COMP1511 - Programming Fundamentals

Term 2, 2019 - 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 (+,-,*,/)
- Relational Operators (<, >, ==, etc)
- Logical Operators (&&, ||, !)
- If statements

What are we covering today?

If statements continued . . .

- Recap of some of the concepts introduced last week
- A look at some more if statements
- Some extra problems and solutions
- 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 the same format as 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

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

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 try adding a bit to that:

- What if the user enters the wrong values?
- What if we want to use different sized dice?

Problems and Solutions

Deciding how to approach a problem

- What if the user enters the wrong values?

We have a few options:

- Reject the incorrect values?
- Correct the values?
- Punish the user for trying to cheat?

Solutions become code ideas

First we test our input to see if it’s correct

- A specification for correct input
  - Are we assuming six sided dice?
- How do we test if the input given to us was correct?

Testing our Input

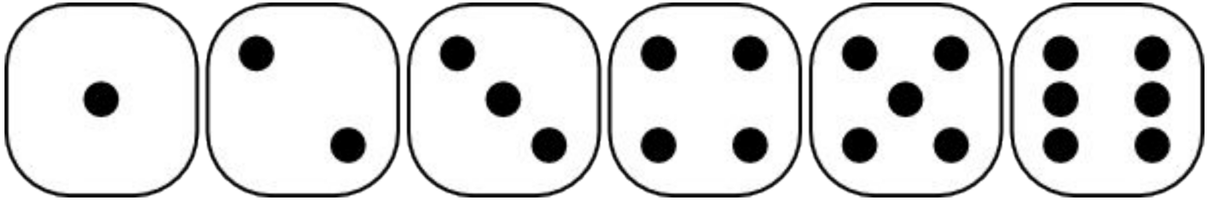
Let’s assume we have this input code:

```
// 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);
```

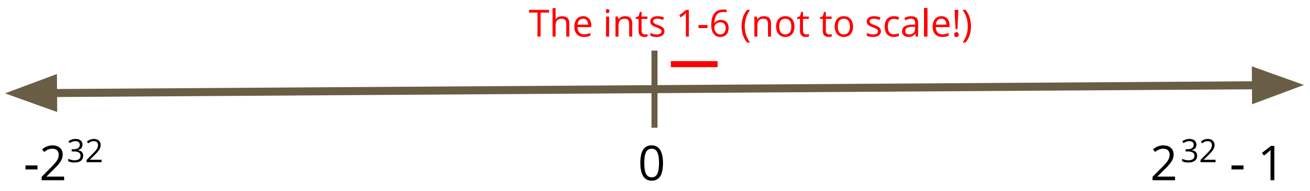
Testing Input Range

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
}
```

Dealing with Incorrect Input

What will we do with the incorrect input?

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

```c
// 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
    return 1;
}
```

Pros and Cons

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 us

```
// 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
    printf("Input for first die, %d was out of
range. Program will exit now.\n", dieOne);
    return 1;
}
```

Can we do better?

Trick question . . . obviously we can

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

```
// 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
    printf("Input for first die, %d was out of the
range 1-6. Program will exit now.\n", dieOne);
    return 1;
}
```

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 use dice ranging from 4 to 20 sides (in general)
- We're going to look at random number generation later in the course so you'll be able to simulate your own dice

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 < 1) {
    dieOne = 1;
} else if (dieOne > 6) {
    dieOne = 6;
}
```

Pros and Cons

- 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

```c
// 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 % 6;
// make any 0 into a 6
if (dieOne == 0) {
    dieOne = 6;
}
```

Pros and Cons

Pros

- We guarantee a number between 1 and 6
- 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?

What about different dice?

Making our programs easier to change

- If we want to use much of the same code for a 20 sided die, what would we need to change?
- If we replace all 6s with 20s . . .
- How long does it take to go through all of your code, find all the references to 6 and then change them?
- Are you sure you got them all?
- We want a smarter solution!

The dice size might be a constant

Constants allow us to use a value throughout our code

```
#define DICE_SIZE 6

int main (void) {
    int dieOne;
    // Read user's rolls
    printf("Please enter your first die roll: ");
    scanf("%d", &dieOne);

    // Use mod to keep result within dice range
    dieOne = (dieOne % DICE_SIZE);
    if (dieOne == 0) {
        dieOne = DICE_SIZE;
    }
}
```

The dice size could also be a variable

A Variable can be changed

```
int main (void) {
     int diceSize = 6;
     int dieOne;
     // Read user's rolls
     printf("Please enter your first die roll: ");
     scanf("%d", &dieOne);

     // Use mod to keep result within dice range
     dieOne = (dieOne % diceSize);
     if (dieOne == 0) {
          dieOne = diceSize;
     }
}
```

Constants vs Variables

Constants never change!

- Usually only used for specific types of numbers
- PI, Speed of Light, e, etc

Variables can still be set once and then used throughout a program

The Upgraded Dice Checker

- The user can set the size of the dice
- The user can enter any number and it will produce a valid roll
- The program will still report back success, tie or failure

Starting from our previous Dice Checker program, it will be easy to make some modifications to give it some new capabilities

Setting up

We’ll start with our plan, as usual

```c
// 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>
```

Variables and Constants

Set up the Target constant and the variables

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

int main (void) {
     int diceSize;
     int dieOne;
     int dieTwo;
     int total;
}
```

Taking user input

Dice size and the two rolls will be taken as input

```
// Find out the Dice Size from the user
printf("How many sides are on the dice? ");
scanf ("%d", &diceSize);

// Process the first die roll
printf("Please enter your first die roll:  ");
scanf("%d", &dieOne);
// check and fix the die roll
if (dieOne < 1 || dieOne > diceSize) {
    printf("%d is not a valid roll for a D%d.\n", dieOne, diceSize);
    dieOne = (dieOne % diceSize);
    if (dieOne == 0) {
        dieOne = diceSize;
    }
}
```

Calculate and report the total

This is identical to last week's code

```
// calculate the total and report it
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 if (total == SECRET_TARGET) {
    // tie
    printf("Skill roll tied!\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
- 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)