ENGG1003 - Lab 2 (And Beyond)

Brenton Schulz

1 Feedback

To help me write better labs please fill out this survey when you leave your **Week 2** lab:

Click here to complete the survey before you leave.

I just want to know how many problems the "average" student can complete in 3 hours. It will also tell me which problems are more engaging than others (does everyone *hate* probability? Do you love it? I have no idea.)

You likely won't be able to complete *all* the problems, that is intentional. If 60% of you complete the lab it means I don't know how many questions 60% can do in 3hrs.

2 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 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* (2007).

[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*.

3 Pseudocode to C Conversion Examples

Pseudocode is any rough working out which you write (or are given) which describes a computer algorithm in the most human-readable form practical. It will *not* run when typed into a C compiler. It only exists to document algorithms and assist in helping you convert a "real world" problem into computer code.

The table below covers the flow control structures seen so far. The first entry is what I consider to be the *absolute minimum* required for all code in this course. The `main()` function is required, it is where program execution begins.

Pseudocode	C Code

`BEGIN`
`  // do things`
`END`
	`#include <stdio.h>`
	`int main() {`
	`  // do things`
	`  return 0;`
	`}`

`IF condition`
`  // do things`
`ENDIF`
	`if(condition) {`
	`  // do things`
	`}`

`IF condition`
`  // do things`
`ELSE`
`  // do things`
`ENDIF`
	`if(condition) {`
	`  // do things`
	`} else {`
	`  // do things`
	`}`

`IF condition`
`  // do things`
`ELSE IF condition`
`  // do things`
`ELSE`
`  // do things`
`ENDIF`
	`if(condition) {`
	`  // do things`
	`} else if(condition) {`
	`  // do things`
	`} else {`
	`  // do things`
	`}`

`WHILE condition`
`  // do things`
`ENDWHILE`
	`while(condition) {`
	`  // do things`
	`}`

4 Data Type Subset and Format Specifiers

Although C contains many data types you only need to consider the following subset when completing this lab:

Type	Value Range	Format Specifier
`int`	-2 147 483 647 to +2 147 483 647	`%d`
`float`	1.2×10^{-38} to 3.4×10^{38}	`%f`

The *Value Range* indicates the maximum and minimum values which can be stored in each type. For the `float` data type both positive and negative numbers can be stored. If an arithmetic statement causes an `int` to exceed the value range the result will "wrap around"; a behaviour known as *overflow* or *underflow*. If you type in a number which is too big and read it with `scanf("%d", &x);` the number stored will be different to that which you typed in.

The *format specifier* is the text which goes inside `scanf();` and `printf();` when reading or printing numbers, respectively. It tells the compiler how ASCII text should be converted into a number (and back again).

Eg, to read and print an `int`:

```
int value;
scanf("%d", &value);
printf("You entered: %d\n", value);
```

Change the two `%d`'s to `%f` to read and print a `float`. Always consider what data type is being read or written when typing format specifiers.

If you write the wrong format specifier it will result in garbage. For example, the following code uses `%f` inside `scanf();` when it should have been `%d`:

```
#include <stdio.h>

int main() {
   int x;
   printf("Enter an integer: ");
   scanf("%f", &x); // %f is incorrect!
   printf("You entered: %d\n", x);
   return 0;
}
```

It exhibits the following behaviour on when executed:

```
Enter an integer: 1
You entered: 1065353216
```

The compiler, gcc, produces a *warning* when the code is compiled:

```
format.c: In function   m a i n   :
format.c:6:10: warning: format    % f    expects argument of type   f l o a t  *
      , but argument 2 has type   i n t  *    [-Wformat=]
  scanf("%f", &x);
         ~^    ~~
         %d
```

but, unfortunately, OnlineGDB does not show it to you. It only displays warnings if errors also occurred. We will be moving to a more advanced development environment which *does* print warnings as soon as IT has made it work on the lab machines.

5 Usage of `printf();` and `scanf();`

The basic usage of `printf();` and `scanf();` when working with single integers is:

```
scanf("%d", &x); // Read an integer, store it in x
printf("%d \n", x); // Print the value of x followed by a new line
```

If using `float` or `double` "fractional" numbers the `%d` gets replaced with `%f`.

