ENGG1003 - Tuesday Week 2

Calculating Pi
C Arithmetic
Datatypes

Brenton Schulz

University of Newcastle

February 27, 2019

Before we start...

- ▶ Today's lecture is information overload
  - ▶ It is a long list of "stuff" to rote learn
  - ▶ Half of programming is just "playing with Lego" with this "stuff"
  - ▶ There is a lot to learn before we can solve interesting problems
- ▶ It will probably push you out of your comfort zone
- ▶ Lab experience will hopefully rebuild your confidence
  - ▶ Ideally, before the HECS census date
 - ▶ I have 2 labs to do this ... oh crap

Case Study: Calulating π

- ▶ Computers are *really* good at repetitive things
- ▶ Lets use this fact to calculate π 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 π

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 $\pi = 4 \times \frac{A2}{A1}$
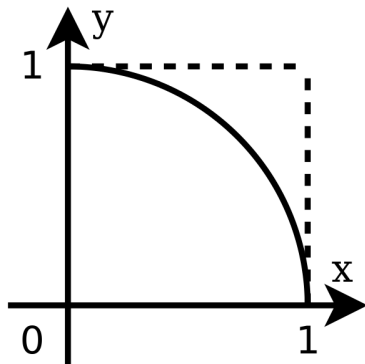
Case Study: Calulating π

- We can't calculate the area ratio without knowing π
- Estimate it by:
    - Randomly picking many points inside the square
    - Test if the point is inside the circle with $x^2 + y^2 < 1$
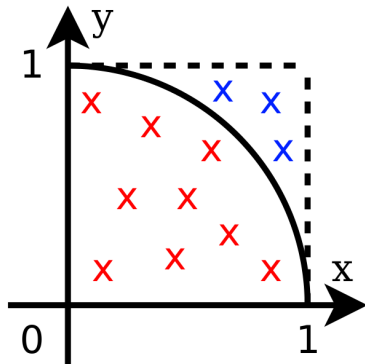
- $\pi \approx 4 \times \dfrac{\text{Number of points which land inside circle}}{\text{Total number of points tested}} =$ $4 \times \frac{9}{12} = 3$

Algorithm for Calculating π

- ▶ How can the above *mathematics* be turned into an *algorithm*?
  - ▶ **NB:** You only have to understand the 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 π

- ▶ 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 π

- ▶ 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"
  - ▶ Maths people would write: $x_{n+1} = x_n + 1$

- ▶ The point under test needs two "real" variables: x and y
  - ▶ "Real" comes from mathematics: any 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 $\pi = 4 \times \frac{\texttt{countInside}}{\texttt{countTotal}}$

Algorithm for Calculating π

```
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
```

Missing Knowledge for C Implementation

▶ More information about arithmetic
  ▶ Relational operators (less/greater-than) 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:

$$z = x^2 + 5(y + b) \qquad \texttt{z = x*x + 5*(y + b);}$$
$$u = \frac{x+1}{x-1} \qquad \texttt{u = (x + 1)/(x - 1);}$$
$$v = z^3 + \frac{5(y+b)}{2} \qquad \texttt{v = z*z*z+(5*(y + b))/2;}$$

- ▶ 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	x++; or ++x;	x = x + 1;
Decrement	x--; or --x;	x = x - 1;

▶ It also supports the following shorthand syntax:

x = x + y;	x += y;
x = x - y;	x -= y;
x = x * y;	x *= y;
x = x / y;	x /= y;

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 ignores (truncates) any fractional component
- ▶ The *modulus* operator provides the remainder after division
  - ▶ Implemented with the `%` character
  - ▶ `a % b` = remainder of `a / b`
  - ▶ Very useful for tasks performed every *nth* loop
- ▶ Example:
  - ▶ `10 / 3 = 3`
  - ▶ `10 % 3 = 1`

Modulus Example 1 - Printing Every *nth* Loop

```c
#include <stdio.h>
int main() {
  int x = 0;
  while(x < 1000)
  {
    // Presumably something useful is done with x
    // inside this loop
    if(x%100 == 0)
      printf("%d\n", x);
  }
  return 0;
}
```

