

# ENGG1003 - Tuesday Week 2

Calculating Pi  
C Arithmetic  
Datatypes

Brenton Schulz

University of Newcastle

February 24, 2019

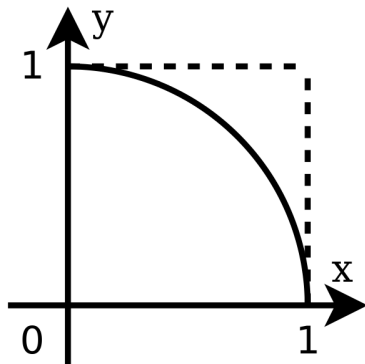
# Case Study: Calculating $\pi$

- ▶ Computers are *really* good at repetitive things
- ▶ Lets use this fact to calculate  $\pi$  using a “monte-carlo” method
  - ▶ Informally, these are methods which solve problems by repeating the same thing with different inputs until patterns emerge
  - ▶ It could repeat millions or billions of times
  - ▶ Name comes from the Monaco Principality’s high concentration of casinos
- ▶ Algorithm pseudocode will be written before an implementation in C

# Case Study: Calculating $\pi$

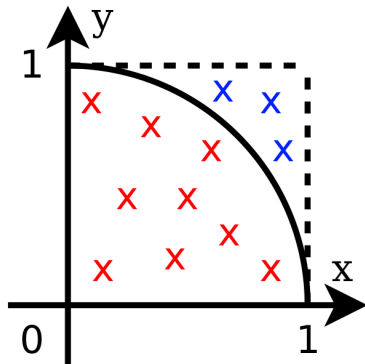
Consider a quadrant of a unit circle ( $r = 1$ ) with a square around it:

- ▶ 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 $\pi$

- ▶ How can the above *mathematics* be turned into an *algorithm*?
- ▶ The algorithm needs to repeat the *same thing* multiple times
  - ▶ This implies use of a loop
  - ▶ The loop's *exit condition* needs to be defined
- ▶ As the loop repeats, we need to keep track of the following *variables*:
  - ▶ The number of points tested
  - ▶ The number of points which landed inside the circle
  - ▶ The  $(x, y)$  coordinates of the point under test

# Algorithm for Calculating $\pi$

- ▶ The number of points tested will be an *integer*, we will call it `countTotal`
- ▶ The number of points found to be inside the circle is also an integer, we will call it `countInside`
- ▶ Before these variables are used they should be *initialised*
  - ▶ ie: The algorithm will explicitly include `countTotal = 0` and `countInside = 0`
  - ▶ So-called *uninitialised* variables have undefined (or random) values

# Algorithm for Calculating $\pi$

- ▶ Incrementing the `countInside` variable is *conditional* on the values of  $x$  and  $y$ 
  - ▶ This implies `IF...ENDIF` *flow control*
- ▶ The condition on incrementing `countInside` is  $x^2 + y^2 < 1$
- ▶ Incrementing a variable in pseudocode takes the form:
  - ▶ `variable = variable + 1`
  - ▶ This can be read as “variable *becomes* variable plus 1”
  - ▶ Mathematics would write:  $x_{n+1} = x_n + 1$

- ▶ The point under test needs two “real” variables:  $x$  and  $y$ 
  - ▶ “Real” is the generic term for a number with integer and fractional components. Eg: 1.45
- ▶ These values take new random values each loop
- ▶ The pseudocode doesn't need to describe how a random number is generated
  - ▶ Stating: “ $x =$  a random number between 0 and 1” is totally acceptable
- ▶ At the end of the algorithm the final step will be  $\pi = 4 \times \frac{\text{countInside}}{\text{countTotal}}$



# 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  $x^2 + y^2 < 1$ 
      countInside = countInside + 1
    ENDIF
  ENDWHILE
  pi = 4*countInside/countTotal
  PRINT pi
END
```

# Missing Knowledge for C Implementation

- ▶ More information about arithmetic
  - ▶ Relational operators look useful
  - ▶ Is there a neat way to do `count=count+1`?
  - ▶ `countInside` and `countTotal` are both integers. What happens when we divide?
- ▶ Datatypes and how they are handled in arithmetic statements
- ▶ How do we generate random numbers?
- ▶ Syntax for WHILE loops and IF statements

# C Arithmetic

- ▶ Basic arithmetic was seen in the lab
  - ▶ You all did the lab, right?

Operation	C Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/

Table: Basic arithmetic operators in C

- ▶ Complex expressions can be built from these operators and parentheses

# C Arithmetic

## Examples:

$$\begin{array}{ll} z = x^2 + 5(y + b) & z = x * x + 5 * (y + b) ; \\ u = \frac{x+1}{x-1} & u = (x + 1) / (x - 1) ; \\ v = z^3 + \frac{5(y+b)}{2} & v = z * z * z + (5 * (y + b)) / 2 ; \end{array}$$

- ▶ Multiplication is not assumed. If you write  $5(y+b)$  the compiler will generate a syntax error.
- ▶ To be valid C expressions the semicolon is required.

# C Arithmetic

- ▶ C supports two time-saving *unary* operators:
  - ▶ Very useful in loops.

Operation	C Syntax	Replaces
Increment	<code>x++;</code> or <code>++x;</code>	<code>x = x + 1;</code>
Decrement	<code>x--;</code> or <code>--x;</code>	<code>x = x - 1;</code>

- ▶ It also supports the following shorthand syntax:

<code>x = x + y;</code>	<code>x += y;</code>
<code>x = x - y;</code>	<code>x -= y;</code>
<code>x = x * y;</code>	<code>x *= y;</code>
<code>x = x / y;</code>	<code>x /= y;</code>

# C Arithmetic

What's the difference between  $x++$  and  $++x$ ?

- ▶  $x++$  is a *post-increment*
- ▶  $++x$  is a *pre-increment*
- ▶ If they appear in an arithmetic expression, pre-increment is processed *before* the variable is used and post-increment is processed *after* it is used.
- ▶ In isolation there is no difference.

# Increment Example