They can, however, be used to read or print arbitrarily many variables. To do this you:

1. Use multiple format specifiers (`%d` and `%f`) inside the *format string*
2. List multiple variables, separated by commas, after the format string

The format string is everything between the pair of double quotes:

```
printf("This is the format string.\n")
```

For example, to read three integers, `x, y,` and `z` from the console:

```
int x,y,z; // This declares three int variables on one line
scanf("%d %d %d", &x, &y, &z);
```

The integers you type need to be separated by spaces (note that spaces are present inside the format string) and <enter> typed to end the line. The `scanf();` function won't process any data until an end of line is read.

Note that the order is *crucial*; the numbers you type will go into `x, y` and `z` in the the same order that the variable names are listed in `scanf();`.

You don't *need* to separate the input with spaces. You can, for example, read an hh:mm time with:

```
scanf("%d:%d", &hours, &minutes);
```

and `scanf();` will match the integer-colon-integer pattern with the input and extract hours and minutes.

`scanf();` even supports incredibly advanced "pattern matching" but it is *well* beyond the scope of ENGG1003.

Note that the numbers don't all need to be integers. You can mix integers and, say, `doubles`:

```
int x,y; // This declares three int variables on one line
double z;
scanf("%d %d %f", &x, &y, &z);
```

Likewise, the `printf();` function can print multiple numbers in arbitrary locations. You can write a mixture of `%d` and `%f` anywhere inside the format string and values will be pulled from the list of variables in order.

For example:

```
int a = 1, b = 2;
float c = 1.23;
double d = 8.2;
printf("%d is an integer, %d is an integer, have some more numbers: %f %f\n",
    a, b, c, d);
```

will print:

```
1 is an integer, 2 is an integer, have some more numbers:  1.23 8.2
```

Lastly, if you get this compiler warning:

```
main.c:6:13:  warning:  format %d expects argument of type int *, but argument
3 has type float * [-Wformat=] scanf("%d:%d", &h, &m);
```

it means that the data type and format specifier don't match. You have made a mistake, fix it.

6 Getting Started Questions

These problems are *not* meant to be executed in a C compiler. Read through them and write your answers on paper (or Word etc). Solutions can be found on the last page of this document.

1. What are the values of `x, y, v,` and `z` after execution of each line? On which lines are each variable undeclared (ie: their value is known)? You don't need to run this code, only read and think about it.

```c
int x = 2;
int y, v, r;
float z;
x = x+5;
y = 2*x + 3;
v = y/x; // NB: integer division
r = y%x;
z = (float)y/x; // Forced to floating point division
```

2. What is printed to the console if the user enters `13<enter>`? You don't need to run this code, only read and think about it.

```c
float k;
printf("Enter a number: ")
scanf("%f", &k);
k = 1.43*k + 84;
printf("%f\n", k);
printf("%d\n", (int)k); // Integer truncation
```

3. Write a `printf();` statement which takes an integer variable `temp` and prints:

   `The temperature reading of <temp> is dangerous`

   to the screen (with `<temp>` replaced with the actual value, ie: write the %d format specifier in that spot).

4. A variable is required which stores integers with a maximum possible value of ten billion. What C data type is required? Why can't you use an `int`? See Section 11.6 for a list of C data types.

5. A variable is required to store a real number with 10 significant figures. What C data type is required? See Section 11.6 for a list of C data types.

6. Write a C "code snippet" which implements the following pseudocode (decrement implies a subtraction by 1):

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

   A "code snippet" is just the code required to implement the pseudocode, you don't need to write a full program. In this case, you are only converting the pseudocode `IF..ENDIF` into something that looks like:

```c
if(condition) {
   /* stuff */
}
```

7. Write a C code snippet which implements the following pseudocode:

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

8. Write a C code snippet which implements the following pseudocode:

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

7 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? Use the `%.nf` format specifier to limit floating point precision to `n` decimal places

