

# ENGG1003 - Friday Week 2

Fixing the Mistakes I Made on Tuesday  
Lets do Some Examples

Brenton Schulz

University of Newcastle

March 7, 2019

# Example: Testing Factors

Task: Write a C program which reads two integers from the user and tests if the second is a factor of the first, printing the result.

Eg: If the user enters 12 and 3 the program would print something like:

```
3 is a factor of 12
```

# Example: Testing Factors

Lets convert the problem down into “high level” pseudocode:

```
BEGIN
    integer x
    integer y
    READ x from the user
    READ y from the user
    Test if y is a factor of x
    Tell the user the result
END
```

Can every line of pseudocode be turned into C?  
How are we going to test if one number is a factor of another?

# Example: Testing Factors

Lets make an attempt at factor testing, perhaps start with:

Definition: *Factors* are numbers we can multiply together to make another number. Eg: 2 and 3 are factors of 6 as  $2 \times 3 = 6$ .

Is this definition useful for this problem? Is it easy to turn this definition into C code?

## Example: Testing Factors

No, not really. C can't easily do “can I find another integer that multiplies with  $y$  to make  $x$ ”. That instruction is not *executable*.

Lets try again:

An integer,  $y$ , is a factor of a number,  $x$ , if the integer evaluation of  $x \div y$  has no remainder.

Can *this* become C code?

# Example: Testing Factors

YES! We can use the modulus operator, `%`, to test if a division has a remainder with the code:

```
1 if ( (x % y) == 0) {  
2     // y is a factor of x  
3 }
```

# Modulus Example - Factor Testing

With this fact, lets tweak the pseudocode:

```
BEGIN
  integer x
  integer y
  READ x from the user
  READ y from the user
  IF (x % y) == 0
    PRINT y is a factor of x
  ELSE
    PRINT y is NOT a factor of x
  ENDIF
END
```

# Modulus Example - Factor Testing

...and convert each line to C:

(printf(); output changed to fit on slide)

(C Code is almost missing #include <stdio.h>)

## Pseudocode

```
BEGIN
integer x
integer y
READ x from the user
READ y from the user
IF (x % y) == 0
    PRINT y is a factor
ELSE
    PRINT y isn't a factor
ENDIF
END
```

## C Code

```
int main() {
    int x;
    int y;
    scanf("%d", &x);
    scanf("%d", &y);
    if( (x % y) == 0 ) {
        printf("%d is a factor\n", y);
    } else {
        printf("%d isn't factor\n", y);
    }
}
```



# Modulus Example - Code with Prompt

```
1 #include <stdio.h>
2 int main() {
3     int x, y;
4     printf("Enter an integer: ");
5     scanf("%d", &x);
6     printf("Enter another integer: ");
7     scanf("%d", &y);
8     if(x % y == 0) { // ie: if the remainder is zero
9         printf("%d is a factor of %d\n", y, x);
10    } else {
11        printf("%d is NOT a factor of %d\n", y, x);
12    }
13    return 0;
14 }
```

Listing 1: factorTest.c

# Factor Testing Discussion

- ▶ Is this code *robust*?
- ▶ Can the user enter numbers which make the code produce the wrong number?
- ▶ What happens if  $y > x$ ?
  - ▶ It might be fine, it might not. Have a think about it and do some testing in the lab
- ▶ Get in the habit of testing code, both with “expected” input and “weird” input
- ▶ What happens if you enter letters instead of numbers? Or negatives? Or ask where the bathroom is?

# Modulus Example 2 - Factorisation

- ▶ We now know how to test if a number is a factor of another
- ▶ What about a full factorisation?

**Task:** Write a C program which reads an integer from the user and outputs all of its factors.

# Modulus Example 2 - Factorisation

- ▶ How does factorisation happen, anyway?
  - ▶ Normally? In your head. “Dream up” the answer.
  - ▶ On a computer? We need to *brute force* it.
    - ▶ ie: Simply test every integer which might work
  - ▶ All factors of a number,  $k$  are in the range  $[1,k]$ 
    - ▶ 1 and  $k$  are always factors, so explicitly testing them is optional
  - ▶ They are also all integers
  - ▶ Thankfully, this is a finite number of tests
- ▶ Faster algorithms *might* exist
  - ▶ You would need to consult number theory literature
  - ▶ This is beyond my (Brenton's) knowledge

# Modulus Example 2 - Factorisation

- ▶ How can we test lots of numbers?
  - ▶ The program needs to count
  - ▶ The input is unknown, so we need a loop
  - ▶ ie: we can't "hard code" counting when we don't know when to stop!

