ENGG1003 - Lab Week 5

Brenton Schulz

1 Introduction

These lab tasks include some reasonably non-trivial mathematics. I have received feedback from both demonstrators and students that the mathematics so far has, for some, been difficult to follow. Unfortunately, I can't do much about this. Engineering is *very* mathematics heavy. You can't get away from it. The mathematics in 2nd year only gets *much* more advanced.

If you find the mathematics here difficult to understand please take some time to learn it in your own time. You can ask the ENGG1003 PASS leader for help and there are PASS sessions for MATH1110 (or MATH1002) you can attend. You can also ask ENGG1003 demonstrators or your MATH* tutors and lecturers for help.

The 2019 PASS timetable can be found here.

I (Brenton) speak from bitter personal experience here. If you do not finish 1st year with solid foundational knowledge in mathematics you will have immense struggles continuing. In the mid-2000s I failed *over half* of my 2nd year subjects the first time through engineering for this reason.

If mathematical understanding is getting in the way of learning programming I *strongly* recommend that you attempt the *Sieve of Eratosthenes* task as a priority. It only requires basic arithmetic but the algorithm is non-trivial to implement.

2 New Content

2.1 Functions

A C *function* is a block of code which can be *called* from anywhere in your program. A C function is defined by:

- A unique name
  - The name must be unique among all functions within your program, the libraries which are `#include`'ed, and variables within the scope of where the function is called
- Zero or more *arguments*. Each argument is separated by a comma, has a fixed data type, and name unique to that function's scope
  - eg: (float x, int num, char a)
  - Functions which don't take arguments must list `void` in place of arguments in their prototype and definition
- A *return value* data type
  - Functions with no return value have a return data type of `void`

A *function prototype* summarises the above information with the syntax:

```
return_datatype function_name(arguments);
```

for example, the `sin()` function in the mathematics library returns a `double` and accepts a single `double` as an argument. It therefore has the following prototype:

```
double sin(double x);
```

A function *definition* is a repeat of the function prototype followed by the code which gets executed when the function is called.

A function *body* is just the code which gets executed. It is a subset of the function definition.

Example: the following code listing includes a function called `printResult` which takes a single `float` argument and prints a human-readable message including that number:

```
#include <stdio.h>

// This line is the function prototype
// "float x" indicates that it takes a single argument of type float.
// The return type is void because it does not return a value
void printResult(float x);

int main() {
   float a = 2.0;

   printResult(a); // Test the function by calling it
}

// The lines below are the function definition
void printResult(float x) {
   // This printf() statement is the only code in the function body
   // The argument "x" is a pre-initialised variable within this scope
   printf("The final calculation result was: %f\n", x);
}
```

2.2 Static Variables

A static variable is one which retains its value between function calls. Normally, all variables declared in functions (including arguments) are lost when the function returns. Any variable declared in a function with the `static` keyword is kept safe and retains its value between function calls.

For example, the following code prints the numbers 1 to 5 because, within the definition of `counter()`, `x` is declared with the `static` keyword:

```c
#include <stdio.h>

// Takes no arguments, so void argument list
// Returns an int equal to the number of times it has been called
int counter(void);

int main() {
   // Call 5 times just to test its operation
   printf("%d\n", counter() );
   printf("%d\n", counter() );
   printf("%d\n", counter() );
   printf("%d\n", counter() );
   printf("%d\n", counter() );
}

// Takes no arguments, so void argument list
// Returns an int equal to the number of times it has been called
int counter(void) {
   // Declaration of x initialises it to zero on first call only
   static int x = 0;
   // Increment x. Initialisation is only done once
   x = x + 1;
   return x;
}
```

2.3 Arrays

An *array* is a collection of variables, all of the same type, arranged in a long list and accessed with an *array index*. In C, an array is declared with a fixed length, data type, and name with the following syntax:

```
data_type name[size];
```

The data type can be any of the standard C types seen so far (`char, int, unsigned long, float,` etc). The name must be unique among all variables within its scope. Arrays may be initialised when declared with the following syntax:

```
data_type array_name[] = {1,2,3,4,5,6};
```

If the above syntax is used the array size (inside the []'s) is calculated from the initialisation and doesn't need to be explicitly listed.