The following template can be used for all these exercises:

```c
#include <stdio.h>

int main() {
   // Declare the required variables
   // eg: int x; float y; etc

   // Get input from the user
   // eg:   printf("Enter a number: ");
   //       scanf("%f", &y);

   // Do the calculation
   // x = ....whatever

   // Print result to user
   // eg: Printf("%d times %d is %d\n", x, y, z);

   return 0;
}
```

7.1 Example

Write a C program which calculates a miles to kilometres conversion with the following formula:

$$\texttt{km} = 1.60934 \times \texttt{miles}. \tag{1}$$

Going through the steps above:

1. We need a variable `miles` and another `km`
2. We are multiplying by a fractional number so they need to be of type `float` (or `double`)
3. In C, the equation is implemented with: `km = 1.60934 * miles;`
4. To read a `float` from the user we need the `%f` format specifier in `scanf();`. To store the read value in `miles`, we can use: `scanf("%f", &miles);`
5. To print the result of type `float` we need the `%f` format specifier, so we can print it with: `printf("%f\n", km);`

To make the above "code snippets" *actually run* they need to be written inside `main() { }` and we need to write `#include <stdio.h>` at the top. Putting it all together, the following code will implement the problem:

```c
#include <stdio.h>
int main() {
   float km;
   float miles;

   scanf("%f", &miles);
   km = 1.60934 * miles;
   printf("%f\n", km);

   return 0;
}
```

All the Tasks in this section can be implemented by modifying the above template.

7.2 Tasks

Task 1: Temperature Conversion

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

$$C = \frac{5}{9}(F - 32). \tag{2}$$

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) as 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, $X_1, X_2, ..., X_N$, can be calculated as:

$$s^2 = \frac{1}{N-1}\sum_{k=1}^{N}(X_k - mean(X))^2. \tag{3}$$

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

Given that $N = 4$ (and some algebra) Equation 3 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: $\texttt{var} = \frac{1}{3} \times ((X_1 - \texttt{mean})^2 + (X_2 - \texttt{mean})^2 + (X_3 - \texttt{mean})^2 + (X_4 - \texttt{mean})^2)$

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 1: Be careful with the $\frac{1}{4}$ and $\frac{1}{3}$ constants in the equations above. They may experience integer division problems. If in doubt, convert them to floating point literals.

Observation: Did you find programming this task is really tedious? What if you had 100 samples? Or a million? In later weeks we will learn about *arrays*. These are a C language feature which greatly simplify the code when you need to keep track of a lot of data.

Task 3: Linear Interpolation

There are many instances in science and engineering when data needs to be *interpolated*. Informally, this is the process of predicting a quantity's value somewhere between two (or more) data points.

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 imagine these data points to be measurements of temperature / brightness / sound intensity / pressure / etc as a function of time.

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

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1. \tag{4}$$

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 $x_2 > x_1$:

$$x_2 > x > x_1. \tag{5}$$

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

The process of estimating values outside the $[x_1, x_2]$ interval is known as *extrapolation*[a] and is beyond the scope of this task.

Given the above, 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 and 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. Perform several tests with known-correct sets of values to confirm that your code is correct.

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.

[a] https://xkcd.com/605/

8 Flow Control Exercises

A template is not explicitly provided for these tasks. You can, however, use the "hello world" template provided by OnlineGDB or the code written for previous tasks as a starting point.

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 it in a
  Read an integer from standard input, store it in b
  IF b is zero
    PRINT "Division by zero error"
  ELSE
    c = a/b
    PRINT "a divided by b is" <the value of c>
  ENDIF
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 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

This is where most (approx 77%) students have gotten up to. Don't feel bad if you're out of time by now.

Task 5: Calculating Square Roots

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

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{n}{x_k}\right) \tag{6}$$

Task: Write a C program which implement 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* (x_n will move towards the result, not away) given any sane value (eg: not infinity), but will converge `faster` (ie: in fewer iterations) if x_0 is closer to the true value of $\sqrt{n}$. You may play with this value and observe how accurate the result is after a fixed number of iterations (say, 5). Common choices for x_0 would be $x_0 = n$ or $x_0 = 1$.

