ENGG1003 - Lab 2

Brenton Schulz

1 Introduction

This lab leaves you to perform a series of programming exercises on your own. This can be daunting at first but confidence and perseverance will be crucial skills.

The example tasks are mostly "toy" programs. They don't do anything particularly useful because most "real-world" problems are too advanced for this stage of the course. Things will get more interesting in the coming weeks (I hope...).

Do not expect programs to compile (let alone *work*) first go. It is totally normal, even for experienced programmers, for several lines of hand typed code to generate several errors the first time it is compiled.

You will gain experience interpreting the error messages, typically it will be a syntax error due to a missing parenthesis, semicolon, or double quote. Don't be afraid to ask for help and work with other students in the lab.

2 Input-Processing-Output Exercises

These exercises all require the use of `scanf();` to read data, some arithmetic to process data, and `printf();` to print results to the console.

Every task requires the following steps:

1. Work out what variables are required
2. What *type* do the variables need to be?
3. Convert an equation to C code. Do we need to be careful with data types?
4. Write a `scanf();` statement to read the user's input
5. Write a `printf();` statement to print the result
6. (Optional) Do you want the user to be *prompted*? This is when text appears in the console telling the user what to enter. This requires a `printf();` prior to the `scanf();`
7. (Optional) Does the output need to be in a human sentence? Write the `printf();` statement.
8. (Optional) Should you limit the precision of the printed result? Are 6 decimal places appropriate? Should it only be 1?

The following template can be used for all these exercises:

```c
#include <stdio.h>

int main() {
   // Declare the required variables

   // Get input from the user

   // Do the calculation

   // Print result to user

   return 0;
}
```

Task 1: Temperature Conversion

The formula for converting temperatures in Farenheit, F to Celsius, C, is:

$$C = \frac{5}{9}(F - 32) \tag{1}$$

Given the C template, write a program which performs this conversion. It should read the Farenheit temperature from `stdin` and print the result to `stdout`. The program should correctly handle any practical real values (eg: 19.38) input. Print the result to 2 decimal places.

Test your code with several values calculated with Google.

Task 2: Variance Calculation

From statistics, the *sample variance*, s^2, of N samples can be calculated as:

$$s^2 = \frac{\sum_{k=1}^{N}(X_k - mean(X))}{N - 1}. \tag{2}$$

Write a C program which reads **four** numbers and outputs their sample variance.

Given that $N = 4$ (and some algebra) Equation 2 can be expanded into two steps:

1. Calculate the sample mean: $\texttt{mean} = \frac{1}{4} \times (X_1 + X_2 + X_3 + X_4)$
2. Calculate the variance: $\texttt{var} = \frac{1}{3} \times ((X_1 + X_2 + X_3 + X_4 - 4 \times \texttt{mean})$

Your program will have to declare four `float` variables to store the four input values. To read four numbers in one instance of `scanf();` you can do this:

```
float x1, x2, x3, x4;
scanf("%f %f %f %f", &x1, &x2, &x3, &x4);
```

Test your program given that the variance of the set (1, 2, 3, 4) = 1.6667

NB: Be careful with the $\frac{1}{4}$ and $\frac{1}{3}$ constants in the equations above. They may experience integer division problems.

3 Flow Control Exercises

This first task presents you with all the required building blocks to implement a simple algorithm. All this program will do is read in 2 integers, perform a division, then print the result. A check is performed to make sure a division by zero does not occur.

Task 3

Write a program which implements the following pseudocode:

```
BEGIN
  Integer a
  Integer b
  Integer c
  Read an integer from standard input, store in a
  Read an integer from standard input, store in b
  IF b is zero
    PRINT "Division by zero error"
    RETURN // stop executing
  ENDIF
  c = a/b
  PRINT "a divided by b is" <the value of c>
END
```

Notes:

- Reading each integer can be done with: `scanf("%d", &a);` (with an appropriate substitution for the variable name). You may prompt the user by placing a `printf();` *without* newline character prior to `scanf();`, eg:

```
printf("Enter an integer: ");
scanf("%d", &a);
```

- Note that <the value of c> in the pseudocode is implying that a number should be printed there, ie:

```
printf("a divided by b is %d\n", c);
```

- The `RETURN` pseudocode can be implemented with:

```
return 0;
```

- The condition "b is zero" needs to be implemented with the == (*two* equals symbols) operator.
- You may start with the default code listing provided by OnlineGDB

Task 4: Square Root Evaluation

The square root of a number, $x = \sqrt{n}$, can be calculated with the iterative formula:

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{n}{x_k}\right) \tag{3}$$

Implement an algorithm which uses this formula to calculate square roots of real numbers. The value of n is to be read from the console with `scanf();` and the result printed with `printf();`.