The size must be an integer but does not need to be known at compile time. For example, the following code snippet declares an array with a size given by the user at runtime:

```
1 int length;
2 printf("Enter an array length: ");
3 scanf("%d", &length);
4
5 float numbers[length];
```

A single variable stored inside an array is known as an array *element*. Each element is accessed with a unique *index*. The index is an integer ranging from zero to length-1. Array elements are accessed with the following syntax:

```
arrayName[index]
```

Arrays can be used anywhere that other variables can and their index can be an integer variable or literal.

Examples:

```
x = array[2]; // Simple arithmetic example
y = 2.0*x[12] + y[0]; // Non-trivial arithmetic
n[i] = sin(x); // As lvalues (NB: array can't be declared "const")
printf("%f\n", nums[12]); // As an argument to a function
```

3 Tasks

Task 1: (Simple Function): Arithmetic

Write a C function, `arithmetic()` which takes two `float` arguments, `a` and `b`, and returns their sum.

Sub-tasks:

1. Write the function prototype.
2. Write the function definition.
3. Place the prototype and definition inside the test code below. It includes test variables `x` and `y`, declared within `main()`. Compile, run, and compare your program's output with a manual calculation of `x + y`:

```c
#include <stdio.h>

// Place prototype here

int main() {
   float x = 1, y = 5;
   float result;

   result = arithmetic(x, y);
   printf("%f plus %f is %f\n", x, y, result);
}

// Place definition here

```

Task 2: (Switch Statements): Arithmetic Function

Extend your function from the previous task to accept a third integer argument, `operation`. Use a `switch()` statement inside the function definition so that it exhibits the following behaviour:

- If `operation` is zero, return `a+b`
- If `operation` is one, return `a-b`
- If `operation` is two, return `a*b`
- If `operation` is three, return `a/b`
- If `operation` is anything else, print an error message (eg: `printf("Illegal operation %d", operation)`) and return zero.

This task can be broken down into the following sub-tasks:

1. Modify the function prototype to include the third argument `operation` of type `int`
2. Modify the first line of the function definition to include the third argument
3. ~~Draw the rest of the owl~~ Write the `switch` statement which makes a choice between the four arithmetic operations based on the value of `operation`.

   NB: You may write `return` at any point inside a function. As such, your `switch()` statement may either:
   (a) Allocate the arithmetic result to a temporary variable (of type `float`) followed by a `break` statement, `return`'ing the temporary variable at the end of the function, OR
   (b) have `return` statements inside the `switch()` statement, directly after each `case`.

Task 3: Functions: Prime Number Testing

Write a C function, `isPrime()` which tests if an integer argument is prime.

The return value and argument should both be `int` data types. If the argument is prime the function returns 1 and returns 0 otherwise.

For the purposes of this task you should perform a "brute force" search for any integer factor of the argument. ie: for an integer `x` test if it is divisible by all integers between 2 and `x/2+1`. If any factors are found the function should return zero.

Revision: An integer, `x`, is prime if it is not divisible by any other integer, `k`. In C, `x` is divisible by `k` if `x%k == 0`.

You may use the template below. It has the function prototype, function definition, and `main()` "test code" pre-written:

```
#include <stdio.h>

// Returns 1 if x is prime, zero otherwise
int isPrime(int x);

// main function for testing isPrime()
// Only modify if I made a mistake or you want to play around
int main() {
   int testNum;
   for(testNum = 0; testNum < 20; testNum++) {
      if(isPrime(testNum)) {
         printf("%d is prime\n", testNum);
      }
   }
   return 0;
}

// Returns 1 if x is prime, zero otherwise
// EDIT THIS BIT
int isPrime(int x) {
   return 0;
}
```

Task 4: (Arrays): Sieve of Eratosthenes

This task is mathematically simple but algorithmically complex. It is an excellent programming task if you find the mathematics of other tasks overwhelming.

