



INDIVIDUAL ASSIGNMENT COVERSHEET

STUDENT ID	FIRST NAME	SURNAME	
16945724	Nikkolas	Diehl	
Paper Code:	COMP500	Paper Name:	Programming 1
Assignment Name:	Reporting Journal - Stage 1		
Lecturer's Name:	Steffan Hooper		
Assignment Due Date:	24/03/2017	Date Submitted:	24/03/2017

Please read the following and **tick**  to indicate your understanding:

- I understand it is my responsibility to keep a copy of my assignment.
- I have signed and read the **Student's Statement below**.
- I understand that a software programme (Turnitin) that detects plagiarism and copying may be used on my assignment.

Tick 

Tick 

Tick 

Plagiarism and Dishonesty are methods of cheating for the purposes of General Academic Regulations (GAR) <http://www.aut.ac.nz/students/regulations.htm>

Student's Statement:

This assessment is entirely my own work and has not been submitted in any other course of study. I have submitted a copy of this assessment to Turnitin, if required.

In this assessment I have acknowledged, to the best of my ability

- The source of direct quotes from the work of others
- The ideas of others (includes work from private or professional services, past assessments, other students, books, journals, cut/paste from internet sites and/or other materials)
- The source of diagrams

Signature: 

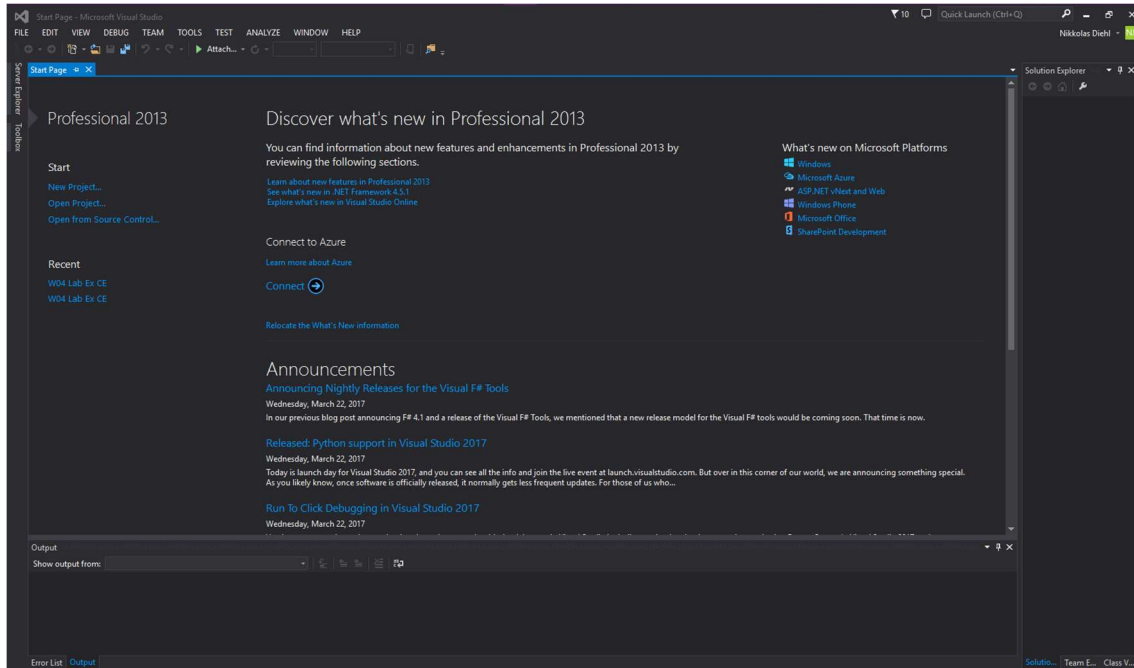
Date: 20/03/2017

The information on this form is collected for the primary purpose of submitting your assignment for assessment. Other purposes of collection include receiving your acknowledgement of plagiarism policies and attending to administrative matters. If you choose not to complete all questions on this form, it may not be possible for the Faculty of Design and Creative Technologies to accept your assignment.

Entry ID:	Date:	Day	Start Time:	Duration:	Session Type:	Location:
Week 1						
1	27/02/2017	Monday	9:00am	1 hour	Lecture – 1	WG403
2	27/02/2017	Monday	10:00am	3 hours	Lab Tutorial – 1	WT201
3	02/03/2017	Thursday	8:00am	1 hour	Lecture – 2	WG403
4	03/03/2017	Friday	11:00am	1 hour	Lecture – 3	WG403
WEEK 2						
5	06/03/2017	Monday	9:00am	1 hour	Lecture – 4	WG403
6	06/03/2017	Monday	10:00am	3 hours	Lab Tutorial – 2	WT201
7	09/03/2017	Thursday	8:00am	1 hour	Lecture – 5	WG403
8	09/03/2017	Thursday	10:00am	3 hours	Self-Study	AUT Library
9	10/03/2017	Friday	11:00am	1 hour	Lecture – 6	WG403
10	10/03/2017	Friday	1:00pm	5 hours	Self-Study	Home
WEEK 3						
11	12/03/2017	Monday	9:00pm	1 hour	Lecture – 7	WG403
12	12/03/2017	Monday	10:00am	3 hours	Lac Tutorial – 3	WT201
13	12/03/2017	Monday	8:00pm	4 hours	Self-Study	Home
14	16/03/2017	Thursday	8:00am	1 hour	Lecture – 8	WG403
15	17/03/2017	Friday	11:00am	1 hour	Lecture – 9	WG403
16	18/03/2017	Saturday	3:00pm	4 hours	Self-Study	Home
WEEK 4						
17	20/03/2017	Monday	9:00am	1 hour	Lecture – 10	WG403
18	20/03/2017	Monday	10:00am	3 hours	Lab – 4	WT201
19	20/03/2017	Monday	7:00am	2 hours	Self-Study	Home
20	23/03/2017	Thursday	8:00am	1 hour	Lecture – 11	WG403
21	24/03/2017	Friday	11:00am	1 hour	Lecture – 12	WG403

1. 27/02/2017: Lecture - 1: WG403a. **Program Descriptions:**

- i. I simply got an introduction into the course and learnt the values and outcomes of taking said course. I got a simple understanding of what the course will go through and all its contents and I got a look at some basic starter code.
- ii. I learned about the debugger window and its basic uses as well as got a basic understanding for what programming IDE we were going to be using for the year/semester and how it all worked. (visual basic)

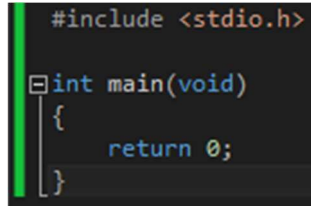
b. **Lessons learnt:**

- i. An introduction into the course. We learnt who the teacher and lecturers were, and that to pass the year. It is vital to attend every lecture and every lab. We had to get 80% or over attendance to pass the year, meaning it is only feasible to miss 3 labs out of the 12 in the year before we fail. Of course, 100% attendance is much more desired.
- ii. The values and outcomes of taking said course. The outcomes include being able to code in C, at a high level from which we can use in the working world. Another outcome is the ability to understand the practice and values of coding and using coding correctly; morally and ethically so as to *follow the rules* and abide by them. The values include following orders and sticking to a tight schedule of what to do and how to do it.
- iii. I got a simple understanding of what the course will go through and all its contents. We would be going through a lot of the C language throughout the semester and be required to take 2 assignments/exams and an end of semester exam.
- iv. I got a look at some basic starter code. The starter code we looked at was simply how to set up the main body of the code using `#include <stdio.h>` and `int main(void)` set up and initialisation. In this specific lecture, we only got to see a `printf` command to print text to the console/screen for the user/coder to view.

2. 27/02/2017: Lab Tutorial - 1: WT201

a. **Program Descriptions:**

- i. The program learnt on the 27/02/2017 was a simple program that was used to print text to the consol. Within the program, I learnt about escape sequences and how to display new lines, back slashes, commas, quotation marks and so on.
- ii. The Program was only a `#include <stdio.h>` with a simple `int main(void)` with the print functions inside it. There was no additional code besides the `return 0;` at the end.



```
#include <stdio.h>

int main(void)
{
    return 0;
}
```

b. **Known bugs or errors:**

- i. Forgetting simple semi colons and forgetting the new line code (`\n`) sometimes would cause the text to be broken and look weird. Also, forgetting to put an escape sequence where there needs to be sometimes breaks the code.

c. **Lessons learnt:**

- i. I learnt how to alt select and type on multiple lines.
- ii. I learnt how to code in simple C.
- iii. I learnt how to look out for bugs and bug test on a very simple degree.
- iv. I learnt how properly fix and select all text symbols that needed an escape sequence for them to work. Before I was looking for them all individually, but I found I could easily push CTRL + H and then select everything that needed a `\` and add one.

3. 02/03/2017: Lecture - 2: WG403a. **Lessons learnt:**

- i. Algorithms: A procedure for solving problems. They are instructions that describe a computation. They are a self-contained, finite number of step-by-step operations. They start at initial starting point and follow through a list of operations until they get to an end and may contain both input and output.
- ii. Before a program can be written, before a programmer even goes near a computer, it is imperative that they write down a detailed step by step guide on exactly what they want the program to do:

b. **Flow charts (pseudo code):**

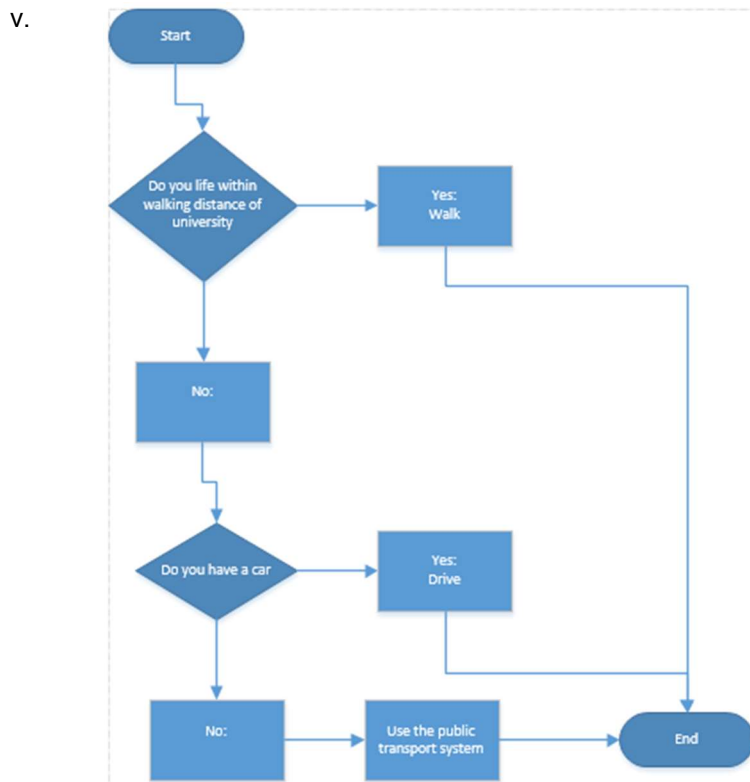
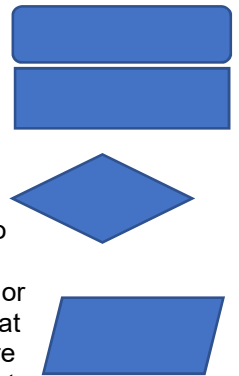
- i. Do you live within walking distance from University?
 - 1. Yes – Walk
 - 2. No – Go to step 2
- ii. Do you have a car?
 - 1. Yes – Drive
 - 2. No – Public transport.
- iii. It is important to break down the system into small and refined steps in a numbered list. They need to be direct and self-contained. They can't be too simple and they can't be too complicated that they extend beyond a single step.

