ENGG1003 - Lab 3

Brenton Schulz

1 Introduction

This lab is written to prepare you for the week 4 assessment, it covers material similar to that in the assessment task.

All these tasks are based on pure mathematics which I can reasonably expect you understand from high school. Sequences and series are described in Section 7 of the NSW Mathematics 2/3 Unit syllabus. I understand that some of you have only completed General mathematics or have not studied mathematics for many years. To achieve a passing grade you will only need to understand very basic examples.

These tasks may not create the most *interesting* programs but I need to set a solid foundation of knowledge among the whole cohort before we can tackle more complicated (and realistic) problems.

2 Change: `scanf();` Not Used This Week

In order to reduce the amount of C language features you will require for the assessment task I have decided to not include user input via `scanf();` in this week's lab.

Instead, use variable initialisation as "input". Type input values directly into your code when variables are declared.

For an example please see the Friday Week 2 lecture at timestamp 29 min 41 sec where this technique is used in the code which factorises a number.

3 Eclipse Che (Advanced)

If you wish to use Eclipse Che to complete this lab you are welcome to. As noted in last Friday's lecture it does not, by default, accept `stdin` but as seen above this will not be required yet.

Please note that a Che server has a hard limit of 30 concurrent users (thanks for discovering that on Friday!). As such, I have deployed a second server (they are named `che1` and `che2`). If your "workspace" doesn't load then please try the second server.

The ***first time*** you use either of the Che servers click either of these links and login via the "GitHub" link at the bottom of the login page:

`http://che1.vk2dds.net:8080/f?id=factorypqq4zicbo4hin59w`

`http://che2.vk2dds.net:8080/f?id=factorynhhgh4tkbklv5t4j`

That link will create a workspace in your account and load several code examples from GitHub. *You only need to do this once!*

On subsequent logins please use these links:

`http://che1.vk2dds.net:8080`

`http://che2.vk2dds.net:8080`

For now the only documentation is the demonstration given in lectures. More will be created in the future as I have time.

At time of writing university IT claims to have fixed Che on the laboratory computers but I have not yet tested it personally.

Task 1: Arithmetic sequence

An arithmetic sequence is a list of numbers, $x_1, x_2, x_3, ...$, where each number, x_n, is a fixed increase, d, from the previous one:

$$x_n = x_{n-1} + d. \tag{1}$$

The difference, d, is therefore:

$$d = x_n - x_{n-1}. \tag{2}$$

Given an initial value, x_1, and *either* the difference, d, or second value, x_2, the series can be uniquely described. For example, the sequence:

$$0, 2, 4, 6, ... \tag{3}$$

is described by $x_1 = 0$ and $d = 2$. It could also be calculated given $x_1 = 0$ and $x_2 = 2$ and the value of d calculated to be: $d = x_2 - x_1 = 2$.

In more human language: an arithmetic sequence is counting in steps of d.

Task: Write a C program which takes two initial values, x_0 and x_1 and generates further numbers in their arithmetic series.

Your program should follow the following rules:

- It should print the initial values
- It should print a total of 10 numbers
- It should print each number on a new line
- Numbers in the sequence can be positive or negative
- Numbers in the sequence must be integers

The initial values are to be "hard coded" as variable initialisations. Do not read them from the console. Your program is only given the first two numbers, it needs to calculate everything else required to print the following 8 values.

Check if your code works by comparing its output with a sequence calculated by hand. Test several combinations of x_1 and x_2.

The following template can be used (if you are using a new Che factory it is already loaded or you can pull it from `https://github.com/bschulznewy/seriesTemplate.git`):

```
#include <stdio.h>
int main() {
  int x1 = 0;
  int x2 = 2;
  int d;
  int xn; // the current x value
  int xnm1; // xn minus 1, the previous value

  // d = ... COMPLETE THIS EXPRESSION

  while ( /*condition*/ ) {
    // Calculate current x value
    xn /* = ... */;
    // printf(" "); // FILL ME IN
    xnm1 = xn; // Make past value into present value
  }
  return 0;
}
```

Task 2: Arithmetic Series

An arithmetic *series* is the sum of the numbers in an arithmetic sequence.