The *Sieve of Eratosthenes* is an algorithm for generating prime numbers. The method is attributed to the ancient Greek scholar Eratosthenes of Cyrene (276-194 BC).

If implemented by hand, the algorithm requires the following steps:

1. Write a list of integers from 2 to some "large" number, N
2. Observe that 2 is prime and that any multiple of 2 would *not* be prime
3. Moving down the list, cross out every multiple of 2
4. Choosing the next highest number which is not crossed out, 3, move down the list and cross out every multiple of 3
5. Choosing the next highest number which is not crossed out, 5, move down the list and cross out every multiple of 5
6. Repeat until you reach the end of the list
7. All numbers which were not crossed out are prime

Example:

Listing integers from 2 to 16:
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Crossing out every multiple of 2:
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~

Crossing out every multiple of 3:
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~

Crossing out every multiple of 5 (ok, they're all crossed out by now already so we can stop):
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~

All the numbers left over are prime:
2 3 5 7 11 13

Task: Write a C program which uses this method to find all prime numbers strictly less than 100.

Since we need to remember a "long" list of numbers (and whether or not they are "crossed out") this is a perfect application for an array.

To complete this task, declare an array of type `char` with 100 elements:

```
char numbers[100]; // Index ranges from 0 to 99
```

The list of indices (plural of index) will form the list of integers. The array elements (the numbers stored at each index) indicate whether that index has been "crossed out" or not.

The array includes indices 0 and 1 which are integers not considered by the original algorithm. Your code may either ignore them or treat them as special cases when printing the result (ie: have explicit `if(index == 0 || index == 1) statements`).

The array should be initialised to all 1's to indicate that, at the start of the algorithm, nothing has been crossed out. As elements are "crossed out" they should be set to zero.

Your program should implement the following pseudocode:

```
BEGIN
  numbers[MAX+1] is an array of char's
  Set every element of numbers[] to 1

  FOR index = 1, 2, 3, ..., MAX
    IF numbers[index] is 1
      FOR j = 2*index, 3*index, 4*index, ... MAX
        numbers[j] = 0
      ENDFOR
    ENDIF
  ENDFOR
  Print every index for which numbers[index] is 1
END
```

You may use the following template:

```c
#include <stdio.h>

int main() {
   // Maximum integer to test. Declared constant
   // This is declared so that the maximum integer
   // can easily be changed
   const int MAX = 100;
   char numbers[MAX+1];
   int index;
   int j;

   // Initialised numbers[]
   for(index = 0; index <= MAX; index++) {
      numbers[index] = 1;
   }

   // DO THE SIEVE ALGORITHM HERE
   // for(index = 2; ...etc

   // Print results
   printf("The following numbers are prime:\n");
   for(index = 2; index <= MAX; index++) {
      if(numbers[index] == 1) {
         printf("%d\n", index);
      }
   }

   return 0;
}
```

HINT: The inner FOR loop in the pseudocode can be implemented with the C `for()` statement:

```c
for(j = 2*index; j < MAX; j = j + index) {
   // Do stuff
}
```

Extension: The Wikipedia article covering the Sieve of Eratosthenes describes several optimisations. Implement the one which only crosses out `index` values starting at $index^2$ (instead of `2*index`) and the algorithm exit condition of not attempting to cross out values larger than $\sqrt{MAX}$.

Task 5: Arrays: Statistics Calculations

Write a C program which accepts a "large" number of real numbers and calculates their sample mean and sample variance.

