure?
- Can you adapt your structure so that other people can use it?
- Can you provide support to your teammates?
- Teams made of people who get along are usually more successful than teams made of very skilled individuals!

Resilience

Work is hard.

- We need to look after ourselves
- If a job is so hard you can only survive it for a year, it’s not a good job
- We will sometimes be stuck in “impossible” situations
- Can you deliver your best work, even while knowing that it is not enough?
- Failure is inevitable, what counts is how you recover, not whether you fail

Technical Skills

How's your programming?

- Yes, this comes last in the list
- It's considered the easiest of the four to learn
- Still, we have the majority of COMP1511 to learn technical programming

More about Resilience and Surviving

You have an assignment due in 3 days

- Success isn’t about getting everything done
- It’s about prioritising your effort so you don’t have to do as much work!

Priorities:

- What gets you the most marks with the least amount of time?
- Code Style?
- Legal Play?

Don’t Panic!

Surviving is about acting rationally in panicky situations

- Take a moment to assess where you’re up to
- Figure out what your options are
- Break everything down into small bits
- Complete small pieces one at a time
- Aim for whatever gets you the highest marks

In Practical Terms

Get the most marks from your time

- Clear style, following the style guide
- Aim for legal play:
  - Blindly follow the “Valid Moves” diagram
  - Separate it into pieces that work individually
  - You can get more marks for 60% legal play than for “I tried for 100% and didn’t get anything running”
- Know how the marking and late penalties work

Break Time

Skills you'll want to learn . . .

1. Communication
2. Teamwork
3. Resilience
4. Technical Skills

COMP1511 News

Due Dates shift

- This week's lab has a delayed due date
- Wednesday next week, so you don't have to work on labs and the assignment at the same time

Assignment Tournament

- Please stop labelling your assignments as if they're me or something official!
- Players made by subject staff will be obvious because they don't have a rank number

COMP1511 News

Assignment 1 Benchmarks

Marc has made 3 players that you can use to compete against:

- Illegal Marc doesn't always play legally
- Legal Marc always plays legally, but doesn't think about what to play
- Tactical Marc makes simple, but intelligent decisions

COMP1511 News

Help Sessions

Due to popular demand, there are now more help sessions!

- Monday 12-3pm (Tabla)
- Tuesday, Wednesday, Thursday 6-8pm (Bugle + Horn)
- Thursday 9-11am (Kora/Sitar)
- Friday 12-5pm (Viola + Cello)

Recap - Structs

A struct is a collection of variables held together under one name

They're more specific than arrays, in that each element gets a variable name

```
struct student {
    int studentID;
    char name[64];
    char tut_lab[16];
    int assign1_mark;
};
```

Accessing Structs

We use a . to access member variables inside the struct

```c
int main(void) {
    struct student student1;
    student1.name = "James";
    student1.studentID = 1234567;

    printf("Student: %s ID: %d", student1.name, student1.studentID);
}
```

Accessing Structs through pointers

Remember that we can use -> to dereference a pointer and access a member variable inside a struct

```
int main(void) {
    struct student student1;
    student1.name = "James";
    student1.studentID = 1234567;

    struct student *sPointer = &student1;
    printf("Student: %s ID: %d", sPointer->name, sPointer->studentID);
}
```

Structs as Variables

Structs can be treated as variables

- Yes, this means arrays of structs are possible
- It also means structs within structs are possible
- It also means structs can contain other structs

Let's write some code

A village is besieged by Orcs. Only one brave knight stands in their way!

- A small game-like simulation
- We'll use structs to hold information about combatants
- Some of them are in a group, so we'll use a struct to store them
- We'll set up a loop so that they automatically fight each other

Create Structs for Characters

Create some structs to allow us to represent the characters

```
struct fighter {
    char name[20];
    int strength;
    int health;
};

struct warband {
    char name[20];
    struct fighter grunts[NUM_GRUNTS];
};
```

Set up our two teams (inside the main)

```
// Set up defender
struct fighter defender;
strcpy(defender.name, "Anduin Lothar");
defender.strength = 5;
defender.health = 5;

// Set up attackers
struct warband orcs;
strcpy(orcs.name, "Blackrock Orcs");
int i = 0;
while(i < NUM_GRUNTS) {
    strcpy(orcs.grunts[i].name, "Grunt");
    orcs.grunts[i].strength = 3;
    orcs.grunts[i].health = (rand() % 6) + 1;
    i++;
}
```

Interesting Structs

We're using structs now in interesting ways

- We're using a struct that contains an array of other structs!
- We're also using the same struct for two different entities
- Doing this allows us to access them in a consistent way
- Later on, we'll write a function that takes fighter structs as input and it won't matter whether it's our defender or attacker

rand()

A random number generator from C's Standard Library

- Calling rand() will return a number from a generated sequence
- While that sequence will appear random . . .
- The sequence will always be the same!

- 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
- We're also using a Standard Library function to convert the input string into an integer

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

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

```
int attack(struct fighter *attacker, struct fighter *target) {
    printf("%s attacks %s for %d damage.\n", attacker->name,
                           target->name, attacker->strength);
    target->health -= attacker->strength;
    if (target->health <= 0) {
        // target has died
        printf("%s has died.\n", target->name);
        return 0;
    } else {
        return 1;
    }
}
```

Passing references to functions

- We're passing references 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)

Now we loop through the fight

```
// Fight through the grunts, one at a time
int front = 0;
while (defender.health > 0 && front < NUM_GRUNTS) {
    if(attack(&defender, &orcs.grunts[front])) {
        // Orc didn't die, allow it to counterattack
        attack(&orcs.grunts[front], &defender);
    } else {
        // Orc at the front has died, move on to the next
        front++;
    }
}

// Who won?
if(defender.health > 0) {
    printf("%s is Victorious!\n", defender.name);
} else {
    printf("%s have overrun the village!\n", orcs.name);
}
```

Streamlining our code

- We're running the attack function inside an if condition
- This means the attack function will run as normal, but we'll be using its output as our if condition

What did we learn today

Some ideas of what it's like to work as a professional

- Communication, Teamwork, Resilience, Technical Skills

Structs as variables

- 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 libraries, functions, pointers and structs