The choice of x_0 is somewhat arbitrary, the algorithm should converge given any sane value (ie: not infinity), but will converge faster if x_0 is closer to the true value of $\sqrt{n}$. You may play with this value and observe any differences. Common choices would be setting it to n or zero.

The algorithm will require a loop. The exit condition could be one of several choices. In order of difficulty:

1. Exit after a fixed number of iterations (eg: 10)
2. Exit when $|x_k - x_{k-1}| < e$ where e is some pre-defined precision
3. Exit when either of the above conditions are met

To calculate precision you will need to explicitly save x_k and x_{k-1} in different variables.

Hint: You can place `printf();` statements inside the loop to help you debug. Calculate a particular value (eg: $\sqrt{2}$) by hand with a known x_0 so that you can compare your program's output with a known correct output.

Task 5: Quadratic Equation Analysis

The Friday Week 1 lecture introduced the following pseudocode for solving quadratic equations:

```
BEGIN
  INPUT: a, b, c
  D = b^2 - 4ac
  IF D < 0
     N = 0
  ELSEIF D = 0
    N = 1
    x1 = -b/(2a)
  ELSEIF D > 0
    N = 2
    x1 = (-b + sqrt(D))/(2a)
    x2 = (-b - sqrt(D))/(2a)
  ENDIF
  OUTPUT: N, x1, x2
END
```

Everyone: Write a C program which takes the `a, b,` and `c` values and outputs the number of real valued solutions which exist (ie: 0, 1, or 2).

Advanced: Using the square root algorithm coded previously, implement the full quadratic equation solution algorithm.

Task 6: Mandelbrot Set - Advanced

This problem is presented for students interested in mathematics. It may take several lab sessions to correctly implement this task. If you do not understand anything here, do not be troubled. The mathematics is beyond the scope of ENGG1003.

The *Mandelbrot set* is the set of all complex numbers, C, which can be iterated with the formula:

$$z_{n+1} = z^2 + C \tag{4}$$

and initial condition $z_0 = 0$ and have the value of z_n stay bounded (ie: not "blowing off to infinity") as n becomes "very large". Its calculation can be used to create mathematical artwork.

Task: Write a C program which tests a given point C for inclusion in the Mandelbrot set.

To do this you will be need to know that:

- z and C are complex numbers with real and imaginary parts
- We will use the notation:
  $z = x + iy$
  $C = x_0 + iy_0$
- Using rules of complex arithmetic it can be shown that:
  $z^2 + C = x^2 + i2xy - y^2 + x_0 + iy_0$
- In

4 C Summary

This section will be included in all future lab documents and lists a summary of C language features taught prior to the lab session. It will grow each week.

Not everything listed in this section is required to complete a particular lab.

4.1 Basic Structure

```c
#include <stdio.h>
int main() {
   // Your program goes here
   return 0;
}
```

4.2 Comments

```c
// This is a comment to end of line

/* this is a block comment
   which could span
   multiple
   lines
   */
```

4.3 Code Blocks

Any section of code encompassed by { . . . } is a *block*.

4.4 Operators

Operation	C Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Increment	++
Decrement	--
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	==
Not equal to	!=
Boolean AND	&&
Boolean OR	||
Boolean NOT	!

Table 1: Arithmetic operators in C

4.5 Operator Shorthand

Many arithmetic operators support the following shorthand syntax. The left and right columns present equivalent statements.