Revision: From statistics, the *sample variance*, s^2, of N samples, $X_1, X_2, ..., X_N$, can be calculated as:

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

There are many algorithms for calculating a data set's variance. For this task you should implement the reasonably robust *two-pass* algorithm which calculates the sample mean and uses this value to calculate the variance as follows:

```
integer n = 0
float sum1 = 0
float sum2 = 0

FOR each data point Xn
  n = n + 1
  sum1 = sum1 + Xn
ENDFOR

mean = sum1 / n

FOR each data point Xn
  sum2 = sum2 + (Xn - mean)*(Xn - mean)
ENDFOR

variance = sum2 / (n-1)
```

Your program should "input" the data via an array initialisation. This is to avoid having to manually repeatedly type test data into `scanf()` while debugging.

You may use the following template:

```c
#include <stdio.h>
int main() {
   // The mean of x[] is 5.000000
   // The variance of x[] is 7.500000
   // x[] has 9 elements
   float x[9] = {1,2,3,4,5,6,7,8,9};
   float variance, mean;
   float sum1 = 0, sum2 = 0;
   int n; // Counter variable

   // Calculate the sample mean (NB: we are assuming N=9)
   for(n = 0; n < 9; n++) {
      // Write your calculation here
   }

   // Calculate the sample variance
   // Write your own for() loop

   printf("Mean: %f\nVariance: %f\n", mean, variance);
   return 0;
}
```

Task 6: Functions: Numerical Integration

NB: *I belive the mathematics of this task is much more difficult to understand than the C implementation. Read the linked Wikipedia page and ask a demonstrator for an explanation if you don't follow it.*

Write a C program which implements the following equation:

$$f(x) = e^{x^2}$$

as a C function and uses the trapezoidal rule to numerically calculate its integral between two given points: x_1 and x_2 (assume $x_1 < x_2$). You should use the `expf()` function from the C maths library; be sure to `#include <math.h>`.

The function e^{x^2} is impossible to integrate by hand (ie: there is no "closed form" solution for $\int e^{x^2} dx$) so we are *forced* to use numerical methods such as the trapezoidal rule (or Riemann integrals, Simpson's Rule, etc) to evaluate it.

For the purposes of this task, the trapezoidal rule requires the interval, $[x_1, x_2]$ to be broken up into several smaller "chunks" of trapezoidal shape. The area of each of these trapezoids is easy to calculate and sum together.

Graphical illustration of the method can be found on Wikipedia. This task implements the method described under numerical implementation on a uniform grid.

To implement this, your program needs a number, `N`, equal to the number of trapezoids the $[x_1, x_2]$ interval is broken into. For this task, you may "hard code" a value of `int N=5`. You are encouraged to vary this value and observe how the accuracy of the result changes.

Given the interval, $[x_1, x_2]$, and trapezoid count, `N`, the horizontal width of each trapezoid, Δx, can be calculated as:

$$\Delta x = \frac{x_2 - x_1}{N}. \tag{2}$$

The integral of an arbitrary function, $f(x)$, may then be calculated with the following equation:

$$\int_{x_1}^{x_2} f(x)dx = \Delta x \left(\frac{1}{2} f(x_1) + \sum_{k=1}^{N-1} f(x_k) + \frac{1}{2} f(x_2) \right) \tag{3}$$

where x_1 is given and the x_k's inside the sum can be calculated as:

$$x_k = x_1 + k\Delta x.$$

In your C program, evaluation of $f(x)$ is simply written as `f(x)` so long as you have declared the variable `float x;` and have written a function with the prototype `float f(float x)`.

The evaluation of $\sum_{k=1}^{N-1} f(x_k)$ implies the use of a `for()` loop with the following form:

```
int N = 50;
int k; // Loop counter
float x1 = 0, x2 = 2;
float dx = (x2 - x1) / N;
// ... other code
float sum = 0;
for(k = 1; k < N; k++) {
   float x;
   x = x1 + k*dx;
   sum = sum + f(x)
}
```

With the values for $x1$ and $x2$ above the final answer should be ≈ 16.4526.