c. **A good algorithm:**

- i. Define a task clearly, Solve a single task at a time. They algorithm should do a job well and handle all possible errors. And it needs to report errors if data is used.

d. **Flow charts:**

- i. This shape is a terminator. It defines the start and the end of a program.
- ii. This shape represents a step or an instruction. Each box needs detailed and concise operation of step.
- iii. This shape box represents a decision or choice. These are used when there is an if, or an else, or an ifelse. If there are two or more options for a program to follow. Like in the above example. Do you, live within walking distance from UNI? If it is a yes, they you can walk, if it's a no, they you go to the next step. That would be called an ifelse.
- iv. This box defines an input or an output. If you want to scan/input a variable or a number or anything, this box defines that. This also defines any outputs that come from the program. You might put this as an output for looking at where in the ram a variable is stored before moving to the next step of printing that variable.



e. **Lessons learnt:** White Space

- i. White space is built up of characters that don't have any visible effect or anything at all. White space is space made up from ¶ or space, or anything else that takes up a space (including new lines)
- ii. The lectures taught us that there is good white space and bad white space. Each new line should not have a line in-between in it, unless it is to separate sections of code. There should one – two lines of white space before the return 0; module, and indenting correctly is extremely important.

4. 03/03/2017: Lecture 3: WG403a. **Lessons learnt: Hardware:**

- i. RAM = Random access memory. Any byte in memory can be accessed at will without needing to access the proceeding byte of memory.

b. **Software:**i. Byte: Binary Term

1. A unit of storage capable of holding a single character.
2. 1 byte = 8 bits (stores 127 different values)

c. **Lessons learnt: Programing code**

i. VARIABLES:

1. A variable is a name given to any byte of the computer's memory and stores a piece of information.
2. Variables can be named with letters, digits and underscores. But the name can't start with a digit.
3. Each variable used in C must specify the type of into to store in the memory.

ii. Different types of variables:

1. Unsigned and signed char and char.
2. Short and unsigned short
3. Int and unsigned int
4. Long and unsigned long.

- iii. These are all different types of variables and each one holds a different amount or type of information. All the ones above all hold a numerical value with char holding -128 to 127 and the int holding -2 billion to around 2 billion numerical values. Unsigned version contains only positive numbers.

iv. Floating point

1. These types of variables hold real numbers up to six decimal numbers of accuracy. They allow the computer to store numbers with values bellow and in-between the decimal values.
2. Float = 4 bytes or 6 decimal places, and a double = 8 byte or 15 decimal places.

- v. Local Variables: Variables that are within the main function as they are declared in the main.

d. **Lessons learnt: Variable Assignment:**

- i. To save information into a variable we must write to the computer memory (RAM) when we assign data to a variable.

e. **Lessons learnt: Debugger watch window:**

- i. The visual studio debugger allows programmers to trace the flow of a program as it executes. You can view the contents of a variable. Make a break by clicking on the position you want it and step along through the program by clicking F10.

f. **Lessons learnt: Manipulation of variables: Mathematics:**

- i. Addition: +
- ii. Subtract: -
- iii. Multiply: *
- iv. Divide: /

1. Setting a and b as variables, you can use them for maths. $A + B$ and so on.
2. If you set $A = A + B$. It over writes A as the answer to $A + B$.

The screenshot shows a C program in a code editor. The code defines two integers, A and B, both set to 5. It then calculates A + B and prints the result. The output window shows the number 10.

```
#include <stdio.h>

int main(void)
{
    int A;
    int B;

    A = 5;
    B = 5;

    A = A + B;

    printf("%d\n\n", A);
    return 0;
}
```

Output: 10

v. Increment: ++ or -

1. $E++$; is the same as $e = e + 1$
2. $E--$; is the same as $e = e - 1$

vi. Modulus: %

1. $A \% B$ gives the remainder of A divided by B. If A was 7 and B was 2: $A \% B$ is 7/2 with a remainder. The modulus operation shows us that remainder.

vii. Prefix vs Pro-fix:

1. When using ++ or – for increments. It is important to decide where you place the ++ or – code. If you place it in front of the variable: `h = ++b;` It first gets b+1 then sets h to that answer.
2. If you place the ++ or – on the other side: `h = b++;` then it first sets h to b, then adds 1.

viii. Compound assign:

```
int f = 10;  
f = f + 10;
```

1. is the same as saying

```
f += 10;
```

(get f, go f+5)

2. This can also work with +=, -=, *=, and /=

g. **Formatting of printf:**

```
printf("example is %d \n", example);
```

- i. When run, the %d is replaced with the example variable. Any variables after the comma need to be put in order that you want them to display.
- ii. %d and %i mean an int and %f means a floating number.
- iii. %c means ASCII character and %s means a string.

5. 05/03/2017: Lecture - 4: WG403

a. **Lessons learnt: Printf formatting:**

- i. Printf (f means formatting)
- ii. %d means to replace that command with an int variable. Alternatively, you can use %i
- iii. %f is a float
- iv. %x is a hexadecimal value.
- v. %% means to print one percent.

b. **Lessons learnt: Printf % formatting with a d for example:**

- i. %(number) d = The number is the width of the full print out at the least.
- ii. %- (number) d = This left aligns the text printed.
- iii. %0 (number) d = 0 means it will fill the spaces that are created from the width being larger than a print out.
- iv. %+ (number) d = adds a + symbol to the print out.
- v. %. (number) f = determines the accuracy of a decimal number.
- vi. EXAMPLE: %6.2f = 6 is the width of the full output with a decimal accuracy of 2
- vii. FINAL OUTPUT FORMATTING: %(flags) (width) (. precision) (length)specifier.

c. **Lessons learnt: Input/output:**

- i. Every piece of code is saved into an address somewhere on the ram. To output the address (hardly needed) You need to use: & symbol in front of the variable in the print list. If you want to input a value from the user, you need to save it into an address.

```
printf("example is at : %p / n", &example);
```

d. **Scanf**

- i. Standard input

```
int a = 0;  
  
scanf("%d", &a);
```

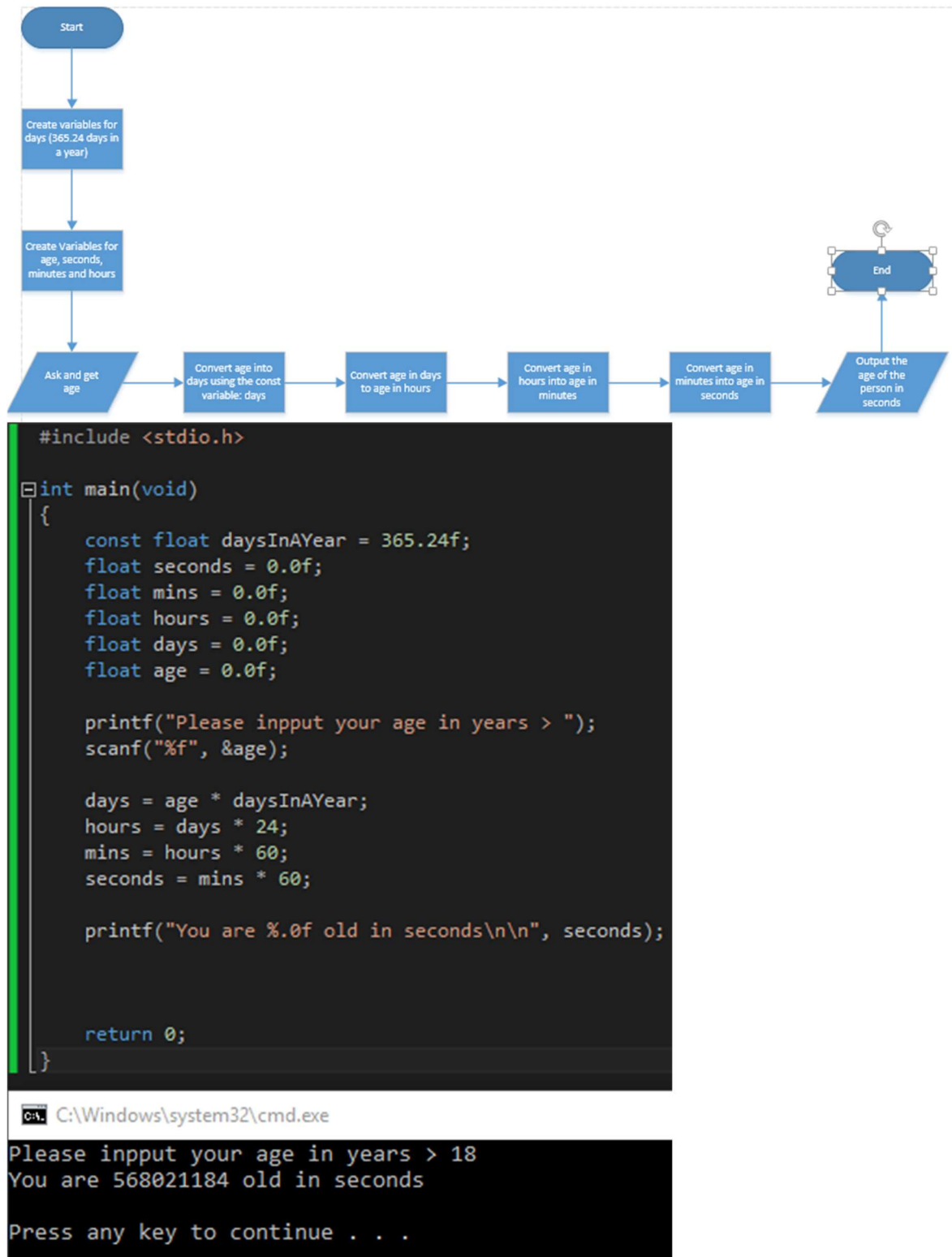
e. **CONST. A constant variable that cannot be changed by any other means. Replacing magic numbers or repeating numbers that pop up a lot in your code, with a constant variable already set is good:**

- i. USE & to scan information into the system – This is very important as without it, the program could crash and wouldn't work.

