ENGG1003 - Lab 2 (And Beyond)

Brenton Schulz

1 Introduction

This set of programming exercises leaves you to perform a series of tasks on your own. This can be daunting at first, but your ability to remain resolute and resourceful in an atmosphere of extreme pessimism[1] will be a useful skill when studying Engineering[2]. (In truth, I hope you don't find it *that* hard and that you gain satisfaction and confidence from getting programs to work).

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 multiple lines of hand typed code to generate several errors the first time compilation is attempted. You will then find *more* errors when they are executed and found to produce the wrong output.

In time you will learn to interpret and deal with error messages quickly. Typically they will be a syntax error due to a missing parenthesis, semicolon, or double quote. Don't be afraid to ask for help from demonstrators and work with other students in the lab.

In short: all of your time spent programming will be fixing problems because as soon as all the (known) problems are fixed you stop programming and ship the software to your customer.

Generally speaking, the development process follows repeated application of the following steps:

1. Make a code change
2. Compile
3. Fix compile errors, repeat until code compiles
4. Test program with known correct input-output data
5. Repeat whole process until correct output is generated for multiple test inputs

I don't know how long these problems will take you to complete. With some adjustments, this may become a problem set for weeks 2 and 3.

[1] Yes, this is a reference to the computer game Portal.

[2] Talk to your demonstrators about how true this is. I write it only partially in jest. I've found it a necessary skill for *life*.

2 Getting Started Questions

Run though these questions quickly and ask your demonstrator if any appear confusing. These all cover basic fundamentals required to complete the lab.

1. Write a `printf();` statement which takes an integer variable `counter` and prints `The value of counter is 123` to the screen (with 123 replaced with the actual value).
2. A variable is required which stores integers with a maximum possible value of ten billion. What C data type is required?
3. A variable is required to store a real number with 10 significant figures. What C data type is required?
4. Write C code which implements the following pseudocode:

```
IF x is not equal to zero
  decrement x
ENDIF
```

   In C, decrement implies a reduction of magnitude by 1.

5. Write C code which implements the following pseudocode:

```
WHILE x is greater than 1
  y = y*x;
  x = x-1;
ENDWHILE
```

6. Write C code which implements the following pseudocode:

```
BEGIN
  integer result
  READ x from the console
  IF x is zero
    result = -1
  ELSEIF x is -1
    result = 0
  ELSE
    result = 1
  ENDIF
END
```

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

Task 3: Linear Interpolation

There are many instances in science and engineering when data needs to be *interpolated*. This is the process of predicting a quantity's value at a point in time (or space) where a measurement doesn't exist. The value at the unknown point needs to be interpolated (predicted) from data nearby.

In this task you will be using the *two-point formula* (you saw this in high school, right?) to interpolate between two data points. You can imaging these data points to be measurements of temperature / brightness / sound intensity / pressure / etc as a function of time.

The two-point formula states the equation of a line passing through two points (x_1, y_1) and (x_2, y_2) is:

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1. \tag{3}$$

The value of y at some given x can then be estimated. For the purposes of interpolation x will always be in between the given data points. ie, assuming $x_2 > x_1$:

$$x_2 > x > x_1. \tag{4}$$

Note that y is only an *estimate* of the true value. You will often see this notated as $\hat{y}$, especially in statistics.

Task: Write a C program which implements the following pseudocode:

```
BEGIN
   READ x1 and y1 from the console
   READ x2 and y2 from the console
   READ a point x from the console
   Estimate the value of y at x using linear interpolation
   PRINT y
END
```

To understand this problem you will find it useful to draw some sketches. Draw an x-y axis, dot two points, draw a line between them. If you pick two neat integer points (eg: 0,0 and 5,5) then the linear interpolation will be easy to see by eye.

Program implementation notes:

- Reading a pair of points can be done with:

```c
float x1, y1;
printf("Enter point 1: ");
scanf("%f %f", &x1, &y1);
```

  you would then type two numbers into the console, separated by a space, and press enter.
- Keep all variables as `float` or `double`. It doesn't *really* matter which, just be consistent.

4 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 4

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 5: Rolling Dice

Using the `rand();` function, write a C program which estimates the probability that, when two dice are thrown, at least of the dice rolled a 3. This can be done with the following algorithm:

```
BEGIN
  counter = 0;
  aThreeHappened = 0
  WHILE counter < 1000
    a = random number between 1 and 6
    b = random number between 1 and 6
    IF (a == 3) OR (b == 3)
      aThreeHappened = aThreeHappened + 1
    END
    counter = counter + 1
  ENDWHILE
  PRINT aThreehappened / counter
END
```

The theoretical value is $\frac{11}{36} = 0.3055$

Generation of a random number between 1 and 6 can be done with:

```
#include <stdlib.h> // At top of code listing
...
a = (rand() % 6) + 1;
```

Task 6: Calculating Pi

Implement the π calculation algorithm talked about during lectures. The pseudocode is:

```
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
```

Remember[a] that a random number can be created with the `rand();` library function. To use it be sure to add `#include <stdlib.h>` to the top of your source code.

A number between 0 and 1 can then be generated with:

```
1 float x;
2 x = (float)rand()/RAND_MAX;
```

NB: If running this on OnlineGDB use a "small" value for the maximum loop count. Try one million, if this takes longer than around 10 seconds to execute use a smaller value. You are occupying OnlineGDB's shared CPU resources when running this program.

[a] Hopefully this is "remember" and not "I didn't get time to cover this yet".