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 predefined precision of your choosing
3. Exit when either of the above conditions are met

To calculate precision you will need to explicitly save

8.1 Else-If

The C syntax for an `IF .. ELSEIF .. ENDIF` flow control structure is:

```
if( condition ) {
    // Statements
} else if( condition ) {
    // Statements
}
```

There are many variants, for more examples see Section 11.10.1.

Please be aware of a common mistake: a semicolon should *not* be placed after an `if()` statement. Doing so tells the compiler "if this condition is true, do nothing". C will always do exactly what you tell it to, even if you tell it to do something that appears to be useless.

Task 6: 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).

Everyone: Introduce a "temporary" variable (ie: short lived, one we stop caring about soon after it is used) which explicitly removes the requirement to calculate `sqrt(D)` twice.

Advanced: Using the square root algorithm in Equation 6, implement the full quadratic equation

9 Random Numbers

The following tasks involve generating and using random numbers. The standard C library (glibc, in Linux and MinGW, provided by the GNU project) contains a function called `rand();` which creates a random number between 0 and a constant called `RAND_MAX`. In glibc `RAND_MAX` is 2147483647, or the maximum value an `int` can store. It may vary between implementations (older systems may use 32767, for example).

Because `rand();` is part of a "library" and not built into the C language you need to include a *header file* which describes to the compiler what `rand();` is before it is used. To do this, type the following line at the top of your source listing (anywhere above `main()`):

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

The `rand();` function can then be included in your code. You can write it in all the same places that you would write a variable or literal. All of the following statements are valid:

```
x = rand();
y = 10 + rand() % 5;
printf("A random number is: %d\n",
  rand() );
```

Generally you aren't after a random number between 0 and `RAND_MAX`. To limit the rand of integers generated

we can use the modulus (`%`) operator. For the purposes of this course the following methods can be used to generate:

- A number between 0 and `MAX`:

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

- A number between 1 and `MAX`:

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

- A floating point number between 0 and `MAX`:

```
x = (float)rand()/RAND_MAX * MAX;
```

  NB: `rand();` becomes an `int` so an explicit cast to `float` (or `double`) is required to avoid integer division errors.

IMPORTANT: The `rand();` function is what's known as "pseudo-random". Repeated use of `rand();` will generate a sequence of random numbers but the *same sequence* will be generated every time the program is run.

To change the sequence you can use the `srand();` function (known as "seed-rand") with a manually chosen random number:

```
srand(233534);
```

Every different number given to `srand();` will cause a different random sequence to be generated by `rand();`.

You only need to include `srand();` **once**, somewhere before `rand();` is used.

In the future, we will learn how to "seed" random number generation with the current time. That way the sequence will automatically change each time the program is run.

Example:

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int x;
    srand(2323423);
    x = rand();
    printf("A random number is: %d\n",
    x);
}
```

Task 7: Rolling Dice

Using the `rand();` function, write a C program which estimates the probability that, when two dice are thrown, at least one 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 $\frac{11}{36} = 0.3055$

Task 8: Arithmetic Tutor

Write a C program which can be used to teach basic arithmetic to primary school children.

Your program should follow the following algorithm:

```
BEGIN
  integer max
  READ max from the user
  integer a = a random number between 1 and
  integer b = a random number between 1 and
  integer c = a + b
  PRINT What is a + b? // a and b the numer
  READ an integer d
  IF c == d
    PRINT "Correct!"
  ELSE
    PRINT "Wrong :(. The correct answer is
  ENDIF
END
```

Extension 1: Modify the algorithm so that it keeps generating problems until the user gets one wrong. (ie: Chuck a `while()` around the appropriate lines, create a variable which records if the answer was wrong or correct, and test that variable in the `while()` condition).

Extension 2: Add a variable which keeps score and prints the number of correctly answered

Task 9: 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
                y = random number between 0
                countTotal = countTotal + 1
                IF x*x + y*y < 1
                        countInside = count
                ENDIF
        ENDWHILE
        pi = 4*countInside/countTotal
        PRINT pi
END
```

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.