6. 05/03/2017: Lab 2: WT203:

a. Program Descriptions:

- i. As teams of two, we were given the task of creating a simple program that used all the pieces of code we had learnt so far in the labs and the lectures. Me and my team mate decided to create a code that asked for your age and changed it into seconds to tell you how old you are in seconds.



- ii. The code started with the usual `#include <stdio.h>` and declared the `main(void)` and then set a couple of variables and set them all to 0 for starters. But it declared one of the variables as a constant (so that I was using a constant variable as the task sheet said) and set it as `daysInAYear = 365.24` and set it as a floating-point variable.

- iii. The other variables were all floating-point numbers to make all the calculations easier to do. There were seconds set to minutes * 60, minutes set to hours * 60 and hours set to days * 24.
- iv. A scanf was then used to save your input into a variable called age (previously set to 0).
- v. The age variable was then multiplied with the daysInAYear, then multiplied by hoursInADay, then multiplied with minutesInAnHour and then finally multiplied by secondsInAMinute.
- vi. Finally, the answer was output as a floating number but with a slight prefix change (%.0f) so that it would show a full number but still be useable for floating point calculations later.

7. 09/03/2017: Lecture – 5: WG403

a. **Lessons learnt:**

i. Characters – Or char

1. The char data type stores whole numbers. Unsigned char saves 0-255
2. A char variable can be assigned whole number literal values.
 - a. char example = 65;
3. Char variables can also be assigned to using character literal values.
 - a. char example = 'A';
 - i. A is a number and it resolves itself as a number from ASCII
4. Examples: 'a', 'b', 'A', '4', '&'
5. Character literals are single characters enclosed in single quotation marks ''

b. **ASCII – Character encoding scheme.**

i. Originally based on the English alphabet. There are 128 specified characters assigned into 7 bit integers.

'A' = 65
'B' = 66
'Z' = 90

- ii. An interesting point, is that textual numbers and textual characters are different from their integer values. When printing a textual version of the number 0. The value of a textual 0 is equal to 48, while the integer value of 0 is still 0. Like the textual version of A is equal to 65, while the integer value of A in hexadecimal is equal to 10.
- iii. UNICODE is the upgraded version of ASCII. It was created to fit all the thousands of other characters across all other languages. Such as emoji created by the Japanese to describe certain emotions, all the way to Arabic language character formatting.
 1. Some examples of non-printable characters in ASCII and UNICODE are NULL and NEWLINE. These characters are actual characters represented as integer values.

Decimal	Hex	Character	C char literal
0	0	NULL	\0
10	A	New Line	\n

2. The first printable character has a value of 32 and is a space ' '
3. 65 is the first printable letter = 'A'

```
char example;
example = 'A';
printf("%c", example);
```

- iv. The code above prints A because it's searching for char (%c) and not the value (%d). If you were to set the example variable to 'A' like above, and then print the (%d) value of the char, it would come out as 65. You can save a char as a number then print it out as a %c and it'll translate the number into an ASCII character.

v. Text characters saved into a char must have the '' (single quotation marks) around it.

c. **Lessons Learnt:**

i. Character Arithmetic:

1. Since the char variable type simply stores numbers in ASCII values, you can easily convert back and forth between number values and the characters using %c and %d. Because of this, you can do simple arithmetic with the char values.
2. If you want to convert from lower case letters to uppercase letters, you simply need to shift forward or backwards through the ASCII values. All the characters in the alphabet are 32 values away from their lowercase or uppercase counterparts. Uppercase 'A' is 65, while the lowercase 'a' is 97. If you want to go from uppercase to lowercase, simply add 32, while going lowercase to uppercase means subtracting 32.

```
char x = 'a';
x++;
```

3. The code above simply adds 1 to the char value of 'a', so the lowercase letter a, becomes the lowercase letter b

```
x = x-32
```

4. While the code above shifts the character a held in the char variable x back upwards through the ASCII list to the uppercase version of it. So 'b' is shifted back 32 values and goes from 98 to 66.

d. **Lessons Learnt: Standard input from keyboard**

i. scanf = scan formatted. Programmers call this pause for waiting for the input a parse. Or parsing.

- ii. Stream buffering:
 - 1. When using scanf, input from the keyboard is buffered into the memory. Buffering means data from the input is stored in the temp memory and held before the program can access it.
- iii. Scanf and the %c format specifier:
 - 1. %c scanf will only consume a single byte from the buffer. So, if you input a word, such as 'hello' the buffer will contain the entire hello, but only one letter will be held in the ram itself until it is moved forward again using another scanf to scan in the next piece of information.
 - 2. With printf("%c") it prints the first character you inputted then goes along the characters.
- iv. You must RESCAN the information in to get the next character.


```
scanf ("%c", &input); -----> puts the whole text in buffer
printf ("%c", input); -----> prints out the first byte you typed
scanf ("%c", &input); ----> doesn't stop the user. Scans next letter you had inputted.
printf ("%c", input); ---> prints the next byte that was just scanned in.
```

 - 1. The buffer is still there. Letters just have to be scanned in again.

```
scanf ("%c", &c);
```

 - 2. The white space in the above code before %c removes all white space.
 - 3. This means it skips the enter you must input to input data and all spaces from the scanf. This is EXTREMELY important and needs to be there to scan in the next byte of printable text instead of a new line. If you don't put this here, it enters in the new line and when you print out the bytes that you just inputted previously, it'll print out the new line instead of a word.

e. Lessons Learnt: Security issues with scanf

- i. #define _CRT_SECURE_NO_WARNINGS
- ii. (just remember #define and the option above will pop up.)
scanf ("%d/%d/%d", &day, &month, &year);
- iii. The code above expects the exact text and numerical values of number/number/number with the slashes included. The scanf command can parse multiple things. If the code above gets unexpected data, it won't work.

f. Lessons Learnt: scanning options for scanf.

- ```
int count = printf ("_____");
```
- i. The code above counts all the characters including the variables within. So, if you have a scanf above that scans in a number. Depending on that variable, the int count might be thousands of characters long or only a few.
- ```
int debug = scanf ("%d/%d/%d", &day, &month, &year);
```
- ii. The debug tests how many variables were scanned in. In the example above, there should have been three variables scanned in. The debug should = 3
 - iii. If scanf holds 3, then it's good, if not then it doesn't work.
 - iv. If you didn't put in a slash like the scanf expects, it'll stop and only scan in one number. Debug will only save 1 character being scanned in. You can use the value of debug later for other things.

8. 09/03/2017: Self Study: AUT Library:

a. **Program Descriptions:**

- i. I went over all the remaining exercises for the week 2 labs.
- ii. The exercises ranged from simplistic mathematics to averages and modulus's. I had little to no trouble programming them and the errors encountered were only due to mishaps or simple mistakes in the code such as putting double quotation marks around a single ASCII character which caused the byte to be read into the variable wrong.

b. **Major Implementation issues:**

- i. Getting white space to a proper standard took a little bit of time to turn into a habit, since I was so used to many other coding standards and languages. But afterwards, it was easy to implement simplistic arithmetic.

c. **Lessons Learnt:**

- i. An important piece of info to gather from my study is that whilst not fully understanding it in the labs or the lectures, a float requires a decimal place along with a 0 followed by an f when initialising the variable. If it isn't placed, the IDE assumes the variable is a double and not a floating-point number.
- ii. Going over the previous week of data:

```
scanf ("%f", &a);
```

- iii. This code means that the scan will take your input and save it directly into the *address* of the variable. Originally, I just assumed that it would be easier to store data straight into the variable, but with some bugs, I very quickly learnt that using the & was needed.

9. 10/03/2017: Lecture – 6: WG403

a. **Lessons: Arrays (declaring, manipulating elements, initialising), mathematical functions (using: #include <math.h>), Random numbers (using: #include <stdlib.h> and #include <time.h>):**

- i. Declaring lots of data is very, very inefficient. If you want to declare a lot of numbers, usually, using int is an option, but declaring lots and lots of ints is a bad idea.
- ii. Arrays are a way to hold a lot of data while still being able to operate under normal circumstances like arithmetic and able to follow rules bound to the entire array.

```
int city_ages[360];
```

```
int south_ages[64];
```

← This is how you declare an array. The numbers inside the square brackets dictate the size of the arrays. 360 means there are 360 values inside the city_ages array.

1. [type of data] [name of variable][number of elements inside the array]
2. In the debug watch window, you can open the tree structure of the variable and see each individual element.
3. To save into an array, it's the same as normal variables, but you must set the index.

```
south_ages[0] = 18;
```

- a. The code above sets the index 0 of the south ages array to 18. This is now one index out of 360 filled with a number. The first index is always 0 and the last index is always the number of elements – 1. This is because in programming, you count from 0.

```
printf("index 0: %d\n", south_ages[0]);
```

- b. The code above prints index 0: 18. This is because, above, we set the index 0 of south_ages to 18 and now we're printing the same index.

- iii. If we know exactly what we want the data to be, we can set each element using the initialising syntax:

```
int data[8] = {5, 10, 7, 1, -3, 5, -7, 7};
```

1. It's set as type = int, and the data variable has an element size of 8 with each index filled with a number set inside {} braces.
2. You can also do arithmetic with each index in the array.

```
--= += *= /=
```

- iv. In debugger, you can also look at the memory:

1. Debug >> windows >> memory >> different places (selection menu)
2. You can then type in the variable or the variable address. (hover over the variable, then you can right click and see address). You can set it to view by 4-byte integer or by signed integers.

