
COMP1511 - Programming Fundamentals

— Week 2 - Lecture 4 —

What did we learn last lecture?

- **EVERYTHING!** - A recap of the C we've seen so far
- Problem Solving
- Continuing work with if and else statements
- Showing use of `#define` constants
- Using some Relational and Logical Operators
- Showing some use of Modulus

What are we covering today?

Looping

- Repetitive tasks shouldn't require repetitive coding
- "**while**" loops
- How to start and stop loops
- Some interesting things we can do with them

Executing the same code more than once

Sometimes we need to repeat our work

- C normally executes in order, line by line
- `if` statements allow us to “turn on or off” parts of our code
- But up until now, we don’t have a way to repeat code
- Copy-pasting the same code again and again is not a feasible solution

While Loops

```
// expression is checked at the start of every loop
while (expression) {
    // this will run again and again
    // until the expression is evaluated as false
}
// When the program reaches this }, it will jump
// back to the start of the while loop
```

While Loops

“`while`” is a C keyword that lets us loop code

- Format is very similar to an `if` statement
- The “question” in the (brackets) functions very similarly
- If it’s true, the body of the `while` loop will run
- If it’s false, the body won’t run and the program will continue
- Once a while reaches the end of its `{ }` it will start again

While Loop Control

We can use a variable to control how many times a while loop runs

- We call this variable a "loop counter"
- It's an `int` that's declared outside the loop
- It's "termination condition" can be checked in the while expression
- It will be updated inside the loop

We can also use a variable to decide to exit a loop at any time

- We call this variable a "sentinel"
- It's like an on/off switch for the loop

While Loop with a Loop Counter

```
// an integer outside the loop
int counter = 0;

while (counter < 10) {
    // this code has run counter number of times

    counter = counter + 1;
}
// When counter hits 10 and the loop's test fails
// the program will exit the loop
```


While Loops and Termination

It's actually very easy to make a program that goes forever

Consider the following while loop:

```
while (1 < 2) {  
    // Never going to give you up  
    // Never going to let you down . . .  
}
```

Using a Sentinel Variable with While Loops

A sentinel is a variable we use to decide when to exit a while loop

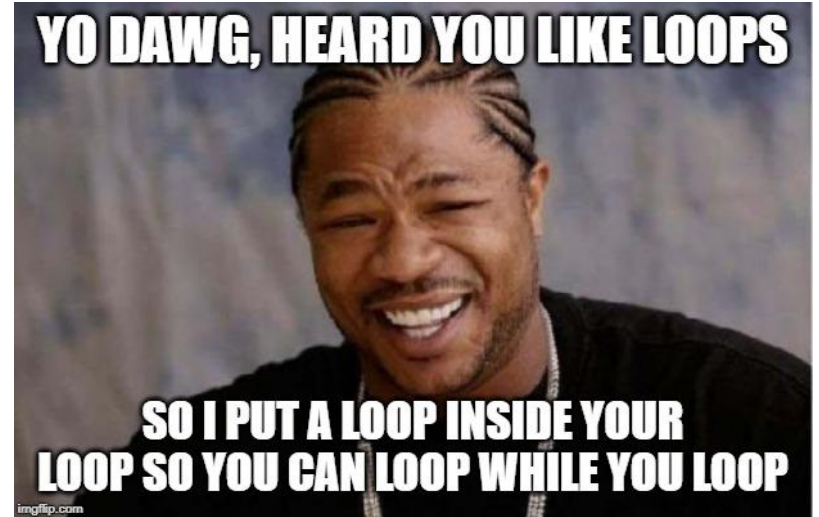
```
// an integer outside the loop
int endLoop = 0;

// The loop will exit if it reads an odd number
while (endLoop == 0) {
    int inputNumber;
    printf("Please type in a number: ");
    scanf("%d", &inputNumber);
    if (inputNumber % 2 == 0) {
        printf("Number is even.\n");
    } else {
        printf("Number is odd.\n");
        endLoop = 1;
    }
}
```

While Loops inside While Loops

If we put a loop inside a loop . . .

- Each time a loop runs
- It runs the other loop
- The inside loop ends up running a LOT of times



A loop within a loop

```
int y = 0;
// loop through and print multiple rows
while (y < 10) { // we have printed y rows
    // print a single row
    int x = 0;
    while (x < 10) { // we have printed x stars in this row
        printf("*");
        x = x + 1;
    }
    // the row is finished, start the next line
    printf("\n");
    y = y + 1;
}
```

Drawing a grid of stars

The previous slide's code:

- Sets up a loop using **y**
- In each loop of **y**, sets up a loop using **x**
- The **x** loop writes multiple *****s to the terminal
- Then the **y** loop finishes, writing `\n` so the line ends

What do the curly braces do?

What goes on inside the curly braces stays inside the curly braces.

- Look closely at the declaration of `int x` in the grid drawing code
- The use of `x` is contained inside a set of curly braces `{ }`
- This means that `x` will only exist inside those braces
- The variable `x` will actually disappear each time the `y` loop finishes!

Curly braces create the "scope" of a program

- Anything created inside them only lasts as long as they do!

Break Time!

While loops, if statements etc, it's all code!

- An `if` statement is some code
- A `while` loop is also some code

This means that you can:

- Put `ifs` inside `while` loops
- Put `while` loops inside `ifs` or `elses`
- Put `while` loops inside `while` loops inside `if` statements etc etc etc!
- Just watch out for confusing ourselves!