# Modulus Example - Factorisation

Lets write some pseudocode for the factorisation problem. We start with something really “high level”:

BEGIN

Integer x

READ a value for x from the user

Calculate x's factors

PRINT x's factors

END

# Modulus Example - Factorisation

Now lets *imply* a loop, but not explicitly write it:

BEGIN

Integer x

READ a value for x from the user

Test if every integer from 1 to x is a factor of x

PRINT x's factors

END

# Modulus Example - Factorisation

- ▶ How do we code “Test if every integer from 1 to  $x$  is a factor of  $x$ ”?
- ▶ Well, we know how to test one integer,  $n$ :  
`if ( x % n == 0 )`
- ▶ How do we *count*?
- ▶ With what we know so far? A `while()` loop!

```
integer count = 1
WHILE count <= n
    count = count + 1 // Counts from 1 to n
ENDWHILE
```

- ▶ A “better” method will be shown later



# Modulus Example - Factorisation

... getting closer, lets write a loop:

```
BEGIN
  integer x
  integer count = 1
  READ a value for x from the user
  WHILE (count <= x)
    IF (x % count) == 0
      PRINT <count> is a factor of <x>
    ENDIF
    count = count + 1
  ENDWHILE
END
```

# Modulus Example - Factorisation

- ▶ Notice the `PRINT` statement *inside* the loop
  - ▶ Previous pseudocode has factorisation and printing as different steps
- ▶ This means we don't need to remember a list of factors as we go
- ▶ We will learn how to work with lists later
  - ▶ C calls a list of variables an *array*
- ▶ Lets read and run the final program...

# Modulus Example - Factorisation

```
1 #include <stdio.h>
2 int main() {
3     int count = 1, x;
4     printf("Enter an integer to factorise: ");
5     scanf("%d", &x);
6     while(count <= x) {
7         if(x % count == 0) // If the remainder is zero
8             printf("%d is a factor of %d\n", count, x);
9         count++;
10    }
11    return 0;
12 }
```

Listing 2: factors.c

# Modulus Example - Factorisation

## Example output:

```
Enter an integer to factorise: 76545478
1 is a factor of 76545478
2 is a factor of 76545478
38272739 is a factor of 76545478
76545478 is a factor of 76545478
```

Is it correct? Check output with Wolfram Alpha

Observation: A modified version of this code (with `unsigned int`) only takes 15 seconds to factorise 4294967294

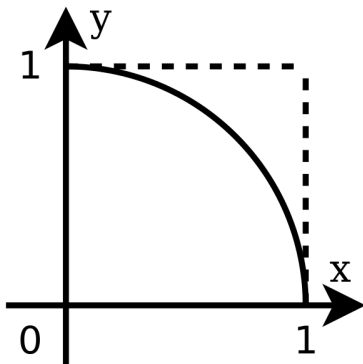
# Discussion

- ▶ Pay close attention to the value of `count`
- ▶ It is initialised to 1
- ▶ It is used *before* incrementing it
- ▶ Incrementing is the last thing in the loop
- ▶ The loop *condition* is “less than *or equal to*” so that `x` itself is explicitly tested as a factor
  - ▶ Remember that 1 and `x` are always factors of `x`?

# Case Study: Calculating $\pi$

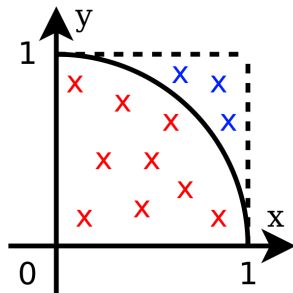
Consider a quadrant of a unit circle ( $r = 1$ ) with a square around it:

- ▶ Area of the square  
 $A_1 = 1$
- ▶ Area of the circle quadrant  $A_2 = \frac{\pi r^2}{4} = \frac{\pi}{4}$
- ▶ Ratio of areas  $\frac{A_2}{A_1} = \frac{\pi}{4}$
- ▶ Therefore  $\pi = 4 \times \frac{A_2}{A_1}$



# Case Study: Calculating $\pi$

- ▶ We can't calculate the area ratio without knowing  $\pi$
- ▶ Estimate it by:
  - ▶ Randomly picking many points inside the square
  - ▶ Test if the point is inside the circle with  $x^2 + y^2 < 1$



- ▶ 
$$\pi \approx 4 \times \frac{\text{Number of points which land inside circle}}{\text{Total number of points tested}} = 4 \times \frac{9}{12} = 3$$

# Algorithm for Calculating $\pi$

```
BEGIN
  integer countTotal = 0
  integer countInside = 0
  WHILE countTotal < A large number
    x = random number between 0 and 1
    y = random number between 0 and 1
    countTotal = countTotal + 1
    IF x*x + y*y < 1
      countInside = countInside + 1
    ENDIF
  ENDWHILE
  pi = 4*countInside/countTotal
  PRINT pi
END
```