The screenshot shows a debugger interface with two main panes. The top pane, titled 'Memory 1', displays a memory dump starting at address 0x00CFFC10. It shows several rows of memory addresses and their corresponding hexadecimal values. The bottom pane, titled 'Source.c', shows the source code of a C program. The code includes <stdio.h> and defines a main function. Inside main, it declares an integer variable 'a' and a character variable 'b', both initialized to 0. It then uses printf to print the values of 'a' and 'b' on separate lines, and finally returns 0.

b. **Lessons Learnt: Array terminology:**

- i. Array = Block of memory
- ii. Element = single variable in the array
- iii. Dimension/size = number of elements the array can store.
- iv. Name = name of the array.

c. **Lessons Learnt: Errors:**

- i. Remember array elements are indexed:
 1. They start at index = 0. The first element in the array.
 2. They end in the index of (dimension - 1). The last element in the array.

- ii. Beware: C allows you to access numbers beyond the end of the array.
- iii. You cannot input an array dimension if you're initialising. The code will set the dimension to the amount of numbers within the initialisation.

```
int data[] = {400, 540, 210, 120};
int index = 0;
```

```
printf("data[%d] holds %d\n", index, data[index++]);
```

- iv. The code above sets an array with 400, 540, 210 and 120 and then sets another variable called index as 0. (You can print out an array with an index of a variable. And you can then edit that variable and do arithmetic to it, to move the index to different positions. The last line of code above prints out 'data[n]' with an index of the variable 'index' and then adds 1 to the 'index' variable so that next print, the index will have moved along by one. It started at 0. So, the print will show that, data[0] holds 400. The next print will show: data[1] holds 540.
- v. Bugs: Array index is out of bounds.

```
int array[5];
array[5] = 0;
```

- 1. The bug is the last line of code. If the array has a dimension of 5, this means the index's go from 0, 1, 2, 3, 4. The index of 5 doesn't exist in the context.

d. **Lessons Learnt: Using: #include <math.h>**

- i. The different codes:

```
float a = 0.0f;
```

```
a = powf (2,3); -----> 23
```

```
a = sqrtf(16.0f); -----> √16.0
```

```
a = expf (1); -----> e1
```

```
a = logf (2.7); -----> Ln (2.7)
```

```
a = log10f (1000); ----> log101000
```

```
a = fabsf (-5); -----> |-5|
```

```
a = floor(3.75f); -----> computes floor of 3.75f
```

```
a = ceilf(3.75f); -----> computes floor of 3.75
```

- ii. Also, you can use sinf, cosf, and tanf.

e. **Lessons Learnt: Random Numbers. Generating them:**

- i. PRNG = Pseudo Random Number Generator.
- ii. Random number generators must be 'seeded' before they can be used to gen a 'unique' sequence of numbers.
- iii. Call srand() first. Only needed once at the start of a program.
#include <stdlib.h>
- iv. Pass in the result from time() function
#include <time.h>

EXAMPLE:

```
srand(time(0)); ←----- seed only needs to be set ONCE
```

```
rand(); function ←---- Rand returns an integer between 0 and around 2 billion.
```

EXAMPLE:

```
int dice_roll = (rand() % 6) + 1
```

The (rand() % 6) part gets a random number from 0 to around 2 billion and then divides by 6 and gets the remainder. The remainder will be between 0 and 5. To shift the numbers up to a 6-sided die, we need to add 1. (the +1 part)

10. 10/03/2017: Self-Study: Home:a. **Program Descriptions:**

- i. For this days' worth of study, I really got excited at the idea of random number generators whilst in the lecture a few hours prior and wanted to do some testing.
- ii. I ended up making a small code that got the input from a user and then generated two random numbers and used the % statement with the variable that the user inputted so it generated two random numbers between 0 and the input.
- iii. Then it takes the two variables and does some simple math with them.

b. **Bugs:**

- i. No known bugs besides not stating the variables or setting the variables wrong. Also, when using the math commands, I sometimes forgot to `#include <math.h>` which resulted in the `powf` or another command to single as undefined.

c. **Lessons Learnt:**

- i. This self-study period, I read up for about an hour in the coding book to gather knowledge on more complex code since the things we learnt this week weren't challenging enough for me. After reading slightly ahead, I acquiesce myself with more complex coding such as arrays c-strings.
- ii. You can't generate a char array and initialise it's dimension with a variable. Which is a bit annoying. I tried generating a random number between 1 and 1000 and then setting the dimension of an array to that number then generating a new random number and saving it into a random index with the array. But it didn't work unfortunately.
- iii. Instead, I just made an array with 1000 dimension and did the other stuff as shown below.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

int main(void)
{
    srand(time(0));
    char test[1000];

    int index = 0;
    char random = '\0';

    random = (rand() % 255) + 33;
    index = rand() % 1000;

    test[index] = random;

    printf("%c\n\n", test[index]);

    return 0;
}
```

11. 12/03/2017: Lecture – 7: WG403:

a. **Lessons Learnt: Character Arrays, declaring them, initialising them, printing char arrays, null characters, c-string, using scanf and character arrays, array dimension vs c-string.**

- i. Applications of arrays:
- ii. The applications of arrays are extremely varied and wide such as holding many different forms of characters. From c-strings to multiple numbers.
- iii. A c-string can hold ASCII values.
- iv. Each element stores 1 byte of text. This is to say, if an array had a dimension of 20, this means there are a total of 20 elements. Counting from 0 to 19, each element can hold one byte or one character of text.
- v. When printing a c-string, you must use a specific printing identifier.
 1. To clarify:
 - %d and %i, are int values.
 - %f is a floating-point number
 - %c is a char value
 - %s is a string value.
 - %x is a hexadecimal value.
- vi. When printing a c-string, it will print every single ASCII character that exists within an array. It will only stop when it gets to a NULL character.

```
Printf ("%s", my_name);
```

- vii. The code above prints out the entire array of the variable "my_name". That's index 0 – whatever and stops at the NULL character. If it doesn't find one that you set, then it'll print beyond the array and into the spare ram bytes.
- viii. You can set the final byte of the char array to a NULL to fix this problem. However, if you're scanning in using a character literal with single quotation marks, then the NULL is needed. But, if you're using double quotation marks, it means it's scanning a string literally which automatically places a NULL character at the end of the string.

b. **Lessons Learnt: Array Initialisation:**

- i. When initialising arrays, you do not have to set the dimension of the array. It will simply look for all string literals or character literals within the compound structure and make the dimension the amount of characters that are being initiated. This is only really used when you know exactly what you want inside the array.

```
Char greeting[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
Char greeting[6] = {"hello"};
```

- ii. The two lines of code above do the exact same thing. The top one uses the character literals to save into the array, whilst the bottom code saves a string literal into the variable.
- iii. To print it out, you use the %s to print out all data within the array.

c. **Reading in text:**

- i. Scanf and %s need to scan into an array. They can't scan into their own form of variable. Like a string variable for example.
- ii. When scanning with an array using string literals, you don't need to search for the address in the scan.
- iii. Using %[number]s means it only scans in the numbers worth of characters or numbers into the array. This is also stops at the white space, so a space (' ') is a stopping point for the scan.
- iv. You can also block input from the user so that the user can only input certain information:

```
Scanf ("%[A-Z]s", &upper_case_only);
```

- v. To scan in multiple things and be specific with what you're blocking, you can simply put [A-Za-z0-9].
- vi. No need for a comma or ; to break up the syntax.

d. **Using c-string commands with string and string arrays:**

- i. There's a lot of different commands that allow the user to test and compare different things using the string commands.
- ii. The commands only work with string literals. You can't use these with character literals or integers or floating point numbers.
 1. strlen ←----- This command gets the string length of an array. It doesn't count the NULL character, so it is usually 1 less than the total array dimension.
 2. strcpy ←----- This command gets a copy of one string literal and copies it into another string array. The syntax is as so:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>

int main(void)
{
    char test[10];
    char test2[] = { "lols, kek" };

    strcpy(test, test2);

    printf("%s", test2);

    return 0;
}

```

3. `strcat` ←----- This is string concatenate. Which essentially means combine one string array to another. This does not move one from the other, just puts them together in a combined string. This can be used for putting a full name together for example. This also copies the NULL. A nice feature with string concatenate is you can combine a variable array to a string literal. Such as `strcat(testing, "yes, testing");`

e. **Converting textual c-string to int values:**

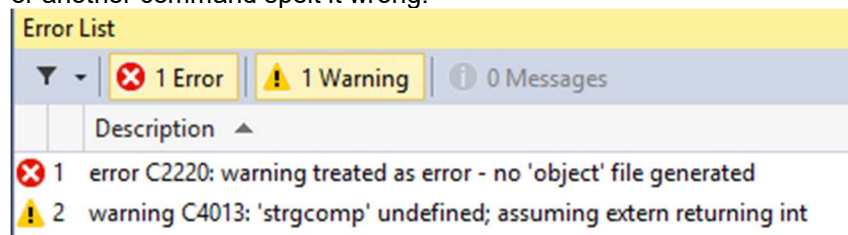
- i. `atoi()`
- ii. This stands for ASCII to integer values and basically does exactly that. Turns ASCII or c-strings directly into integer values of themselves.

```
number = atoi(digits);
```

- iii. The code above moves the digit's value with ASCII values into an integer variable.
- iv. `atof` also works for converting ASCII to floating numbers.

f. **Bugs**

- i. There was only one bug I came across which was when I forgot how to syntax the string copy or another command spelt it wrong:



12. 12/03/2017: Lab – 3: WG403:

a. **Program Descriptions:**

- i. In the lab, we all went through the exercises and did more simple arithmetic to convert different types of variables to different forms of number.
- ii. Float to ASCII, used atoi, temperature convertors, change convertors and such.\

b. **Lessons Learnt:**

- i. I did not really understand how modulus worked and how it got the final answer. So, after some learning and talking to the lab lecturer, I found that it *truncates* the input given to a certain extent. This allowed me to properly do some of the exercises such as change convertor. Or cents to dollars. Bellow, I converted any amount of time you gave it in seconds into higher levels of time. Because I had already read quite far ahead in the coding with C book, I already knew how to do if's and while, but I decided to keep it simple and simply print these variables out. The variables were calculated when going down a level by using modulus to truncate the top of the time off. I managed to get milliseconds by going `milliseconds = ((input % 60) % 60) % 1000`.
- ii. To get the seconds, I just truncated it twice and to get minutes, I used a combination of dividing by 60 and modulus of 60.
- iii. The signed long, long means I can extend my range of input in seconds to 18,446,744,073,709,551,615 allowing the user to input incredibly high numbers. Or 18 quintillion.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>

int main(void)
{
    signed long long milliseconds = 0;
    signed long long seconds = 0;
    signed long long mins = 0;
    signed long long hours = 0;
    signed long long days = 0;
    signed long long years = 0;
    signed long long decades = 0;
    signed long long century = 0;
    signed long long millennium = 0;
    signed long long eon = 0;
    signed long long input = 0;

    printf("Please input the time you want in seconds > ");
    scanf("%lld", &input);
```

c. **Bugs and Debugging:**

- i. The only problems I had with the programs when going through the exercises was when I didn't place the modules or the dividing calculation beside each other. I got some wrap around issues where numbers would sometimes be too high and the calculation would overflow. Resulting negative numbers being outputted.

13. 12/03/2017: Self-Study – 3: WG403:

a. **Program Description:**

- i. A continuation of the coding I did in the lab.
- ii. I added a lot of extra code to do a lot of different things. I wanted to limit each variable to a readable amount. So instead of having billions of days, or thousands of hours, I made a bunch of while loops that did some basic calculations and moved all the data upwards in measurements

```

while (hours >= 24)
{
    while (hours >= 8760)
    {
        while (hours >= 87600)
        {
            while (hours >= 876000)
            {
                while (hours >= 8760000)
                {
                    while (hours >= 8760000000000)
                    {
                        eon += 1;
                        hours -= 8760000000000;
                    }
                    millennium += 1;
                    hours -= 8760000;
                }
                century += 1;
                hours -= 876000;
            }
            decades += 1;
            hours -= 87600;
        }
        years += 1;
        hours -= 8760;
    }
    days += 1;
    hours -= 24;
}

```

- iii. The code above makes sure the hour's variable is limited to only displaying the number of hours within a day. No more. It will never display 26 hours, or a million hours. It will truncate the hours upwards through the measurement system until you get to eons. So, if a big enough number in seconds is inputted into the code, such as 9,223,372,036,854,775,807 (the largest possible number for a long long variable). Then the code will move the time upwards until it all sits inside the eons variable. This same loop is also done for each of the other time measurements.
 - iv. I did this more so for fun and to practice at using while loops, for loops, if's and variables
- b. **Bugs:**
- i. I had quite a lot of bugs from this as you can tell. I had wrap around bugs at first because I was only using an int variable. So, anything above 2 billion overflowed the number's and resulted in negative time. After switching to long variables, I found that wasn't enough to get as high as I would like. So, after spending some time using google, I found how to get variables that are bigger (Lorenzo Donati, 2013). A long long can hold up to a 64-bit number, which is up to: 9,223,372,036,854,775,807. Apparently, there is also a 128-bit number that is possible to use as well using `int128_t` but that apparently doesn't work with older IDE's.
 - ii. Other bugs later included having whiles within whiles sometimes causing an overflow because I forgotten to change a single variable when I copied it downwards.
 - iii. Also, I tried using an if statement that was nested causing the whiles to take much longer to go through. Causing the program to move very slowly.
 - iv. Using a `singed long`, `long`, I tried to extend the amount of numbers that could fit inside the variable, technically, this would push the limit to 18,446,744,073,709,551,615. However, for some reason, using a variable like this caused the program to massively slow down and not even run. After finally figuring that out and changing it to a normal `long`, `long`, the program ran instantly.
 - v. I tried adding code that would separate each of the 3 decimal numbers and then decide whether they were a group of a hundred million, billion, trillion and so on then converting the numbers into ASCII strings. However, this didn't work due to an error C2220: no 'object' file generated:

	Description
❌ 1	error C2220: warning treated as error - no 'object' file generated
⚠️ 2	warning C4013: 'lltoa' undefined; assuming extern returning int
⚠️ 3	warning C4047: 'function' : 'const char *' differs in levels of indirection from 'char'
⚠️ 4	warning C4024: 'strcat' : different types for formal and actual parameter 2
⚠️ 5	warning C4047: 'function' : 'char *' differs in levels of indirection from 'char'

```

if (totalOutPutYears >= 100000000) //hundred million
{
    long long mill = (((((totalOutPutYears / 100000) % 10) % 10) % 10) % 10); //devided by thousand
    singleWord[3] = '\0';
    lltoa(mill, singleWord, 10);
    strcat(singleWord, words[10000]);
    strcat(words[10000], " million, ");
}

if (totalOutPutYears >= 100000) //hundred thousand
{
    long long thou = (((((totalOutPutYears / 100) % 10) % 10) % 10) % 10) % 10; //devided by hundred
    singleWord[3] = '\0';
    lltoa(thou, singleWord, 10);
    strcat(singleWord, words[10000]);
    strcat(words[10000], " thousand, ");
}

if (totalOutPutYears >= 100) //hundred
{
    long long hund = ((((((totalOutPutYears) % 10) % 10) % 10) % 10) % 10) % 10; //devided by hundred
    singleWord[3] = '\0';
    lltoa(hund, singleWord, 10);
    strcat(singleWord, words[10000]);
    strcat(words[10000], " hundred");
}

```

- vi. After this error came about, and I didn't know how to fix it, I decided to abandon that idea and just go with what I had before which was working. The final product could take in any value of seconds up to 9,223,372,036,854,775,807 and convert it to the proper unit(s) of time:

```

C:\Windows\system32\cmd.exe
Please input the time you want in seconds > 9223372036854775807

2 Times the total length of the universe when using the big freeze theory.
For refrence: total age of the universe is ~ 100 trillion +
96579 Eons
251 Millennium
3 Centuries
0 Decades
2 years
25 days
15 hours
30 minutes
7 seconds
7 milliseconds

Press any key to continue . . .

```

14. 16/03/2017: Lecture – 8: WG403:

a. **Lessons Learnt: Coding Standards, Commenting, Designing, Implanting and testing.**

- i. Coding standards are guidelines that recommend certain programming style or practice
 - White Space
 - Indentation
 - Comments
 - Declaration
 - Variable Naming
 - General Good practice rules
- ii. The benefits of high quality code:
 1. Maintaining Source code over time
 2. Your peers or fellow programmers in a company can see exactly what is going on.
 3. You can look back yourself and see exactly what you meant by a bit of code.
- iii. Good Naming
 1. A good way to name variables is either by separating the words with an underscore such as snake_name. Or, you can spell out the words with no spaces or underscore but each word starts with a capital letter (snakeName).
 2. Naming variables with very similar names or the same name but with small changes is really bad practice. Using names like i, l, l, L and 1 are bad because they're all very similar. You can't even tell written on this document right now. The first letter is lowercase i, the second letter is upper case i, the third is lowercase L and the fourth is uppercase L. It can get very complicated.
 3. It is important to also not use names that only have small changes. Such as using the name age, but different spelt or arranged such as:
 - o age
 - o AGE
 - o Age
 - o AgE
 - o aGe.

b. **Lessons learnt: Commenting:**

- i. Comment means to put human readable language or text beside a piece of code so that later, you, the marker, your boss, or your fellow programmers at a company can see exactly what you did and how you did it.
- ii. It is good to be relatively detailed in your comment, but stay at a form like pseudo code

```
signed long long milliseconds = 0;    //sets variable for milliseconds
signed long long seconds = 0;        //sets variable for seconds
signed long long mins = 0;           //minutes
signed long long hours = 0;          //hours
signed long long days = 0;           //days
signed long long years = 0;          //years
signed long long decades = 0;        //decades
signed long long century = 0;        //centuries
signed long long millennium = 0;     //millennium
signed long long eon = 0;            //aeons (eons)
signed long long timesTheUniverse = 0; //This variable dictates the time of the universe * n. If the time extends over a certain amount, then the total time will be equal
signed long long input = 0;          // 1 * the whole age of the universe+ all the other time that is recorded. So a very, VERY long time.
```

- iii. There are two types of commenting: You can either use the style in the code above using double slash. That is an easier way to comment as you know only the things in front of the double slash will be commented. But you can use the /* */ to comment as well. This style of commenting ignores the lines that it exists on and starts commenting from the first opening slash * and only stops at the next * slash. Using this form, you can comment out entire sections of code or entire pages.
- iv. You can't nest comments using the second method. And the first method doesn't have a closing syntax command. So, you can comment in a double slash.

c. **Lessons Learnt: Pseudo Code**

- i. Pseudo code is informal, high-level description of a process or algorithm.
- ii. It is intended to be read easily by a human and can be understood easily.
- iii. There is no syntax needed for pseudo code.

15. 17/03/2017: Lecture – 9: WG403:

a. **Lessons Learnt: Selection using decisions. If and else and else if.**

i. In pseudo code, an if else decision would be written as:

- IF
- If "stuff is true"
- Then "do x"
- END IF

- Else "perform y"
- END ELSE

ii. An if works by checking simple arithmetic inside the () braces. and if it's true then it will run the compound code running inside the {} braces.

iii. *If stuff inside is equal to something = do something.*

iv. If there is no basic logic, and only a variable or a single letter, then that will always equal to true. True is represented as a 1 and false is represented as a 0. Off and on like binary. ANYTHING none 0; negative numbers, positive numbers and floating numbers all equal to true.

v. The basic operations are easy:

- A == B (This checks does A equal to B?)
- A != B (This checks if A is NOT equal to B?)
- A > B (This checks if A is larger than B?)
- A >= B (This checks if A is larger or equal to B?)
- A <= B (This checks if A is larger or equal to B. And the same can be said for just being larger and not equal.)
- A && B (This checks if both are true. Or both are non-zero.)
- A || B (This checks if one or the either or both are equal to non-zero.)
- A ^^ B (This checks if ONLY one or the either is equal to non-zero. If both are equal to non-zero, it sees it as false. This is XOR.)

vi. Else means exclusively an ifelse to the original question. Instead of writing out an else if and looking for all other possible answers, we can just use else.

```
If (yes)
{
}
Else <----- This is technically like saying if yes is not true or is not equal to non-zero. So, false.
{
}
}
```

vii. Else just means if the original 'if' is not true, then do the commands for else.