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

4.6 Data Types

Type	Bytes	Value Range
`char`	1	-128, +127
`unsigned char`	1	0, 255
`short`	2	-32768, 32767
`unsigned short`	2	0, 65535
`int`	4	$\approx \pm 2.1 \times 10^9$
`unsigned int`	4	0, 4294967296
`long`	8	$\approx \pm 9.2 \times 10^{18}$
`unsigned long`	8	$0, 1.8 \times 10^{19}$
`float`	4	1.2×10^{-38} to 3.4×10^{38}
`double`	8	2.3×10^{-308} to 1.7×10^{308}

4.7 Standard i/o

Read a single variable from `stdin` with `scanf();`
`scanf("format specifier", &variable);`

Write a single variable to `stdout` with `printf();`
`printf("format specifier", variable);`

You can use `printf();` *without* a newline (`\n`) to create an input prompt:

```c
printf("Enter a number: ");
scanf("%d", &variable);
```

This prints:

```
Enter a number:  _
```

where _ indicates the terminal prompt (ie: where typed characters will appear).

4.8 Format Specifiers

The following table is woefully incomplete. The compiler *may* generate warnings if `%d` is given something

other than `int` and `%f` is given something other than `float`. An attempt will be made to ensure these are sufficient.

Data Type	Format Specifier
Integers	`%d`
Floating point	`%f`
Float with n decimal places	`%.nf`

Table 2: Basic format specifiers

4.9 Type Casting

Placing the syntax `(type)` before a variable name performs a type cast (ie: data type conversion).

eg: convert `float` to an `int` prior to using its value. This forces a rounding-down to the nearest integer.

```
float a;
// ...
y = (int)a * z;
```

NB: This does **not** modify the original variable.

Data type "upgrades" are done automatically by the compiler but sometimes it is desired to downgrade or force esoteric behaviour. Adding it unnecessarily doesn't have any negative impact. Applications in ENGG1003 will be limited but it comes up regularly in embedded systems and nobody else explicitly teaches type casting. I have used it extensively in the low-level art of *bit banging*: manual manipulation of binary data. This is, unfortunately, beyond ENGG1003.

4.10 Flow control

Flow control allows selected blocks of code to execute multiple times or only under a specified condition.

4.10.1 `if()`

The `if()` statement executes a block of code only if the *condition* is true. The condition is an arithmetic statement which evaluates to either zero (false) or non-zero (true).

Syntax:

`if(condition) {/* other code */}`

Full example:

```
if(x > 10) {
   // Do stuff
}
```

Condition Examples:

- `if(x) // if(x is not zero)`
- `if(x+y) //if((x+y) is not zero)`
- `if(y >= 5)`
- `if(1) // Always executes`
- `if(0) // Never executes`
  - Can be used for debugging. Might be easier than a block comment /* */

NB: *NEVER* place a semicolon after an `if()`, that stops it from having any effect. The block after it will always execute. This bug can take days to find.

If there is only *one* statement after an `if()` the { } braces are optional:

```
if(x > 10)
   printf("x is greater than 10\n");
```

4.10.2 `if() ... else if()`

The C syntax for IF ... ELSE is:

```
if(condition) {
   // Do stuff
} else {
   // Do stuff
}
```

IF ... ELSEIF takes the form:

```
if(condition) {
   // Do stuff
} else if(condition) {
   // Do stuff
}
```

Multiple "layers" of `else if()` can be written. You don't have to stop at two.

4.10.3 `while()`

The `while()` flow control statement executes a block of code so long as a condition is true. The condition is checked before the block is executed and before every repeated execution.

The condition rules and examples are the same as for those listed under the `if()` statement.

Syntax:

`while(condition) {/* other code */}`

Example:

Evaluate the infinite sum:

$$\sum_{n=0}^{\infty} \frac{1}{n^2} \tag{5}$$

to a precision of 1×10^{-6}

```
float sum = 0.0;
int x = 0;
while(1/(x*x) > 1e-6) {
   sum = sum + 1.0/(x*x);
   x++
}
```

4.11 Library Functions

4.11.1 `rand()`

To generate a random number between `0` and `MAX`:

```
#include <stdlib.h> // For rand()
// ...
x = rand() % MAX;
```

For all work in this course you may assume that the above method works well enough.

For more crucial work (eg: cryptography, serious mathematics) this method is considered problematic. Very advanced discussion Here.