COMP1511 - Programming Fundamentals

Week 2 - Lecture 3

What did we learn last week?

COMP1511 as a subject

C as a programming language

- Basic Syntax
- `printf` and `scanf`
- Variables (ints and doubles)
- Maths operators (`+`,`-`,`*`,`/` and `%`)
- Relational Operators (`<`, `>`, `==`, etc)
- Logical Operators (`&&`, `||`, `!`)
- `if` and `else` statements

What are we covering today?

Going slightly deeper in programming . . .

- Recap of some of the concepts introduced last week
- Continuing learning about `if` statements
- Some in-depth thinking about problems
- Processes for solving problems
- Continuing the Dice Checker, but with more nuance

Recap - Variables

- Data storage in memory
- Made up of bits (and bytes are sets of 8 bits)
- Chosen for a specific purpose
    - **int** - 32 bit integer numbers
    - **double** - 64 bit floating point numbers
- We choose the name - try to make it meaningful!
- We can change the value as we go

Recap - Reading and Writing to our Terminal

printf()

- Outputs text to the terminal
- We can format our variables to output them
  - `%d` - decimal integer (works with ints)
  - `%lf` - long floating point number (works with doubles)

scanf()

- Reads text from the user
- Uses a similar format to printf()

Recap - Maths Operators

- `+`, `-`, `*`, `/`
- These four work pretty much exactly as normal maths does
- `(` brackets `)` allow us to force some operations to run before others

`%` - Modulus

- Gives us the remainder (as an integer) of a division between integers
- Does not actually perform the division

Recap - Relational and Logical Operators

Relational Operators

- `>, >=, <, <=, ==, !=`
- Comparisons made between numbers
- Will result in 1 for true and 0 for false

Logical Operators

- `&&, ||, !`
- Comparisons made between true and false (0 and 1) results
- Used to combine Relational Operator Questions together

Recap - if and else statements

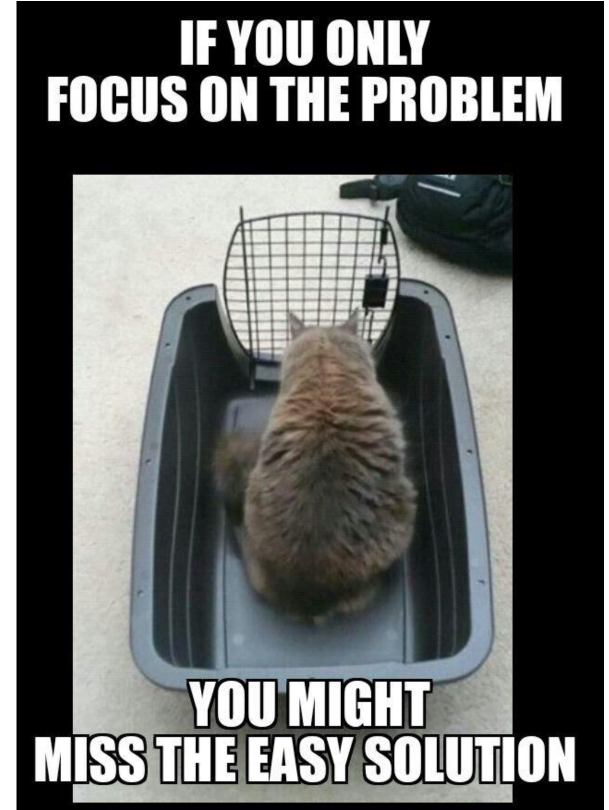
- Branching control of a program

```
// One of the challenges from last week
if (total > TARGET_VALUE) {
    // this runs if the test above is true
    printf("Skill roll succeeded!\n");
} else if (total == TARGET_VALUE) {
    // otherwise if this test is true
    printf("Skill roll tied!\n");
} else {
    // if neither of the others are true
    printf("Skill roll failed!\n");
}
```

Problems and Solutions

What's our problem?