```
printf("> ");
scanf("%d", &x);

if (!(x + 2) == 10);
{
    printf("%d + 2 is not equal to 10\n", x);
    if (x % 2 == 0)
    {
        printf("number is even\n");
    }
    else
    {
        printf("Number is odd\n");
    }
}
```

C:\Windows\system32\cmd.exe

```
> 799
799 + 2 is not equal to 10
Number is odd
Press any key to continue . . .
```

b. **Lessons learnt: String Comparing:**

- i. Strcmp.
- ii. You can use strcmp to compare to different arrays of string characters. To use this, and the other string commands, you must #include <string.h>
- iii. You give the strcmp two strings and if they're the same. It gives an output of 0 to represent it's true. Which is the opposite of the true and false operators with 'if' and 'else'.
- iv. So, you can use this within an if. If (strcmp (name, word) == 0). This code basically means, check the two different strings (name and word) and if they equal, output 0. If that zero is equal to zero. Then the if becomes true. If name and word do not equal zero and thus equal a non-zero (or false), then it will not equate to zero using logic operations meaning the if is false.

c. **Bugs:**

- i. There are a lot of common mistakes that people make when coding. One of them includes when saving to a variable. You must know if you're using a string literal or a character literal. If it's a string literal, you must use double quotation marks, and if it's a character literal, then you must use single quotation marks.

16. 17/03/2017: Self-Study: Home:

a. **Programing Descriptions:**

- i. I kept practicing using all the previous week's work of lab exercises and worked on some more complex self-set challenges.
- ii. The first was to create a code that took a variable number of cents and converted it into dollars + cents.
- iii. The second was to create a program that took the number of videos on you tube that you have liked, gets the average video length from a constant variable and then calculates the average watch time that you have spent on YouTube +- a few hundred hours.
- iv. The third and final code that I set out to try and create was a basic guessing AI. It took me more than 2 hours to get just the basic parameters set up and I didn't get very far. After spending so long creating the incredibly complicated code using only whiles, ifs, and else's and variables and inputs and getting confused on how to compare different types of variables, I gave up for the day.

b. **Bugs:**

- i. The most common bugs I found where the wrap around bug when trying to code with char values. When I tried using a system to go from lowercase to uppercase, I found I had problems with wrap around. And when I tried generating random numbers for my code, I found that it had a lot of problems with char values. Eventually, I went about simplifying my code and not coding to such a complicated degree.

```

srand(time(0));

char yes[] = {"yes"};
char no[] = {"no"};
char yesno[10];
char input = '\0';
char a = 'a';
char b = 'b';
char c = 'c';
char d = 'c';

start:
printf("This is a guessing machine. You think of something and it'll guess what it is. An object, animal, or whatever");
printf("\nThink of something, anything. When you are ready, type yes. If you're not ready, type no > ");
scanf("%s", &yesno);
for (int i = 0; i < 40; i++)
{
    printf("\n");
}
yesno:
if (strcmp(yesno, yes) == 0)
{
    goto begin;
}
else if (strcmp(yesno, no) == 0)
{
    goto start;
}
else
{
    printf("Please enter a yes or a no > ");
    scanf("%s", &yesno);
    goto yesno;
}

```

- ii. My code for my 'AI' Guessing game was meant to try and get answer for anything you thought of. But after some thinking and google, I found that most guessing AI, like the black ball, or the Radica 20Q Artificial Intelligence Game (<http://www.20q.net/> 1988 - 2015).

17. 20/03/2017: Lecture – 10: WG403:

a. **Lessons Learnt: Logical NOT, OR, AND. Going over else if briefly.**

- i. Else if crushes down a nested if essentially. It can replace an if nest because it acts like a second if but only happens if the first if is not correct and so on. Down a line of else if.
- ii. If there is no else, it doesn't matter. If there is nothing else to do, there is no need for else.

b. **Lessons Learnt: Boolean OPERATORS:**

- i. There are a lot of Boolean logical operators.
- ii. And to be completely honest, *breaking the fourth wall here. I already knew a LOT about Boolean operators from Minecraft. Using Redstone which is either off, or on. I created a Boolean logic calculator within Minecraft where each bit change to 1/10th of a second. It was very slow, but worked very well.*

AND	&&
OR	
NOT	!
XOR	^^

- iii. AND means that both operations must be true for the if to run. If 1 or neither are true, then it equals a false.
- iv. OR means that either 1 or the other or both operators or operations must be true for the if to run. If neither is true, only then, is it false.
- v. NOT means it inverts the output of an answer. If you set a variable such as a with it equalling to the NOT of b + c, then it will equal the opposite of b+c. If b+c is a non-zero, then b+c is equal to true, meaning the NOT will invert that and change it to a false. If b+c is equal to zero, then it means it is false and the NOT will invert that and change it to true.
- vi. XOR means that either 1 or the other of the operations outputs a true. If NEITHER or BOTH are true, then it outputs a false. So only one of the operations can be true for the if to run.

c. **Bugs/Debugging:**

Edge/boundary cases:

- i. Should the programmer test for either < or <=?" inclusion or exclusion.
- ii. AND and NOT are not the same. AND outputs a true ONLY if both operations are true. NOT inverts whatever operator it's given. If the operator is equal to true, then it means it's false. If the operator is equal to false.
- iii. 'Else if' is different from just 'if'. 'Else If' means that if the above 'if' failed, it will run the next 'if'. But if it's only using an 'if' and not an 'else if' then it should be nested so that it's included within the first 'if' so that it *continues* the operation flow.

18. 20/03/2017: Lab – 4: WT201:a. **Program Descriptions:**

- i. This week's labs were a lot more complicated than the previous and used the if's and else's and all other coding material we learnt from week 1-4.
- ii. The programs were simply going through all the exercises:

b. **Bugs:**

- i. Most of the bugs with the exercises came from not knowing exactly what we wanted to compare or if together. The exercise descriptions were more abstract and left much up to the imagination and knowledge of the programmer. But I still found the coding easy and fun. The bugs came from using string compare to compare characters or to compare integers, and using normal comparing to try and compare to strings. Which resulted in an undefined string compare or similar errors.
- ii. Also, when using greater than, less than or greater and equal to and less than and equal to, I got confused a lot of the time with what the calculation was asking. Especially when coding when programmers count from 0 up.
- iii. This exercise especially tripped me up as I was still confused a little bit by what the question was asking and what happened with overflow and such. I got the comparing wrong a couple of times in certain areas like when finding the greater or equal to of a number, I got confused and was comparing it a number under the value it was supposed to be.

The retail store's discount rewards program:

Purchase Amount:	Discount Percentage:
$x < 2500$	0.00%
$2500 \leq x < 6500$	5.00%
$6500 \leq x \leq 10000$	10.00%
$x > 10000$	12.50%

An example of the program is as follows:

```
Input the total purchase price: 5000
Discount is: 250.00
Payable Total is: 4750.00
```

Another example is as follows:

```
Input the total purchase price: 25234
Discount is: 3154.25
Payable Total is: 22079.75
```

Another example is as follows:

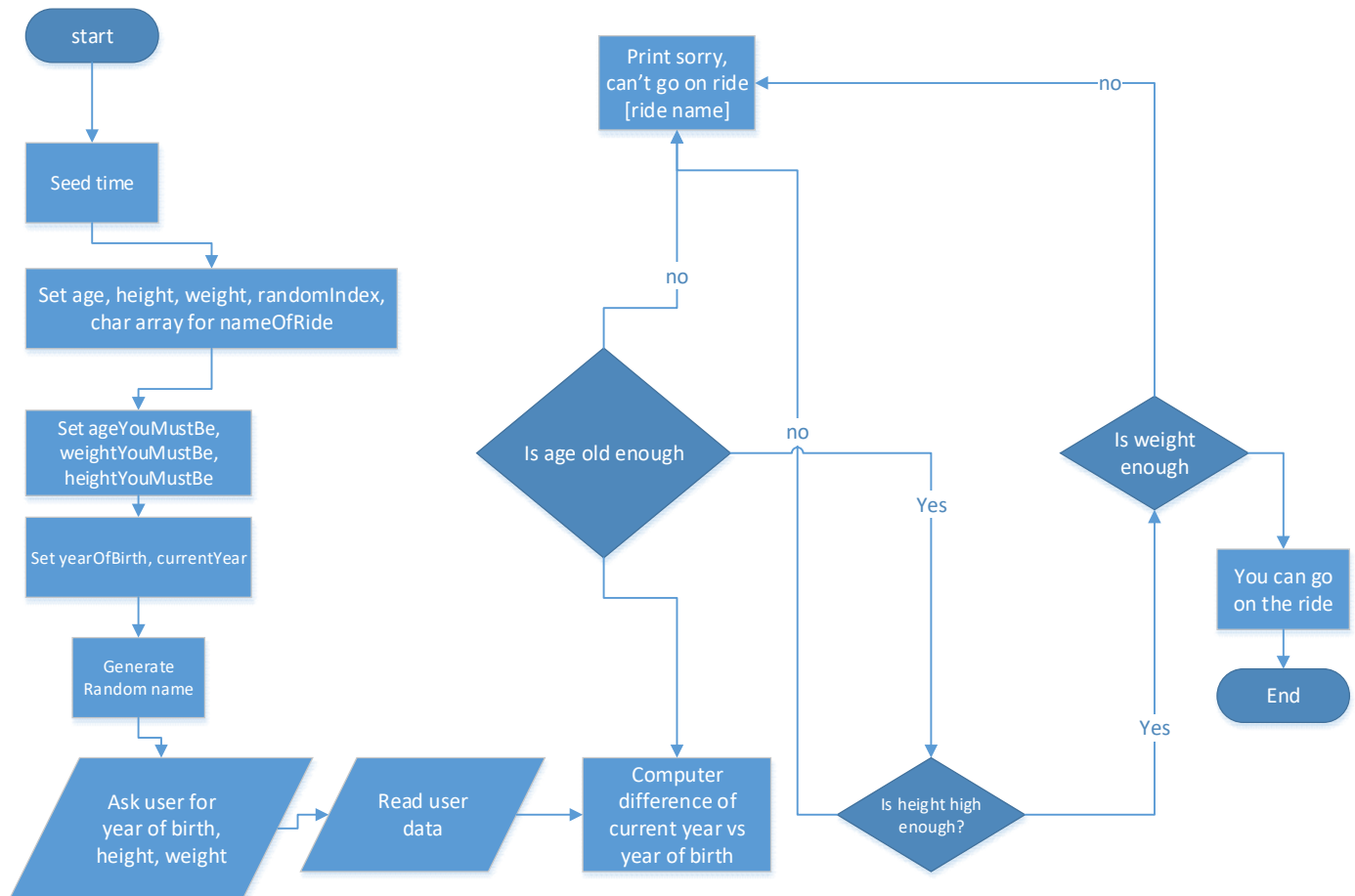
```
Input the total purchase price: 7777
Discount is: 777.70
Payable Total is: 6999.30
```

c. **Program Descriptions – Group activities.:**

- i. The group exercise that me and a friend ended up doing was using constant variables, if's, random numbers and calculations using variables to find out the name, height, weight and age of a user to see if they could go on a slide at a water park.

d. **Descriptions:**

- i. We had very little next to no bugs during our coding, but at that stage, it was quite a simple code. Using the knowledge and practice we had just gotten from the exercises, we quickly made a Microsoft Visio flow chart of what we wanted and started coding:



The code for this was relatively easy, however we only finished up to the random generate name command. The idea was to make an array that was to be the name of the ride, and then make a while loop and generate a random character each time and put each character into each index of the array and increment it forward until every index was filled with a character.

Then it got the name of the user and checked for any bad names like swearing and such. It then got the age, weight and height of the user and checked all those things against what they were supposed to be to enter the ride.

19. 20/03/2017: Self-Study: Home:

a. Program Descriptions:

- i. Continue the code that was being studied during the lab previously that day.

b. Lessons Learnt:

- i. After 2 hours of coding, I was finally happy with the overly complicated code to find out if you can ride the slide or not:
- ii. The first thing the code does is set an array with variables that your age, height, weight and year of birth must pass in order to be allowed to go on the slide. I set it as an array because it was quite busy anyways and using multiple ints resulted in a mess. After that it sets to string arrays with the strings Hitler and Trump. This was a test to see if I could stop people from using names that are 'banned' and thus not allowed to be entered.
- iii. The next thing the code does is set variables that the user will affect. Age, yearOfBirth, height, weight. It then sets a character which is to be the random character to go inside the ride array to set its name.
- iv. The first while uses the variable count and generates a random character, puts it into the ride name with an index of count, iterates it and loops again until the name array is full.
- v. After that, it welcomes the user to the ride and asks for their height, weight and year of birth.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>

int main(void)
{
    srand(time(0));
    const int dataYouMustPass[] = { 13, 48, 130, 2017 };
    const char bannedName1[] = { "Hitler" };
    const char bannedName2[] = { "Trump" };

    int age = 0;
    int yearOfBirth = 0;
    int height = 0;
    int weight = 0;
    char character = '\0';
    int count = 0;
    char nameOfRide[10];
    char firstName[100];
    char lastName[100];

    while (count < 10)
    {
        character = '!' + (rand() % 94);
        nameOfRide[count++] = character;
    }
    nameOfRide[9] = '\0';

    printf("WELCOME TO THE AMAZING %s RIDE!!!\n\n", nameOfRide);
    printf("Please input your Name, your year of birth, your height and your weight > \n");
    printf("Name in format: First Last > ");
    int debugName = scanf(" %s %s", if (!debugName == 2))
    {
        printf("Please Input your full Name. First Last > ");
    }
    else
    {
        printf("\nYear of birth > ");
        int debugBirth = scanf(" %d", &yearOfBirth);
        if (!debugBirth == 1)
        {
            printf("Please input your year of birth > ");
        }
        else
        {
            printf("\nWeight > ");
            int debugWeight = scanf(" %d", &weight);
            if (!debugWeight == 1)
            {
                printf("Please input your weight > ");
            }
            else
            {
                printf("\nHeight > ");
                int debugHeight = scanf(" %d", &height);
                if (!debugHeight == 1)
                {
                    printf("Please input your height > ");
                }
            }
        }
    }
}
```

```
else
{
    printf("\n\nThank you for your info. Please hold while we compute the data...\n");
    age = dataYouMustPass[3] - yearOfBirth;

    if ((strcmp(firstName, bannedName1) == 0) || (strcmp(firstName, bannedName2) == 0) || (strcmp(lastName, bannedName1) == 0) || (strcmp(lastName, bannedName2) == 0))
    {
        printf("%s and %s are Banned names. Thank you for attempting %s ride!!!\n\n", bannedName1, bannedName2, nameOfRide);
    }
    else
    {
        if (!(age >= dataYouMustPass[0] || age < 60))
        {
            printf("%s %s, At age of %d, You are too old/young to go on %s ride\n\n", firstName, lastName, age, nameOfRide);
        }
        else if (!(weight >= dataYouMustPass[1]))
        {
            printf("%s %s, With a height of %d, You are too tall/small to go on %s ride\n\n", firstName, lastName, height, nameOfRide);
        }
        else if (!(height >= dataYouMustPass[2]))
        {
            printf("%s %s, With a weight of %d, You are too fat/thin to go on %s ride\n\n", firstName, lastName, weight, nameOfRide);
        }
        else
        {
            printf("%s %s, You are able to go on %s ride.\n", firstName, lastName, nameOfRide);
            printf("With an age of %d, you are not too old, nor too young.\n", age);
            printf("With a weight of %d, you are not too fat no too thin.\n", weight);
            printf("And with a height of %d, you are not too tall nor too small.\n", height);
            printf("Have a good day!!!!\n\n");
        }
    }
}
```

- vi. It also checks each scan to make sure the user does not put in unreadable or unusable data as well as checks to make sure all the data is scanned in using an int debug on the 'scanf'.
- vii. It then goes through all the variables that the user entered and checks them against the values they are supposed to be to be able to go on the ride and calculates the users age from the user inputted year of birth compared to the current year (2017).
- viii. After checking everything, it will tell you if you can or cannot go on the ride.

20. 23/03/2017: Lecture – 11: WG403:

- a. **Lessons Learnt: Common selection bugs. Switch, Case, Break, Default, continue.**
 - i. Switch is another form of an if but has different uses and different means to achieve something.
 - ii. You can still nest switches but they only check for the validity of something.
- b. **First, Lessons Learnt: BUGS:**
 - i. Common bugs found everywhere and almost every coder seems to do them and find out for themselves what it's like to make a bug.
 - 1. Using a string literal instead of a character literal. C-strings are character arrays. Input for a character literal is supposed to be for a single byte, and thus inside single quotation marks. This can cause a runtime bug.
 - 2. Using double quotation marks around a single character is like saying "y" = y NULL for example. It counts the NULL as a single character along with the y, and thus is seen as two bytes by the computer, not a single byte as it should be with a character literal.
 - 3. Using single quotation marks around a string literal doesn't work either and causes another runtime error.
 - ii. Common mistakes with operators include not using the or operation, or not fully working with the or operation. This means you might go `0 < x < 50` which is not right. You must fully write out the comparison commands like so: `(x < 50 || x > 0)`
- c. **Lessons learnt: Empty compound commands or null statements:**
 - i. The statement ';' is seen as the end of a command. However, using it by itself means that it's a command to do 'nothing' and as such can break some codes if you accidentally leave it in front of a while, or an if. Because if the while command is meant to iterate and move forward inside the loop, then because of the NULL statement, the loop will loop forever and never move onwards, because the question inside the loop is never being changed or altered.
 - ii. The compound statement can also be left along and by itself. But having them empty means a 'nothing step'. If it's inside an if, or a while, the step does nothing.
- d. **Lessons Learnt: Switch, case, break.**
 - i. The Switch checks the value of the expression/variable against a list of case values. The block with the correct case is then executed.
 - ii. Break keyword terminates the case, and the flow then returns to after the switch scope.
 - iii. If break is not used with case, case will execute the fall through and execute the case below as well.

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    scanf("%d", &i);

    switch (i)
    {
        case 0:
            printf("i holds zero");
            break;
        case 1:
            printf("i holds one");
            break;
    }

    return 0;
}
```

- iv. If none of the cases are correct, then it skips all the cases and jumps outside the switch.
- e. **Coding Standards: Switch Fall through:**
 - i. You must comment when and where you wish the code to fall through and continue and where you don't want to it to fall through.


```
Case: 'n' //fall through
Case: 'N'
      Break
```


- ii. In the case of using the if command, it runs of logic and arithmetic. However, if you want to catch everything at the end, when using the if command, you can use else or if else.
- iii. In the case of switches, you must use the command: 'default'.
- f. **Lessons Learnt: Switch syntaxes.**
 - i. Default catches everything else. If none of the cases where correct, it then runs default.
 - ii. There some huge differences between Switches and if. If has more logic based operations. It can check if multiple equation match up or are logical correct. And the switch just checks for input or the value of something.
 - iii. Cases can also have compound structures as well. So, each case can have a switch inside it or other commands.

```
#include <stdio.h>

int main(void)
{
    char key;
    int data;

    scanf("letter %c, amount: %d", &key, &data);

    switch (key)
    {
        case 'r': //fall through
        case 'R':
        {
            switch (data)
            {
                case 1:
                case 2:
                {
                    printf("stuff happened");
                }
            }
        }
    }

    return 0;
}
```

- g. **Lessons Learnt: The last kind of decision making command syntax: Ternary (conditional) operator.**
 - i. It needs three operators and it's all in one line.
 - ii. Syntax:
Condition? value_if_true: value_if_false
 - iii. Example:
Result = a > b? x : y;
 - iv. The code above means that if a is bigger than b, it will save the variable x into the variable Result. But, if b is bigger than a, it will save the variable y into the variable Result.
- h. **Debugger:**
 - i. Conditional Break points. If you need to set a break when something specific happens like if you have a certain bug such as overflow, you can set a break point and set it with basic logic. So, if you have an int, and it becomes a negative due to overflow, then you can set a break point to break the code when that happens.

21. 24/03/2017: Lecture – 12: WG403

a. **Lessons Learnt: Repetition.** At this point, the classes have caught up to my study. I don't know much beyond this besides if loops using labels and the for command:

- i. Repetition
- ii. While
- iii. Validation of the user input
- iv. The do Keyword
- v. The continue keyword.

b. **Standards:**

- i. Actions need repeating a lot, so having a loop command is very useful
- ii. There's so far selection, using decisions with if or switches or ternary operations.
- iii. Then there's Repetition using decision. But the decision dictates if some code is going to run again or not.
- iv. The loop command uses the `while` command and uses the same basic logic as if's, switches and other decision commands.