Task 7: Functions: Numerical Differentiation

In pure mathematics, the following formula is used to differentiate the function $f(x)$ at a point x_0 from first principles:

$$\frac{df(x_0)}{dx} = \lim_{h \to 0} \left[\frac{f(x_0 + h) - f(x_0)}{h} \right] \tag{4}$$

On a computer it is not possible to implement Equation (4) exactly because a division by zero causes numerical errors and is mathematically *a little bit naughty*.

Instead, we can *estimate* the differential (ie: gradient of $f(x_0)$) by choosing a small, but non-zero, value for h and acknowledging that the answer is not exact. Thankfully engineering never *requires* exact values so the approximation can still be very useful.

Explicitly, we calculate an estimate for the gradient as:

$$\frac{df(x_0)}{dx} \approx \left[\frac{f(x_0 + h) - f(x_0)}{h} \right] \tag{5}$$

for a "small" but non-zero value of h.

Task: Write a C program which calculates the gradient of $f(x) = x^2 + 2x - 1$ at a given point x_0.

Procedure:

1. Write a C function which takes a single argument of type `float` and returns a `float` equal to `x*x + 2*x - 1`. The function prototype should be:

```c
float f(float x);
```

2. Using the above function, write a *second* function which takes two `float` arguments, `x0` and `h`, and returns an estimate of the gradient of x^2+2x-1 at `x0` using Equation (5). The function prototype should be:

```c
float diff(float x, float h);
```

3. Test your program given that:

$$f\prime(x) = 2x + 2$$
$$f\prime(0) = 2$$
$$f\prime(1) = 4$$
$$f\prime(-1) = 0$$

You may use the following template:

```c
#include <stdio.h>
float diff(float x);
float f(float x);
int main() {
   float h = 0.0001;
   printf("%f\n", diff(0, h));
   printf("%f\n", diff(1, h));
   printf("%f\n", diff(-1, h);
}
float diff(float x) {
   float d; // Differentiation result variable
   d = ; // Fill in this line
   return d;
}
// Write the function definition of f(float x) below here
```

Task 8: Functions and Static Variables: Fibonacci Sequence Generator

Review: The Fibonacci Sequence is a sequence of integers:

$$x_1, x_2, x_3, ...,$$

where:

$$x_1 = 1$$
$$x_2 = 1$$
$$x_n = x_{n-1} + x_{n-2}$$

Task: Write a C function which, each time it is called, returns the next value of the Fibonacci sequence. This will require the use of `static` variables within the function. Use the following function prototype:

```
unsigned int fib(void);
```

You can use the following template:

```
#include <stdio.h>

unsigned int fib(void);

int main () {
   int n;

   for(n = 0; n < 10; n++)
      printf("%d %x\n", n, fib() ); //%x formats an unsigned int
}
```

After correctly implementing the function definition for `fib()` your program output should be:

```
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
9 34
```

Task 9: Mandelbrot Images - Advanced

Starting from a completed solution to the Week 3 Mandelbrot problem modify your code to generate an ASCII netpbm "portable PixMap" (PPM) formatted image.

PPM Image Format

The PPM format specification allows images to be written to files using only plain text (ASCII) data. This results in a *very* simple file format which is easy to create with basic programs.

The file header has the following format:

```
P3
<x resolution> <y resolution>
<max value>
```

where:

- `P3` is a "magic number" identifying the format
- `<x resolution>` is the number of pixels horizontally
- `<y resolution>` is the number of pixels vertically
- `<max value>` is the pixel value corresponding to maximum brightness. Typically `255`.

An example header for a 100x100 pixel image would be:

```
P3
100 100
255
```

After the header comes the *body*. This is a list of **integer** RGB triplet numbers, separated by whitespace. The exact format is not critical, but I recommend the following:

```
R G B
R G B
R G B
...etc
```

The pixel data scans left-to-right top-to-bottom (ie: top left pixel is first, bottom right pixel is last). This is the same order in which the Week 3 task calculated pixel values.