10 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:

$$z_{n+1} = z_n^2 + C, \tag{7}$$

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:

$$z_1 = 0 + 0.1 \tag{8}$$
$$z_2 = 0.1^2 + 0.1 = 0.11 \tag{9}$$
$$z_3 = 0.11^2 + 0.1 = 0.1121 \tag{10}$$
$$\ldots \tag{11}$$
$$z_{14} = 0.112701665^2 + 0.1 = 0.112701665 \tag{12}$$
$$Z_{15} = 0.112701665^2 + 0.1 = 0.112701665 \tag{13}$$

...and it just sits at 0.112701665 forever.

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

$$z_1 = 0 + 1 \tag{14}$$
$$z_2 = 1^2 + 1 = 2 \tag{15}$$
$$z_3 = 2^2 + 1 = 5 \tag{16}$$
$$z_4 = 5^2 + 1 = 26 \tag{17}$$
$$z_5 = 26^2 + 1 = 677 \tag{18}$$
$$\ldots \tag{19}$$
$$z_{11} = (3.79 \times 10^{90})^2 + 1 = 1.4 \times 10^{181} \tag{20}$$

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

Task 10

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

To do this you will be need to know that:

- z and C are complex numbers with real and imaginary

parts

- Introducing some notation:

$$z_n = x_n + iy_n \tag{21}$$
$$C = x_0 + iy_0 \tag{22}$$

Where x denotes a real part and y the imaginary.

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

$$z_{n+1} = z_n^2 + C \tag{23}$$
$$= x_n^2 + i2x_ny_n - y_n^2 + x_0 + iy_0 \tag{24}$$

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

$$Re(z_{n+1}) = x_{n+1} = x_n^2 - y_n^2 + x_0 \tag{25}$$
$$Im(z_{n+1}) = y_{n+1} = 2x_ny_n + y_0 \tag{26}$$

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

$$|z| = \sqrt{x^2 + y^2} \tag{27}$$

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:

$$x^2 + y^2 > 4 \tag{28}$$

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 needs 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 5 iterations
  - $C = 0.4 + i0.1$ takes 8 iterations
  - $C = 0.38 + i0.1$ takes 36 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.

11 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.

11.1 Basic Structure

This is the absolute minimum amount of code you need to make a C program compile, run, and interact with the user via a console:

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

11.2 Comments

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

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

11.3 Code Blocks

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

11.4 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

11.5 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;`

11.6 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}

Type	Precision
`float`	6 sig. figs.
`double`	15 sig. figs

11.7 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:

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

This prints:

`Enter a number:` _

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

11.8 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

11.9 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.

to either zero (false) or non-zero (true).

11.10 Flow control

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

`if()` Syntax:

11.10.1 `if()`

The `if()` statement executes a block of code only if the *condition* is true. The condition is an arithmetic statement which evaluates

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

Full `if()` 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");
```

11.10.2 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.

NB: *NEVER* place a semicolon directly after the `else if()`. Semicolons only go after the statements inside the `if()` block (ie: between the curly braces {}).

11.10.3 `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:

$$\sum_{n=0}^{\infty} \frac{1}{n^2} \tag{29}$$

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++
}
```

NB: *NEVER* place a semicolon directly after a `while()` line. Semicolons only go after the statements inside the loop.

11.10.4 `for(;;)`

As of week 2 this has not yet been covered in lectures.

The `for(;;)` loop syntax is:

```
for( initial ;
  condition ;
  increment ) {
    // Do stuff
}
```

The three sub-parts have the following behaviour:

- **Initial:** Code which is executed *once*, before the loop is entered
- **Condition:** A condition which is tested *before* every loop iteration
- **Increment:** Code which is executed *after* every iteration

`for(;;)` Example:

```
int x;
for( x = 0 ; x <
  10 ; x++ ) {
    printf("%d ", x
  );
}
```

will print:

```
0 1 2 3 4 5 6 7 8
9
```

The `for(;;)` loop example doesn't print `10` because the condition is "strictly less than 10". When x is incremented to 10 the condition fails and the loop exits. It prints 0 because the increment is only applied *after* the loop

has run once.

11.11 Library Functions

11.11.1 `rand()`

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

```
#include <stdlib.h> // For rand()
// ...
x = rand() % (MAX + 1);
```

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.