Dice Statistics, a Looping Program

The following program:

I need a program that will show me all the different ways to roll two dice

If I pick a number, it will tell me all the ways those two dice can reach that total

It will also tell me what my odds are of rolling that number

Break it down

What components will we need?

- We need all possible values of the two dice
- We need all possible totals of adding them together
- Seems like we're going to be looping through all the values of one die and adding them to all the values of the other die

Let's start with this simple program then go for our bigger goals later

Code for all dice rolls

```
int main (void) {
    int diceOneSize;
    int diceTwoSize;

    // User decides the two dice sizes
    printf("Please enter the size of the first die: ");
    scanf("%d", &diceOneSize);
    printf("Please enter the size of the second die: ");
    scanf("%d", &diceTwoSize);

    // Then loop through both dice
```

Code for all dice rolls continued

```
// loop through and see all the values that the two dice can roll
int die1 = 1;
while (die1 <= diceOneSize) { // seen die1 - 1 values
    int die2 = 1;
    while (die2 <= diceTwoSize) { // seen die2 - 1 values
        printf(
            "%d, %d. Total: %d\n",
            die1, die2, die1 + die2
        );
        die2++;
    }
    die1++;
}
```

Quick Pause for new C syntax

Incrementing just got a little easier

```
int die1 = 0;
int die2 = 0;

// The following two lines have the
// same effect on their variables
die1 = die1 + 1;
die2++;

// both variables now == 1
```

We can now see all possible dice rolls

- We have all possibilities listed
- We know all the totals
- We could also count how many times the dice were rolled

Let's try now isolating a single target number

- Check the targets of the rolls and output only if they match our target value

Now with a target number

```
int main (void) {
    int diceOneSize;
    int diceTwoSize;
    int targetValue;

    // User decides the two dice sizes and target
    printf("Please enter the size of the first die: ");
    scanf("%d", &diceOneSize);
    printf("Please enter the size of the second die: ");
    scanf("%d", &diceTwoSize);
    printf("Please enter the target value: ");
    scanf("%d", &targetValue);
```

Output the rolls that match the target

```
// loop through and output rolls with totals that match the target
int die1 = 1;
while (die1 <= diceOneSize) { // seen die1 - 1 values
    int die2 = 1;
    while (die2 <= diceTwoSize) { // seen die2 - 1 values
        int total = die1 + die2;
        if (total == targetValue) {
            printf(
                "%d, %d. Total: %d\n",
                die1, die2, total
            );
        }
        die2++;
    } // die2 = diceTwoSize + 1
    die1++;
} // die1 = diceOneSize + 1
```

Getting there!

We now have a program that can identify the correct rolls

- If we want the odds, we just compare the target rolls vs the rest
- If we count the number of rolls that added to the target value
- And we count the total number of rolls
- We can do some basic maths and divide the successful rolls by the total
- That should give us our chances of getting that number

How do we keep track of success vs failure?

We can count using ints

- We can keep a counting variable outside the loop
- This will increment only on successes
- We can either calculate or count our total
- Dividing them will give us the fraction chance of rolling our target number

Measuring Successes

Adding some variables to count results

- **integers** (`diceOneSize`, `diceTwoSize`) for the two dice sizes
- **integer** (`targetValue`) for the target value
- **integer** (`numSuccesses`) for the number of successes
- **integer** (`numRolls`) for the number of rolls

Making sure our loop records results

```
// loop through and output rolls with totals that match the target
int die1 = 1;
while (die1 <= diceOneSize) { // seen die1 - 1 values
    int die2 = 1;
    while (die2 <= diceTwoSize) { // seen die2 - 1 values
        numRolls++;
        int total = die1 + die2;
        if (total == targetValue) { // target match
            numSuccesses++;
            printf("%d, %d. Total: %d\n",
                die1, die2, total);
        }
        die2++;
    }
    die1++;
}
```

Output our Percentage

```
// Calculate percentage chance of success
int percentage = numSuccesses/numRolls * 100;
printf("Percentage chance of getting your target number is: %d\n",
       percentage);
```

There's an issue with the previous code ...

Did you notice the issue?

- Our code outputs 0 percent a lot more than it should
- This is even after we know it's counting the successes correctly

Integers do weird things with division in C

- After a division, the integers will throw away any fractions
- Since our "`numSuccesses/numRolls`" will always be between zero and 1
- Our result can only be the integers 0 or 1
- And anything less than 1 will end up having its fraction dropped!

Doubles to the rescue

Luckily we have a variable type that will store a fraction

- Result of a division will be a double if one of the variables in it is a double
- We could change one of the variables in our division to a double
- This could be done in the declaration of the variable
- But we can also just do it at the point it is used!

```
// int * double = double
// The second number will appear as a double to the division!
int percentage = numSuccesses/(numRolls * 1.0) * 100;
```

The Challenge ... did we need to do all this work?

This program didn't actually need everything we did today

- There's a much simpler way to list the rolls that sum to a target number
- There's also a much simpler way to find the total number of rolls
- If we just use a bit more maths and less raw coding . . .

See what you can come up with!

What did we learn today?

While Loops

- Repeating execution of code
- We've made some loops
- We've shown how to loop inside other loops
- We've shown different ways to end loops