- This is always a good question!
- Spending some time figuring out exactly what we aim to do (or what's stopping us from getting there) is important
- Keeping the problem in mind keeps you focused on a solution



A process for problem solving

We can develop a way to approach all problems

1. Figure out what's wrong (or what we need to solve)
2. Find out what our options are (what code could we write or change?)
3. Assess those options
   a. How well do they solve the problem?
   b. Can we make them work?
4. Pick an option to try
5. Did it work?
   a. If it didn't or even if it did, we can get more information for our next attempt

Back to the Dice Checker

We created a program that:

- Asked the user to input their dice values
- Reported back whether the total was above or below a target value

Let's look at one problem that might occur

- What if the user enters incorrect values?
- Too high or too low?

Testing our Input

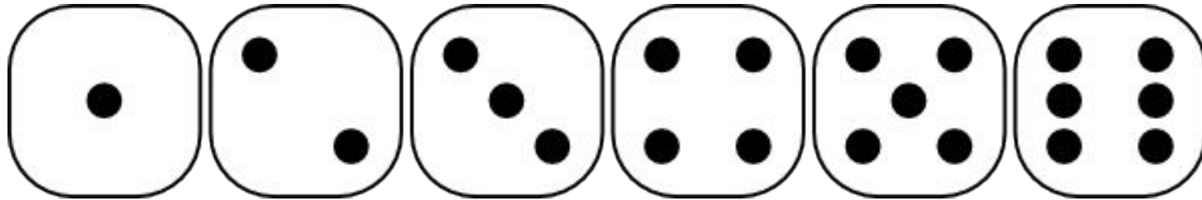
Let's assume we have this input code:

```
// Setup dice variables
int dieOne;
int dieTwo;

// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);
// repeat for the second die
printf("Please enter your second die roll: ");
scanf("%d", &dieTwo);
```

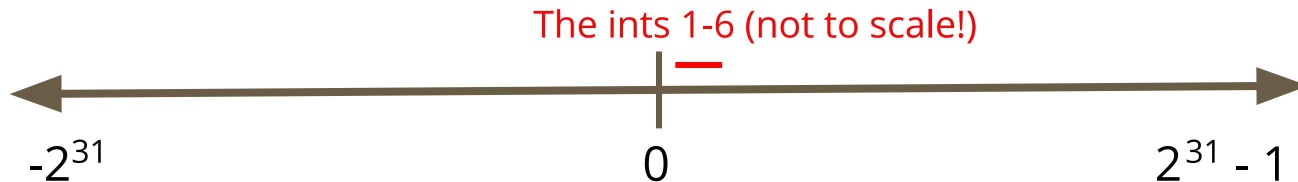
Testing Input Range

A six sided die has a specific range of inputs



We will only accept inputs in this range

But ints have a much wider range!



Testing Input Range in Code

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);

// test for proper input range
if (1 <= dieOne && dieOne <= 6) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die's range
}
```

Using Constants in code

The 1 and 6 are the minimum and maximum values of the dice

- We could use something like this at the start of our program:

```
#define MIN_VALUE 1
#define MAX_VALUE 6
```

Using `MIN_VALUE` and `MAX_VALUE`:

- Makes the program much easier to modify for different dice sizes
- Also makes things much more readable by using English words instead of numbers

Break Time

Dice

- The Egyptians were using flat sticks to randomise movement in Senet
- That dates games with randomisation back past 3000BC
- Six sided dice have been excavated in Iran from 2800-2500BC
- Nowadays we usually use dice ranging from 4 to 20 sides
- We're going to look at random number generation later in the course so you'll be able to simulate your own dice

What are our options?

If we know we have incorrect input, what do we do?

We have several options . . .

- PANIC!!!!!
- Reject the input, end the program
- Let the user know what the correct input is
- Correct the input
- Ask for new input

Reject the input

We can just end the program if the input is incorrect

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);

// test for proper input range
if (MIN_VALUE <= dieOne && dieOne <= MAX_VALUE) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die's range
    return 1;
}
```

Assessing This Option

Is it a good idea to have the program just end?

- What’s a good way for the program to reject incorrect input?
- If we’re testing or using the program, what do we want to see?

Reporting Failure

Information from the program helps the user

```
// test for proper input range
if (MIN_VALUE <= dieOne && dieOne <= MAX_VALUE) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die’s range
    printf(“Input for first die, %d was out of
range. Program will exit now.\n”, dieOne);
    return 1;
}
```

Can we do better?