Modulus Example 2 - Factor Testing

```c
#include <stdio.h>
int main() {
  int x;
  int y;
  printf("Enter an integer: ");
  scanf("%d", &x);
  printf("Enter another integer: ");
  scanf("%d", &y);
  if(x % y == 0) { // ie: if the remainder is zero
    printf("%d is a factor of %d\n", y, x);
  } else {
    printf("%d is NOT a factor of %d\n", y, x);
  }
  return 0;
}
```

Modulus Example 3 - Finding Factors

```c
#include <stdio.h>
int main() {
  int input;
  int x;
  printf("Enter an integer to factorise: ");
  scanf("%d", &input);
  x = input;
  while(x > 0) {
    if(input % x == 0) // ie: if the remainder is zero
      printf("%d is a factor of %d\n", x, input);
    x--;
  }
  return 0;
}
```

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.

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	$!=$

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

Boolean Operators

- ▶ We discussed Boolean algebra last week
- ▶ It has three operators: AND, OR, NOT
- ▶ In C, these are implemented with:

Operator	C Characters
AND	& &
OR	| |
NOT	!

- ▶ I know the | character as a "pipe" symbol or "vertical slash"
  - ▶ Ask your demonstrator to find it on the keyboard

Conditions

- ▶ We have seen that `if()` and `while()` statements need *conditions*
- ▶ A condition can be any combination of variables, operators, and literals (numbers, defined later)
- ▶ The following are all valid:
  - ▶ `if(x > 29)`
  - ▶ `if(x == 2)`
  - ▶ `if(!(x == y))`
  - ▶ `if( (x > 5) && (x < 20) )`
  - ▶ `if(x = y + z)`
    - ▶ Evaluates `(y + z)`, assigns to `x`, uses result as `if()` condition

C Arithmetic Operator Precedence

▶ C has an "order of operations"
▶ eg: `1+5*2` evaluates to `11`
  ▶ You remember BODMAS / PEDMAS, right?
▶ 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

Variable Declaration

- ▶ 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.
    - ▶ eg: int counter = 0;
        - ▶ Type is int
        - ▶ Name is counter
        - ▶ Initial value is 0
- ▶ C is a "strongly-typed" language
    - ▶ Every variable has a fixed type

Variable Naming Rules

- ▶ Variable names:
    - ▶ Are case sensitive
    - ▶ Must **not** be a C keywoard (eg: `int, char, while,` etc)
    - ▶ Must **not** contain special characters (`*-+/,` etc)
    - ▶ Must begin with a letter
    - ▶ Can contain numbers
    - ▶ Can contain underscores
- ▶ What you do with those rules is up to you
- ▶ Modern practice encourages:
    - ▶ camelCaseVariableNames
    - ▶ names_with_underscores

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
  - ▶ `int` and `long` sizes vary by platform
  - ▶ Larger sizes store larger numbers, use more RAM
- ▶ Each type can be *signed* or *unsigned*
  - ▶ Unsigned numbers are always positive but you get double the *value range*

Diversion: Binary Nomenclature

▶ The value range is a result of the underlying binary storage mechanism

▶ A single binary digit is called a *bit*

▶ There are 8 bits in a *byte*

▶ In programming we use the "power of two" definitions of kB, MB, etc:
  ▶ 1 kilobyte is $2^{10} = 1024$ bytes
  ▶ 1 Megabyte is $2^{20} = 1048576$ bytes
  ▶ 1 Gigabyte is $2^{30} = 1073741824$ bytes
  ▶ (Advanced) These numbers look better in hex: 0x3FF, 0xFFFFF, etc.