Modify your code from the previous task to calculate the sum of the first 10 numbers in the arithmetic sequence.

Test your result given the following formula for the sum, S_n, of the first n terms, x_1 to x_n, being:

$$S_n = \frac{1}{2}n(x_1 + x_n) \tag{4}$$

You will need to declare a new variable, `sum`, to "accumulate" the sum as the series is calculated.

Task 3: Geometric Sequence

A geometric sequence is a set of numbers, $x_1, x_2, x_3, ...$, where each number is a constant *multiple* of the previous one:

$$x_n = kx_{n-1}. \tag{5}$$

The value of k can be calculated given any two successive numbers in a geometric sequence:

$$k = \frac{x_n}{x_{n-1}}. \tag{6}$$

Write a C program which is given two numbers, x_0 and x_1, and generates further numbers in their geometric sequence. You will need to calculate k inside the program.

Your program should follow the following rules:

- It should print the initial values
- It should print a total of 10 numbers (ie: n = 0 to 9)
- It should print each number on a new line
- Numbers in the sequence can be positive or negative
- Numbers in the sequence may be any real number

Task 4: Geometric Series Pt I

A geometric *series* is the sum of a geometric sequence.

Modify your geometric sequence program to calculate the sum of the first n terms (ie: $x_1+x_2+...+x_n$). The value of n must be declared as a variable with a given initial value. Test your program with a variety of values for n and initial values, x_1 and x_2.

For testing, you should know that given an initial value, x_1, and the constant multiplier, k, the sum of the first n terms, S_n, of a geometric series can be calculated as:

$$S_n = x_1\frac{1 - k^n}{1 - k}. \tag{7}$$

Evaluate this formula *by hand* (or with Google / Wolfram Alpha / etc) to generate test values.

Task 5: Geometric Series Pt II

If the value of the multiplier, k, in a geometric series is between (but not equal to) -1 and 1 then the series is said to *converge*. This means that, as n increases towards infinity, the sum of the series moves towards a fixed, finite, value. If the series grows towards infinity then it is said to *diverge*. The case of $k = 1$ is not considered here.

Write a C program which, given x_1 and x_2, calculates their geometric series and decides if the series is convergent or divergent. Your program should exhibit the following behaviours:

- All numbers in the geometric sequence can be assumed to be positive
- If the series diverges the program prints the first 10 values then exits
- If the series converges the program calculates the sum to within the following limits:
  - A precision of 1×10^{-4}, OR
  - A maximum of 100 terms

Compare your program's 100th output with the theoretical formula:

$$S_\infty = x_1 \frac{1}{1-k} \tag{8}$$

NB: The precision limit can be implemented by testing the size of each term before it is added to the running sum. If it is less than the specified precision value the program may assume that the series has converged and exit.

At this point I believe that you have completed enough content to comfortably complete the week 4 assessment task. Continue below if you want to easily achieve a high distinction grade.

Task 6: More Complicated Sequences Pt I (Advanced)

All previous sequences have had the property that a sequence value, x_n, only depends on x_{n-1}. Here we extend that idea to sequences for which $x_n = f(x_{n-1}, x_{n-2})$. See footnote[a]. Each value depends on the previous *two* values.

As seen in lectures, the Fibonacci sequence is an example of this kind of sequence. It follows the rule:

$$x_n = x_{n-1} + x_{n-2} \tag{9}$$

with the initial values of $x_1 = 1$ and $x_2 = 1$.

We will extend Equation (9) by introducing two coefficients: b_1 and b_2[b]:

$$x_n = b_1 x_{n-1} + b_2 x_{n-2}. \tag{10}$$

Although the Fibonacci sequence is restricted to integers, assume any code written for this task requires `float` data types.

The actual task: Write a C program which is given initial values for x_1, x_2, b_1 and b_2 and calculates the resulting sequence as-per Equation (10).
You may use the following template (should be copy+past'able):

```
#include <stdio.h>
int main() {
  float b1 = 1, b2 = 1;
  float x1 = 1, x2 = 1;
  // Declare other variables needed

  while( /* condition */) {
    // Calculate and print the sequence
  }

  return 0;
}
```

Exit conditions for your program are not specified. You need to come up with exit conditions which allow you to test if your program is correct.

