

ENGG1003 - Lab Week 5

Brenton Schulz

1 Introduction

This lab covers C *functions*, static variables, and arrays

If I have time the introduction will be expanded with examples, etc. Apologies if this doesn't happen.

2 Tasks

Task 1: 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 evenly 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 2: 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, \dots, X_N , can be calculated as:

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

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 values 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 type test data into `scanf()` multiple times while debugging.

You may use the following template:

```
#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
    for(n = 0; n < 9; x++) {
        // Write your calculation here
    }

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

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

Task 3: Functions: Numerical Integration

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 $x_1 < x_2$).

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.

Task 4: 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:

$$\frac{df(x_0)}{dx} = \lim_{h \rightarrow 0} \left[\frac{f(x_0 + h) - f(x_0)}{h} \right] \quad (2)$$

On a computer it is not possible to implement Equation (2) 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:

$$\frac{df(x_0)}{dx} \approx \left[\frac{f(x_0 + h) - f(x_0)}{h} \right] \quad (3)$$

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 $x^2 + 2x - 1$. The function prototype should be:

```
1 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 (3). The function prototype should be:

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

3. Test your program given that:

$$\begin{aligned} f'(x) &= 2x + 2 \\ f'(0) &= 2 \\ f'(1) &= 4 \\ f'(-1) &= 0 \end{aligned}$$

You may use the following template:

```
#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 5: Functions and Static Variables: Fibonacci Sequence Generator

Review: The Fibonacci Sequence is a sequence of integers:

$$x_1, x_2, x_3, \dots,$$

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:

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

You can use the following template:

```
1 #include <stdio.h>
2
3 unsigned int fib(void);
4
5 int main () {
6     int n;
7
8     for(n = 0; n < 10; n++)
9         printf("%d %x\n", n, fib() ); // %x formats an unsigned int
10 }
```

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 6: 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:

$$p = 255 \times \frac{\text{iters}}{\text{MAX_ITERS}}$$

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:

$$H = 360 \times \frac{\text{iters}}{\text{MAX_ITERS}}$$

$$S = 1.0$$

$$L = 1.0$$

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).

3 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.

3.1 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:

```
1 #include <stdio.h>
2 int main() {
3     // Your program goes here
4     return 0;
5 }
```

3.2 Comments

```
1 // This is a comment to end of line
2
3 /* this is a block comment
4    which could span
5    multiple
6    lines
7    */
```

3.3 Code Blocks

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

3.4 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

3.5 Operator Shorthand

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

<code>x = x + y;</code>	<code>x += y;</code>
<code>x = x - y;</code>	<code>x -= y;</code>
<code>x = x * y;</code>	<code>x *= y;</code>
<code>x = x / y;</code>	<code>x /= y;</code>

3.6 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, 1.8×10^{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.

3.7 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:

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

This prints:

Enter a number: _

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

3.8 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

3.9 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.

```
1 float a;
2 // ...
3 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.

3.10 Flow control

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

3.10.1 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:

```
1 if(x > 10) {
2     // Do stuff
3 }
```

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:

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

3.10.2 if() ... else if()

The C syntax for IF ... ELSE is:

```
1 if(condition) {
2     // Do stuff
3 } else {
4     // Do stuff
5 }
```

IF ... ELSEIF takes the form:

```
1 if(condition) {
2     // Do stuff
3 } else if(condition) {
4     // Do stuff
5 }
```

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 `{ }`).

3.10.3 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:

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

to a precision of 1×10^{-6}

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

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

3.10.4 for(;;)

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

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

```
1 for( initial ; condition ; increment )
2 {
3     // Do stuff
4 }
```

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:

```
1 int x;
2 for( x = 0 ; x < 10 ; x++ ) {
3     printf("%d ", x);
4 }
```

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.

3.10.5 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.

(4) Example:

```
1 switch(x) {
2     case 1: // Jumped to if x == 1
3         break; // Jump out of switch()
4     case 2: // Jumped to if x == 2
5         break;
6     case 3: // Jumped to if x == 3
7         break;
8     default: // Jumped to if nothing
9         else matched
10 }
```

3.11 Library Functions

3.11.1 rand()

To generate a random number between 0 and MAX:

```
1 #include <stdlib.h> // For rand()
2 // ...
3 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](#).

3.12 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`.