Diversion: Binary Nomenclature

▶ Observe that kilobyte, Megabyte, Gigabyte, etc use scientific prefixes

▶ These *normally* mean a power of 10:
  ▶ kilo- $= 10^3$
  ▶ Mega- $= 10^6$
  ▶ Giga- $= 10^9$
  ▶ ...etc (see the inside cover of a physics text)

▶ Computer science stole these terms and re-defined them

Diversion: Binary Nomenclature

▶ This has made some people *illogically angry*
▶ Instead, we can use a more modern standard:
   ▶ 1024 bytes $= 1$ kibiByte (KiB)
   ▶ 1024^2 bytes $= 1$ Mebibyte (MiB)
   ▶ 1024^3 bytes $= 1$ Gibibyte (GiB)
   ▶ ...etc
▶ Generally speaking:
   ▶ KB (etc) implies powers of two to *engineers*
   ▶ KB (etc) implies powers of ten to *marketing*
      ▶ The number is smaller
      ▶ Hard drive manufacturers, ISPs, etc like this

Integer Data Types

▶ The integer data type ranges can be calculated from the data type's size in bits

▶ For unsigned numbers of bit length n:

$$\text{max} = 2^n - 1 \tag{1}$$

▶ For signed numbers of bit length n:

$$\text{max} = 2^{(n-1)} - 1 \tag{2}$$

$$\text{min} = -2^{(n-1)} \tag{3}$$

▶ Signed numbers are stored in *two's complement* format, covered in ELEC1710

Integer Data Types

▶ C includes the `sizeof()` expression so that a program can discover the size of a data type on a given platform

▶ On a modern 64-bit Linux desktop machine:

Type	Bytes	Bits	Value Range
char	1	8	-128, +127
short	2	16	-32768, 32767
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
▶ The `unsigned` keyword placed before the data type makes that variable unsigned
▶ eg: `unsigned char` is 1 byte and has a value range of `0` to `255`

Unambiguous Integer Data Types

- ▶ Because the standard `int` and `long` data types don't have fixed size unambiguous types exist
- ▶ Under OnlineGDB (ie: Linux with `gcc`) these are defined in `stdint.h` (`#include` it)
- ▶ You will see them used commonly in embedded systems programming (eg: Arduino code)
- ▶ The types are:
    - ▶ `int8_t`
    - ▶ `uint8_t`
    - ▶ `int16_t`
    - ▶ ...etc

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*

Overflow

- ▶ *Overflow* occurs when the result of a calculation is too big to fit into the target type

- ▶ Example: `127 + 1 = −128`

```c
#include <stdio.h>
int main() {
  char x = 127;
  printf("%d\n", x);
  x++;
  printf("%d\n", x);
  return 0;
}
```

- ▶ Message: make variables as small as they can be, but no smaller

Floating Point Data Types

- ▶ To store real numbers C has several *floating point* data types
- ▶ As with integers, try to use the smallest you can get away with
- ▶ Quad precision (`__float128`) is supported in gcc but *very* slow
  - ▶ Not used in this course
- ▶ Run benchmark demonstration

Type	Size	Range	Precision
float	4	1.2×10^{-38} to 3.4×10^{38}	6 dp
double	8	2.3×10^{-308} to 1.7×10^{308}	15 dp

Literals

- ▶ A *literal* is any number written in the code
- ▶ Why not "constant"?
  - ▶ That word means something different
- ▶ Examples:
  - ▶ `x = 5; // 5 is a literal int`
  - ▶ `y = 2.0 / z; // 2.0 is a literal double`
- ▶ By default:
  - ▶ An integer literal is stored as an `int` data type
    - ▶ ie: has the value range and arithmetic limits of `int`
  - ▶ A floating point literal is stored as `double`

Literals

- ▶ Integer literals can be in:
  - ▶ Decimal: `123`
  - ▶ Hexadecimal: `0xA34 // Zero-x`
  - ▶ Octal: `O125 // Capital letter O`
  - ▶ (Hex and octal are covered in ELEC1710)
- ▶ Integer literals can be specified as *unsigned* with the `u` suffix:
  - ▶ `938u`
- ▶ They can also be declared `long` with the `l` suffix:
  - ▶ `3726484l`
  - ▶ The compiler will issue a warning if a literal is too big for `int`