Apart from the Fibonacci sequence, you may use the following test sequence with its first ten values:

```
b1 = 0.2, b2 = 0.8
x1 = 1, x2 = 0.5
Output:
1.000000
0.500000
0.900000
0.580000
0.836000
0.631200
0.795040
0.663968
0.768826
0.684940
```

[a] This is read aloud as "x_n is a function of x_{n-1} and x_{n-2}"
[b] Great, now I have the *Bananas in Pajamas* theme in my head.

Task 7: More Complicated Sequences Pt II (Advanced)

The initial condition, x_1 and x_2, and the values of b_1 and b_2 will greatly influence the behaviour of a second order sequence. For example:

- It may converge towards zero
- It may converge to some fixed, non-zero, value
- It may diverge to infinity like the Fibonacci sequence
- It may *oscillate*
  - ie: Wiggle up and down in a sinusoidal pattern
  - This oscillation may converge to zero or diverge to infinity

Task: Modify your second order sequence code to test for *oscillation*. This can be done by calculating the *difference* between subsequent numbers in the sequence and observing whether the difference changes sign (ie: Do they swap from positive to negative or are they *always* positive or negative?).

To complete this task will need to declare a "flag" variable. It can be of type `int` and have any name of your choosing. Given the initial values of x_1 and x_2 calculate the difference: $x_2 - x_1$. If it is positive, make the flag zero. If it is negative, make the flag one. You may assume that the difference will always be non-zero (ie: the initial values will never be equal).

Each time a new entry in the sequence is generated re-calculate the value $x_n - x_{n-1}$. If, at any point, the sign of this calculation is different to the sign of $x_2 - x_1$ the program should print that the sequence oscillates and exit.

If, after 20 iterations, no change of sign has been observed the program should print that no oscillation occurred and exit[a].

Example 1: For the initial values `x1 = 1, x2 = -0.5` and coefficients `b1 = 0.2, b2 = 0.2` the sequence is:

```
 1.00000
-0.50000
 0.10000
-0.08000
 0.00400
-0.01520
```

and the sequence oscillates (first difference is -1.5, then +0.6).

Example 2: For the same coefficients as above but initial values `x1 = 1, x2 = 0.5` the sequence is:

```
1.000000
0.500000
0.300000
0.160000
0.092000
```

and no oscillation occurs as the difference is always negative (ie: the sequence only gets smaller and moves towards zero).

[a] This does not prove that oscillation will *never* occur, just that if it does it takes a "long time" to be detected

Task 8: Mandelbrot Fractal Generator (Advanced)

Only attempt this task if everything else has been completed. It does not related to the upcoming assessment task and is only provided for interested students to extend their skill beyond reasonable expectations.

Demonstrators are not to help with this task. Please do not take their time away from students who need help with the other tasks.

This task assumes that you have completed the Mandelbrot task in Lab 2.

Extend your code to test a grid of points and print an "ASCII art" Mandelbrot set.

To do this you should use two *nested* WHILE loops:

```
BEGIN
  float x = -2
  float y = 1
  WHILE y > -1
    x = -2
    WHILE x < 2
      Re-initialise the Mandelbrot iterator test variables
      Test if the point (x,y) is inside the Mandelbrot set
      IF (x,y) is inside the set
        PRINT #
      ELSE
        PRINT " " (a space character)
      ENDIF
      x = x + 0.05
    ENDWHILE
    PRINT a new line
    y = y - 0.05
  ENDWHILE
END
```

If you understand how, the WHILE loops may be replaced with FOR loops. The C syntax for FOR loops is documented in Section 4.10.4.

When correctly implemented, the pseudocode above will draw the following 80 x 40 pixel image of the Mandelbrot set:

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:

$$\sum_{n=0}^{\infty} \frac{1}{n^2} \tag{11}$$

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.

4.11 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`.