Colouring Fractal Images

In order to colour a fractal image a mapping is used which converts each pixel's *iteration count* to a colour. There are a near-infinite number of choices for this and I strongly recommend you research so-called "colorbar" creation. For this task I recommend the following two options:

1. For debugging, use a grayscale mapping which uses the following formula for a pixel brightness, p, given the number of `iters` a pixel took to escape:

   $$p = 255 \times \frac{\texttt{iters}}{\texttt{MAX\_ITERS}}$$

   To create greyscale images the RGB values of a pixel must be equal. This means that after you calculate p you can just use

   ```
   R = p;
   G = p;
   B = p;
   ```

   to make R, G and B equal. **NB:** The PPM format requires the output numbers to be **integers**!

2. More interesting colour mappings can be built with the use of the HSL (Hue-Saturation-Luminance) colourspace. This is a set of 3 numbers (H, S and L) which define a unique colour based on their hue (base colour from the rainbow), saturation ("colourfull-ness") and

luminance (brightness).

You can then slightly tweak the previous formula to provide a H value and fix S and L at 1.0:

$$H = 360 \times \frac{\texttt{iters}}{\texttt{MAX\_ITERS}}$$
$$S = 1.0$$
$$L = 1.0$$

Because the PPM image format only supports RGB pixel formats you will need to implement formulas which convert HSV triplets to RGB values. Wikipedia lists the set of formulas. Implement them in code, either as three functions or just as "inline" code.

You can attempt to write a single function which accepts a pointer to an array of 3 `floats` but we won't cover the details of this for a couple of weeks.

After RGB values have been calculated the output file data can be written.

Creating Files

For this task we have two options for creating a file:

1. Use command line output redirection
2. Use the C file I/O library

The first option keeps your program simpler but requires the use of a command line when running the program. If your code simply uses `printf()` to output the PPM data (and *nothing* else) it can be written to a file with:

```
# ./a.out > image.ppm
```

The file `image.ppm` can then be downloaded (if you're using Che etc) or directly opened with an image editor. I know `eog`, `gimp`, etc open PPM on Linux and am naively hoping that something on Windows supports it.

The alternative, which will work with C development environments which don't have command lines like OnlineGDB, is to use the C file I/O library. To do this you need several lines of code and an extra function:

```
1 FILE *output; // A pointer variable of type FILE*
2 output = fopen("image.ppm", "w"); // To open the file image.ppm
3 fprintf(output, "%d %d %d\n", r, g, b);
```

The `FILE *output` line is a variable declaration, only do this *once*. Likewise, `output = fopen...` opens the file `image.ppm` for writing. Only do this *once* as well.

The `fprintf()` function is almost exactly the same as `printf()` except it takes an extra argument: the file to write to. In this case passing `output` (the `FILE *` variable declared earlier) will write the formatted text to that file.
Lets see if somebody gets this to work given the above plus your own independent research. We will formally cover file I/O in a few weeks (check the course outline for a planned week).

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

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;
}
```

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}

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

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).

`if()` Syntax:

`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");
```

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.

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 { }).

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:

$$\sum_{n=0}^{\infty} \frac{1}{n^2} \tag{6}$$

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.

4.10.4 `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.

4.10.5 `switch() case:`

The `switch() case:` statement is a flow control mechanism which allows one option from a list of many to be executed. It is a neater replacement from complicated `if() else if()` blocks.

Example:

```
switch(x) {
   case 1: // Jumped to if x == 1
      break; // Jump out of switch()
   case 2: // Jumped to if x == 2
      break;
   case 3: // Jumped to if x == 3
      break;
   default: // Jumped to if nothing
    else matched
}
```

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 + 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.

4.12 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 numerical 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`.