Literals

▶ Floating point literals can be written in many ways:
   ▶ `1.0f` // f suffix forces float
   ▶ `(float)2.3` // Forces float
   ▶ `1.0` // Default to double
   ▶ `1e2` // Double, 1 times 10^2

▶ `1e2` is known as "e-notation"
   ▶ $XeY = X \times 10^{Y}$
   ▶ I will use it *all the time*

▶ Forcing literals to `float` is frequently necessary in embedded systems which lack double precision hardware

Mixing Data Types

▶ C supports arithmetic between different types
▶ Changing a data type is called *casting*
▶ When types are mixed two things can happen:
  ▶ Types get upgraded automatically (*implicit* type casting)
    ▶ Upgrade path is roughly: `short/char` – `int` – `long` – `long long` – `float` – `double`
  ▶ Types get specified manually by the programmer (*explicit* type casting)

Explicit Type Casting

▶ The data type of a variable (or literal) can be forced to change using *type casting*
▶ Write the desired type in parentheses before the variable or constant
▶ Examples:
  ▶ `x = (float)y / k;`
  ▶ `y = (unsigned int)y + 32;`

Format Specifiers

▶ A *format specifier* controls how `printf();`
  converts numerical (or textual) data to a series
  of ASCII characters
▶ Full details are complex, for now just use:
  ▶ `%d` for integer types
  ▶ `%f` for "fixed decimal place" floating point
  ▶ `%e` for e-notation floating point
  ▶ Cast inside `printf()` to suppress compiler
    warnings

Format Specifiers

▶ Casting example:

```
1 long i;
2 // ...
3 printf("%d\n", (int)i); //Breaks when i>2^31
```

▶ %.df produces d decimal places of precision
  ▶ eg:

```
1 float x = 1.23456;
2 printf("%.2f", x); // Prints 1.23
```

Integer Division Example

With *all that* out of the way, what is the output of each printf() statement?

```c
#include <stdio.h>
int main() {
  printf("%d\n", 9/10);
  printf("%f\n", 9/10);
  printf("%f\n", 9.0/10);
  printf("%f\n", 9/10.0);
  printf("%f\n", (float)9/10);
  return 0;
}
```

Random Numbers

- ▶ In C there is a *standard library function* which generates random numbers
- ▶ We will study functions in more detail later
- ▶ To use a library function:
  - ▶ Read the function's documentation
  - ▶ #include the correct *header file*
  - ▶ Take note of the *return value* data type
  - ▶ Add any *compiler flags* (beyond ENGG1003)
  - ▶ Use the function in your code
- ▶ Demonstration: read the rand() man page
  - ▶ Click here

Random Numbers

▶ Observe that `rand();` requires

```
1 #include <stdlib.h>
```

▶ Observe that it returns an `int` between 0 and `RAND_MAX` which, in gcc, is $2^{31} - 1$

▶ Note that is has <u>limits</u> (advanced discussion, beyond ENGG1003)

▶ For us, using the % operator:

```
1 x = rand() % (RANGE+1);
```

produces a *good enough* random number between zero and `RANGE`

Random Numbers

▶ Using the above information, we can roll a standard die with:

```
1 int dieRoll;
2 dieRoll = rand() % 7;
```

▶ Or, if you're a D&D fan, roll a D20 with:

```
1 int D20Roll;
2 D20Roll = rand() % 21;
```

Random Numbers

- ▶ RAND_MAX is a *constant* defined in stdlib.h
- ▶ We can try to generate a floating point number from 0 to 1 with a division:

```
1 float num;
2 num = rand()/RAND_MAX; // This is broken
```

- ▶ Problem: rand() is an integer, RAND_MAX is a (large) integer
  - ▶ The integer division result will always be zero!
- ▶ Solution: use an explicit cast so that a floating point division occurs:

```
1 double num;
2 num = (double)rand()/RAND_MAX;
```

Probably out of time by now, lets implement a π calculator in the lab... In theory you have enough information by now.