```
1 #include <stdio.h>
2 int main() {
3     int x = 0;
4     int y = 0;
5     int z = 0;
6     y = ++x + 10;
7     printf("Pre-increment: %d\n", y);
8     y = z++ + 10;
9     printf("Post-increment: %d\n", y);
10    return 0;
11 }
```

Pre/post-inc/decrements have many applications,  
more details in coming weeks.

# Modulus

- ▶ Computers frequently only deal with integers
- ▶ Integer division in C ignores (truncates) any fractional component
- ▶ The *modulus* operator provides the remainder after division
  - ▶ Implemented with the % character
  - ▶  $a \% b = \text{remainder of } a / b$
  - ▶ Very useful for tasks performed every *nth* loop
- ▶ Example:
  - ▶  $10 / 3 = 3$
  - ▶  $10 \% 3 = 1$



# Modulus Example - Factor Testing

```
1 #include <stdio.h>
2 int main() {
3     int x;
4     int y;
5     printf("Enter an integer: ");
6     scanf("%d", &x);
7     printf("Enter another integer: ");
8     scanf("%d", &y);
9     if(x % y == 0) { // ie: if the remainder is zero
10         printf("%d is a factor of %d\n", y, x);
11     } else {
12         printf("%d is NOT a factor of %d\n", y, x);
13     }
14     return 0;
15 }
```

# Relational Operators

- C supports six *relational* operators:

Operation	C Symbol
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	==
Not equal to	!=

Table: Relational operators in C

# Relational Operators

- ▶ The result of a relational operation is 0 or 1
  - ▶ C treats 0 as Boolean FALSE and non-zero as TRUE
- ▶ They are typically used as flow control conditions
  - ▶ `if(condition) {statements}`
  - ▶ `while(condition) {statements}`
- ▶ While we're here: the above is the correct syntax for IF and WHILE flow control in C

# Modulus Example 1 - Printing Every *nth* Loop

```
1 #include <stdio.h>
2 int main() {
3     int x = 0;
4     while(x < 1000)
5     {
6         // Presumably something useful is done with x
7         // inside this loop
8         if(x%100 == 0)
9             printf("%d\n", x);
10    }
11    return 0;
12 }
```

# Modulus Example 2 - Finding Factors

```
1 #include <stdio.h>
2 int main() {
3     int input;
4     int x;
5     printf("Enter an integer to factorise: ");
6     scanf("%d", &input);
7     x = input;
8     while(x > 0) {
9         if(input % x == 0) // ie: if the remainder is zero
10             printf("%d is a factor of %d\n", x, input);
11         x--;
12     }
13     return 0;
14 }
```

Observe that the `while()` loop loops over every value of `x` from `input` to 1. We will discuss a flow control method designed for this (the `for` loop) later.

# C Arithmetic Operator Precedence

- ▶ C has an “order of operations”
- ▶ eg:  $1+5*2$  evaluates to 11
- ▶ Multiplication and division first
- ▶ Addition and subtraction second
- ▶ Relational operators somewhere below that
- ▶ If in doubt: force order with parentheses
  - ▶ This makes the code more readable
  - ▶ It doesn't cost you anything
  - ▶ C compilers understand algebra and will optimise inefficient expressions automatically

# Data Types

- ▶ In C, all variables are *declared* before use
- ▶ Declaration specifies the variable's:
  - ▶ Datatype
  - ▶ Name
  - ▶ An *initialisation value* (optional)
    - ▶ Always assume uninitialised variables have random values! Behaviour varies between compilers and target platforms.
- ▶ C is a “strongly-typed” language
  - ▶ Every variable has a fixed type

# Integer Data Types

- ▶ There are several integer data types
- ▶ They vary by their:
  - ▶ Size
  - ▶ Support for negative numbers
- ▶ C integer types can be 1, 2, 4, or 8 bytes long
  - ▶ The sizes vary by platform
- ▶ Each type can be *signed* or *unsigned*
  - ▶ Unsigned numbers are never negative but you get double the value range



# Integer Data Types

- ▶ C includes the `sizeof()` expression so that a program can discover the size of a data type
- ▶ The size (given in bytes) defines the *value range*
- ▶ On a modern 64-bit Linux desktop machine:

Type	Bytes	Bits	Value Range
char	1	8	-128, +127
short	2	16	-65536, 65535
int	4	32	-2147483648, 2147483647
long	8	64	-9223372036854775808, ...

# Unsigned Integers

- ▶ Unsigned integers are *always positive*
- ▶ They are the same size as their signed counterparts
- ▶ eg: `unsigned char` is 1 byte and has a value range of 0 to 255

# Why Care About Data Types?

- ▶ You may be thinking “why not make everything a `long`?”
- ▶ Answer: speed and memory
- ▶ Smaller types use less RAM
- ▶ Arithmetic on a type larger than the target platform's *native size* is slow
- ▶ Matters if you store millions of the same type
- ▶ Makes a *huge* difference on embedded targets
  - ▶ Don't declare 32-bit variables on an 8-bit AVR microcontroller *unless you have to*

# Integer Division Example

What is the output of each `printf()` statement?

```
1 #include <stdio.h>
2 int main() {
3     printf("%d\n", 9/10);
4     printf("%f\n", 9/10);
5     printf("%f\n", 9.0/10);
6     printf("%f\n", 9/10.0);
7     printf("%f\n", (float)9/10);
8     return 0;
9 }
```

TODO: Keep hacking until we've implemented  $\pi$ .