```
While (count_down > 0)
{
    Stuff;
    --count_down;
}
```

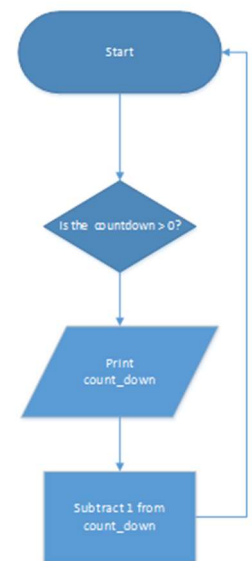
- v. The code above negates one from the `count_down` variable every time it loops so that it goes eventually down to 0 and ends the loop.

c. **Bugs:**

```
int number = 0;
printf("Input a number: ");
int scan_result = 0;
while (0 == scan_result)
{
    scan_result = scanf("%d", &number);
    if (0 == scan_result)
    {
        printf("Please enter a number! Try again: ");
        char bad_input = 0;
        while (bad_input != '\n')
        {
            scanf("%c", &bad_input);
        }
    }
}
printf("Success, the number: %d was entered.\n", number);
```

i. Infinite loops can happen very often. If you set a loop up with a constant variable or an actual character or integer it will always be true and thus, will always loop.

ii. You can also put the NULL statement (;) in front of the loop and that means that it will forever loop a 'nothing' command.

d. **Validation:**

- i. Validation of values, integers and inputs in C language is pretty easy.
- ii. The general rule of thumb when getting information is to NOT pass data onwards as it can break code for a program and sometimes cause millions of dollars in damages when you pass information from a user directly into the code. This happened once with twitter where on one of their spin off websites, they forgot to validate and separate the code for the website from the tweet box. This mean, that any user with knowledge of this, and knowledge of how to code in java could easily hack directly into the website and access information, or create a small script that creates and retweeting tweet. Which is exactly what happened (Scott, 2014)
- iii. You also need to give the user related and useful feedback on what to input as a coding standard.

e. **Validation Between Text and Number:**

- i. `Int scan_result = scanf ("%d", &number)`
- ii. To get rid of any letters or wrong input to the code above, you need to clear out the buffer of any characters like so:

The screenshot shows a C program in a code editor and its execution in a command prompt. The code is as follows:

```
#include <stdio.h>

int main(void)
{
    int input;

    printf("number between 1 - 10 > ");
    scanf("%d", &input);

    while (input < 1 || input > 10)
    {
        printf("Incorrect: ");
        scanf("%d", &input);
    }

    return 0;
}
```

The command prompt output shows the program's execution:

```
C:\Windows\system32\cmd.exe
number between 1 - 10 > -54
Incorrect: 651
Incorrect: 651
Incorrect: 111616
Incorrect: 11
Incorrect: 10
Press any key to continue . . .
```

f. **Do list a sequence when a while is after the commands. Asks the questions after:**

```
Do
{
    Printf ("%d...\n", counter);
    ++counter;
}
While (counter < 5);
```

- i. The code above requires the null statement at the end of the while command because the while comes after the compound code. The do is where the while references to loop around. But in this case, unlike the normal while, it changes the data and prints out stuff first and then asks if it needs to repeat the loop.

g. **Break stops a loop anywhere:**

- i. Using a break inside an if or inside the compound or something, will cause the loop to instantly end, regardless if the question has been satisfied yet.

h. **Continue:**

- i. Continue as a command, restarts the loop anywhere in the loop, regardless if the data has been changed or not. If you put a continue in the wrong place, before data is incremented or changed, then the code will be stuck in permanent loop again.

Bibliography:

Abbott, J. (2017). *Weekly Labs* [Course Exercises and PowerPoint slides]. Learning Basic C. Retrieved from: https://blackboard.aut.ac.nz/webapps/blackboard/content/listContent.jsp?course_id= 85303 1&content_id= 3944184 1&mode=reset

Burgener, R. (1988) *20Q, Twenty Questions*. 20q.net, Wikipedia. Retrieved from: <http://www.20q.net/>
<https://en.wikipedia.org/wiki/20Q>

Donati, L. (2013). *how to represent number bigger than long in c [duplicate]*. Stack Overflow. Retrieved from: <http://stackoverflow.com/questions/18675101/how-to-represent-number-bigger-than-long-in-c>

Hooper, S. (2017). *Weekly Lectures* [PowerPoint slides]. Learning Basic C. Retrieved from: https://blackboard.aut.ac.nz/webapps/blackboard/content/listContent.jsp?course_id= 85303 1&content_id= 3944184 1&mode=reset

Scott, T. (2014). *How the Self-Retweeting Tweet Worked: Cross-Site Scripting (XSS) and Twitter*. YouTube. Retrieved from: <https://www.youtube.com/watch?v=zv0kZKC6GAM>

COMP500 / ENSE501: Week 3 - Peer Feedback and Critique Form

FORMATIVE REVIEW: Reporting Journal, Week 3

Reporting Journal to Review: COMP500/11 Nikkolas Diehl

Peer Reviewer's Name:

Chris Pidgeon

FORMATIVE REVIEW CHECKLIST:

	Tick one:	Yes	No
Is the Nikkolas's Student ID in the header, aligned top-left?			<input checked="" type="checkbox"/>
Does each page have page-number out of total-page-numbers in footer, aligned bottom right?			<input checked="" type="checkbox"/>
Is the first page of the Reporting Journal the Stage 1 tabular timesheet?			<input checked="" type="checkbox"/>
Does the timesheet contain Entry ID, Date, Start Time, Duration, Session Type, and Location?			<input checked="" type="checkbox"/>
Is there evidence of at least 20 hours of learning (weeks 1 and 2)?			<input checked="" type="checkbox"/>
Does the Reporting Journal contain entries that report on lecture sessions?			<input checked="" type="checkbox"/>
Does the Reporting Journal contain entries that report on code examples?			<input checked="" type="checkbox"/>
Does the Reporting Journal contain entries that report on lab tutorial exercises?			<input checked="" type="checkbox"/>
Does the Reporting Journal contain entries that report on self-directed learning?			<input checked="" type="checkbox"/>
Do Reporting Journal entries describe exercises undertaken?			<input checked="" type="checkbox"/>
Do Reporting Journal entries contain design details, such as flowcharts?		<input checked="" type="checkbox"/>	
Do Reporting Journal entries contain implementation details?		<input checked="" type="checkbox"/>	
Do Reporting Journal entries contain testing details?		<input checked="" type="checkbox"/>	
Do Reporting Journal entries contain errors encountered?		<input checked="" type="checkbox"/>	
Do Reporting Journal entries contain debugging details?		<input checked="" type="checkbox"/>	
Does each entry start on a new page, with the Entry ID at the top-left?		<input checked="" type="checkbox"/>	
Does the Reporting Journal contain evidence of learning activities from Week 1?		<input checked="" type="checkbox"/>	
Does the Reporting Journal contain evidence of learning activities from Week 2?		<input checked="" type="checkbox"/>	
Does the Reporting Journal contain evidence of learning activities from Week 3?		<input checked="" type="checkbox"/>	
Does the Reporting Journal follow the formatting style requirements?			<input checked="" type="checkbox"/>
Is the Reporting Journal well structured?		<input checked="" type="checkbox"/>	
Is the Reporting Journal clear and easy to read?		<input checked="" type="checkbox"/>	
Is the Reporting Journal free of spelling errors?		<input checked="" type="checkbox"/>	
Is the Reporting Journal free of grammatical errors?		<input checked="" type="checkbox"/>	
Does the Reporting Journal include personal code samples?		<input checked="" type="checkbox"/>	
Does the Reporting Journal have a bibliography?			<input checked="" type="checkbox"/>
Is the bibliography in the APA 6 th Edition style?			<input checked="" type="checkbox"/>

20/27

Review the Reporting Journal marking criteria on page 7 of the assignment brief...

Based upon the criteria of "Stage 1 – Evidence of learning activities from Weeks 1 to 4", what letter grade would you issue Nikkolas's Reporting Journal?

Peer Grade

B20/27

COMP500 / ENSE501: Week 3 - Peer Feedback and Critique Form

FORMATIVE REVIEW: *Reporting Journal, Week 3*

Reporting Journal to Review: COMP500/11 Nikkolos Diehl

Peer Reviewer's Name: Ryan Humphrey

FORMATIVE REVIEW CHECKLIST:

	Tick one:	Yes	No
Is the Nikkolos's Student ID in the header, aligned top-left?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does each page have page-number out of total-page-numbers in footer, aligned bottom right?		<input type="checkbox"/>	<input type="checkbox"/>
Is the first page of the Reporting Journal the Stage 1 tabular timesheet?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does the timesheet contain Entry ID, Date, Start Time, Duration, Session Type, and Location?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Is there evidence of at least 20 hours of learning (weeks 1 and 2)?		<input type="checkbox"/>	<input checked="" type="checkbox"/>
Does the Reporting Journal contain entries that report on lecture sessions?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does the Reporting Journal contain entries that report on code examples?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does the Reporting Journal contain entries that report on lab tutorial exercises?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does the Reporting Journal contain entries that report on self-directed learning?		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Do Reporting Journal entries describe exercises undertaken?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Do Reporting Journal entries contain design details, such as flowcharts?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Do Reporting Journal entries contain implementation details?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Do Reporting Journal entries contain testing details?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Do Reporting Journal entries contain errors encountered?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Do Reporting Journal entries contain debugging details?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does each entry start on a new page, with the Entry ID at the top-left?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does the Reporting Journal contain evidence of learning activities from Week 1?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does the Reporting Journal contain evidence of learning activities from Week 2?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does the Reporting Journal contain evidence of learning activities from Week 3?		<input type="checkbox"/>	<input checked="" type="checkbox"/>
Does the Reporting Journal follow the formatting style requirements?		<input type="checkbox"/>	<input checked="" type="checkbox"/>
Is the Reporting Journal well structured?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Is the Reporting Journal clear and easy to read?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Is the Reporting Journal free of spelling errors?		<input type="checkbox"/>	<input checked="" type="checkbox"/>
Is the Reporting Journal free of grammatical errors?		<input type="checkbox"/>	<input checked="" type="checkbox"/>
Does the Reporting Journal include personal code samples?		<input checked="" type="checkbox"/>	<input type="checkbox"/>
Does the Reporting Journal have a bibliography?		<input type="checkbox"/>	<input checked="" type="checkbox"/>
Is the bibliography in the APA 6 th Edition style?		<input type="checkbox"/>	<input checked="" type="checkbox"/>

Review the Reporting Journal marking criteria on page 7 of the assignment brief...

	Peer Grade
Based upon the criteria of "Stage 1 – Evidence of learning activities from Weeks 1 to 4", what letter grade would you issue Nikkolos's Reporting Journal?	<u>20/27</u>