Exploring other options

Let's give the user information that helps them correct the input issues

```
// test for proper input range
if (MIN_VALUE <= dieOne && dieOne <= MAX_VALUE) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die's range
    printf("Input for first die, %d was out of the
range 1-6. Program will exit now.\n", dieOne);
    return 1;
}
```

Correcting the input without exiting

If we want the program to finish executing even with bad input

Imperfect, but sometimes we want the program to finish

What are our options?

- Clamping - anything outside the range gets “pushed” back into the range
- Modulus - a possibly elegant solution

Clamping Values

Correcting the values - a brute force approach

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);

// clamp any values outside the range
if (dieOne < MIN_VALUE) {
    dieOne = MIN_VALUE;
} else if (dieOne > MAX_VALUE) {
    dieOne = MAX_VALUE;
}
```

Any Issues with Clamping?

- Definitely end up with input that works
- But is it correct?
- What are the issues with correcting data without the user knowing?

Modulus

A reminder of what it is

- **%** - A maths operator that gives us the remainder of a division

How can we use it?

- Any number “mod” 6 will give us a value from 0 to 5
- If we change any 0 to a 6, we get the range 1 to 6
- This means the user could type in completely random numbers and be given a 1-6 dice roll result

Using Modulus in code

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);

// mod forces the result to stay within 0-5
dieOne = dieOne % MAX_VALUE;
// make any 0 into MAX_VALUE
if (dieOne == 0) {
    dieOne = MAX_VALUE;
}
```

Pros and Cons of using Modulus for dice

Pros

- We guarantee a number between 1 and 6 (or whatever the max value is)
- We don’t shut down unexpectedly due to incorrect input
- We give a very dice-like randomish result (as opposed to clamping)

Cons

- We might accept incorrect input silently
- We might make a change that affects the user's expectations

A Range of Solutions

Which one to use?

- No single answer
- The original purpose of the program can help us decide
- What's our priority?
- Exact correctness?
- Failure on any kind of incorrect data?
- Usability and randomisation over correctness?

The Upgraded Dice Checker

- The programmer can set the size of the dice in `#define` constants
- The user can enter any number and it will produce a valid roll
- The program will still report back success or failure

Starting from our previous Dice Checker program, we can make some modifications to give it some new capabilities

Setting up

We'll start with our description of the program

```
// The Dice Checker v2
// Marc Chee, February 2019

// Allows the user to set dice size
// Tests the rolls of two dice against a target number
// Able to deal with user reported rolls outside the range
// Will report back Success, Tie or Failure

#include <stdio.h>

#define MIN_VALUE 1
#define MAX_VALUE 6
```

Variables and Constants

Set up the Target constant and some variables

```
// The secret target number
#define SECRET_TARGET 7

int main (void) {
     int dieOne;
     int dieTwo;
```

Taking user input

Two rolls will be taken as input (only one is shown here)

```
// Process the first die roll
printf("Please enter your first die roll: ");
scanf("%d", &dieOne);

// Check and fix the die roll
if (dieOne < MIN_VALUE || dieOne > MAX_VALUE) { // dieOne is invalid
    printf("%d is not a valid roll for a D%d.\n", dieOne, MAX_VALUE);
    dieOne = (dieOne % MAX_VALUE);
    if (dieOne == 0) {
        dieOne = MAX_VALUE;
    }
}
```

Calculate and report the total

This is identical to last week's code

```
// calculate the total and report it
int total = dieOne + dieTwo;
printf("Your total roll is: %d\n", total);

// Now test against the secret number
if (total > SECRET_TARGET) {
    // success
    printf("Skill roll succeeded!\n");
} else {
    // failure
    printf("Skill roll failed!\n");
}
```

We have a new Dice Check Program

We’ve added:

- Some measures against user mistakes
- Some modifiability

We made some decisions:

- We will report any user errors
- But we're also delivering a die roll regardless

What we learnt today

- A recap of the technical programming we’ve done so far
- Some discussion of how to approach problem solving
- A walkthrough of a technical problem and its many possible solutions
- Some thinking about why we choose a particular solution

- Some use of logical operators
- Some use of modulus
- Some slightly more complex code (if statements inside if statements)