11.12 Glossary of Terms

I'll sort these alphabetically later, just a brain dump for now. Sorry.

- **Compiler:** The software package which converts *source code* into a *binary*.
- **Source Code:** The text which you type into a programming environment (eg: OnlineGDB) which is sent to the *compiler*.
- **Binary:** A program data file which can be executed on a computer.
- **Variable:** A "thing" which remembers a number within your program. In C they have a *type*, a name, (optionally) an initial value, and (for future reference) a *memory address*. Variables change when they are on the left side of an *assignment*.
- **Pseudocode:** Any

hand-written or typed notes which document the behaviour of a computer program. Pseudocode is for humans to read, not computers.

- **Flow Control:** Any algorithmic statement which breaks the "top-to-bottom, line-by-line" execution pattern of a computer program.

- **Statement:** A line of C code which performs a task and ends with a semicolon. Arithmetic lines, `printf();`, `scanf();`, and `rand();` are all examples of statements.
- **Assignment:** The process of changing a variable's value as your program executes. In C, this is typically performed with the = symbol. Eg: `x = y + 5` calculates the value of "`y + 5`" then allocates the result to the variable `x`.
- **Block:** A section of code "grouped together" by curly braces { ... }. Typically applies to flow control, where a single, say, `if()` controls a block of code listed inside { and }.
- **Literal:** Any numeri-

cal constant written in your code. Eg: In `x < 2.0` the "`2.0`" is a literal. Unless otherwise stated, integer literals are treated as `int` data types (ie: they inherit the `int`'s value range limit) and real valued literals are treated as `doubles`.

12 Getting Started Question Solutions

1. What are the values of `x`, `y`, `v`, and `z` execution of each line? On which lines are each variable undeclared (ie: their value is known)?

```
int x = 2;  // x = 2, all others unknown
int y, v, r;// x = 2, y v r still unknown
float z;    // x = 2, y v r z still unknown
x = x+5;    // x = 7
y = 2*x + 3;// y = 17
v = y/x;    // NB: integer division. v = 2
r = y%x;    // r = remainder of (17 / 7) = 3
z = (float)y/x; // Forced to floating point division, z = 2.4286
```

2. What is printed to the console if the user enters `13<enter>`?

```
float k;
printf("Enter a number: ")
scanf("%f", &k);
k = 1.43*k + 84;      //k = 1.43*13+84 = 102.59
printf("%f\n", k);    // 102.590000
```

```
  (6 decimal places by default)
printf("%d\n", (int)k); // Integer
  truncation, prints: "102"
```

3. Write a `printf();` statement which takes an integer variable `temp` and prints:

   `The temperature reading of 123 is dangerous`

   to the screen (with 123 replaced with the actual value).

```
// Question didn't specify if a new
   line is needed. Consider the
  trailing \n optional
printf("The temperature reading of
  %d is dangerous", temp);
```

4. A variable is required which stores integers with a maximum possible value of ten billion. What C data type is required?

   `long`. An `int` only stores up to approx 2.1 billion so it can't be used

5. A variable is required to store a real number with 10 significant figures. What C data type is required?

   `double`. A `float` only has about 6 significant figures of precision

6. Write a C code snippet which implements the following pseudocode (decrement implies a subtraction by 1):

   `IF x is not equal to zero`

```
  decrement x
ENDIF
```

```
if(x != 0)
{
   x--; // --x will also work. As
  would x = x - 1 or x -= 1
}

```

7. Write a C code snippet which implements the following pseudocode:

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

```
while(x > 1) {
   y = y * x;
   x = x - 1;
} // end while()
```

8. Write a C code snippet which implements the following pseudocode:

```
BEGIN
  integer result
  integer x
  READ x from the console
  IF x is zero
```

```
    result = -1
  ELSEIF x is -1
    result = 0
  ELSE
    result = 1
  ENDIF
END
```

```c
int result;
int x
scanf("%d", &x);
if(x == 0) {
   result = -1;
} else if(x == -1) {
   result = 0;
} else {
   result = 1;
}
```