ENGG1003 - Lab 2 (And Beyond)

Brenton Schulz

1 Introduction

This lab is written to prepare you for the week 4 assessment. The assessment task will be closely related to material in this lab. Every lab time will be given a different problem to solve.

Task 1: Arithmetic series

An arithmetic series is a sequence of numbers, $x_1, x_2, x_3, ...$ where each number, x_n, is a fixed increase from the previous one, x_{n-1}. An arithmetic series is defined by the initial value, x_0, and the difference, $d = x_n - x_{n-1}$.

For example, the sequence:

$$0, 2, 4, 6, ... \tag{1}$$

is defined by $x_0 = 0$ and $d = 2$.

Write a C program which takes two initial values, x_0 and x_1 and generates further numbers in that 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

The initial values are to be "hard coded" as variable initialisations. Do not read them from the console.

2 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.

2.1 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;
}
```

2.2 Comments

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

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

2.3 Code Blocks

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

2.4 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

2.5 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;`

2.6 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

2.7 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).

2.8 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

2.9 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.

2.10 Flow control

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

2.10.1 `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");
```

2.10.2 `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 { }).

2.10.3 `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:

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

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.

2.10.4 `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.

2.11 Library Functions

2.11.1 `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.

2.12 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`.