Task 7: Calculating Square Roots

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

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{n}{x_k}\right) \tag{5}$$

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 of your choosing
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 the series of x_n's for a particular value (eg: $\sqrt{2} = 1.4142$) by hand (with the same value for x_0) so that you can compare your program's output with an output you have confidence in.

Task 8: 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.

5 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, do not feel the need to attempt this task. The mathematics is beyond the scope of ENGG1003.

Demonstrators are not to help you solve this problem. It is intended for students who have advanced beyond expectations, doing it alone is part of the challenge.

A complex number, C, exists within the *Mandelbrot set* if, when iterated with the formula:

$$z_{n+1} = z_n^2 + C, \tag{6}$$

with initial condition $z_0 = 0$, the value of $|z_n|$ stays bounded (ie: doesn't "blow off to infinity") as n becomes "very large". It's calculation can be used to create mathematical artwork. Hanging in my office is a piece of art known as a Buddahbrot. Calculation took several days on an R9 270X gaming graphics card. Come check it out some time. I have an unhealthy love of this branch of mathematics.

Purely real examples:

- $C = 0.1$ is *inside* the Mandelbrot set, as:

$$z_1 = 0 + 0.1 \tag{7}$$
$$z_2 = 0.1^2 + 0.1 = 0.11 \tag{8}$$
$$z_3 = 0.11^2 + 0.1 = 0.1121 \tag{9}$$
$$\ldots \tag{10}$$
$$z_{14} = 0.112701665^2 + 0.1 = 0.112701665 \tag{11}$$
$$Z_{15} = 0.112701665^2 + 0.1 = 0.112701665 \tag{12}$$

  ...and it just sits at 0.112701665 forever.

- $C = 1$ is *outside* the Mandelbrot set, as:

$$z_1 = 0 + 1 \tag{13}$$
$$z_2 = 1^2 + 1 = 2 \tag{14}$$
$$z_3 = 2^2 + 1 = 5 \tag{15}$$
$$z_4 = 5^2 + 1 = 26 \tag{16}$$
$$z_5 = 26^2 + 1 = 677 \tag{17}$$
$$\ldots \tag{18}$$
$$z_{11} = (3.79 \times 10^{90})^2 + 1 = 1.4 \times 10^{181} \tag{19}$$

  ...and it just keeps getting bigger. Fast.

Task 9

Write a C program which tests a *single* point, C, for inclusion in the Mandelbrot set. In the future your code can be expanded to draw an image.

To do this you will be need to know that:

- z and C are complex numbers with real and imaginary parts
- Introducing some notation:

$$z_n = x_n + iy_n \tag{20}$$
$$C = x_0 + iy_0 \tag{21}$$

Where x denotes a real part and y the imaginary.

- Using rules of complex arithmetic it can be shown that:

$$z_{n+1} = z_n^2 + C \tag{22}$$
$$= x_n^2 + i2x_n y_n - y_n^2 + x_0 + iy_0 \tag{23}$$

- To evaluate this on a computer which only deals with *real* numbers it has to be split into real and imaginary parts:

$$Re(z_{n+1}) = x_{n+1} = x_n^2 - y_n^2 + x_0 \tag{24}$$
$$Im(z_{n+1}) = y_{n+1} = 2x_n y_n + y_0 \tag{25}$$

- A complex number's norm (or absolute value, informally its "size") can be calculated as:

$$|z| = \sqrt{x^2 + y^2} \tag{26}$$

and is needed to test if z_n is getting "too big". For this problem, "too big" is typically defined as "outside a circle of radius 2". To avoid having to calculate the square root (which can be slow) you can take the so-called *escape condition* as:

$$x^2 + y^2 > 4 \tag{27}$$

In code, this will go inside a loop's exit condition. Either as a "NOT (x*x+y*y > 4)" or, more simply, "x*x + y*y < 4" as loop conditions need to remain TRUE for the loop to continue.

- The loop which implements the iterative equation need to keep track of the iteration count but does *not* need to record a full history of z_n's. You only need the "next" and "current' values of x_n and y_n. At some point an assignment causes the "next" to become the "current".
- An *iteration limit* needs to be set. A point, C, is considered "inside" the set if the iteration limit is hit without $|z_n|$ exceeding 2.
- If $|z_n| > 2$ before the iteration limit is hit the point, C, is inside the Mandelbrot set
- Full, somewhat optimised, pseudocode for this problem can be found on Wikipedia. Independent research may be required to implement this problem.
- For testing purposes here are some test points with an iteration limit of 1000:
  - $C = 0 + i0$ takes 1000 iterations
  - $C = 0.5 + i0$ takes 4 iterations
  - $C = 0.4 + i0.1$ takes 7 iterations
  - $C = 0.38 + i0.1$ takes 35 iterations
  - $C = -0.38 + i0.1$ takes 1000 iterations
  - If all these test points take an iteration count which is offset by 1 from my solution your code is probably fine. This kind of error happens all the time due to < Vs <= and the like.

6 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.

6.1 Basic Structure

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

6.2 Comments

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

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

6.3 Code Blocks

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

6.4 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

6.5 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;`

6.6 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}

6.7 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).

6.8 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

6.9 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.

6.10 Flow control

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

6.10.1 `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");
```

6.10.2 `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.

6.10.3 `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:

$$\sum_{n=0}^{\infty} \frac{1}{n^2} \tag{28}$$

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++
}
```

6.11 Library Functions

6.11.1 `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.