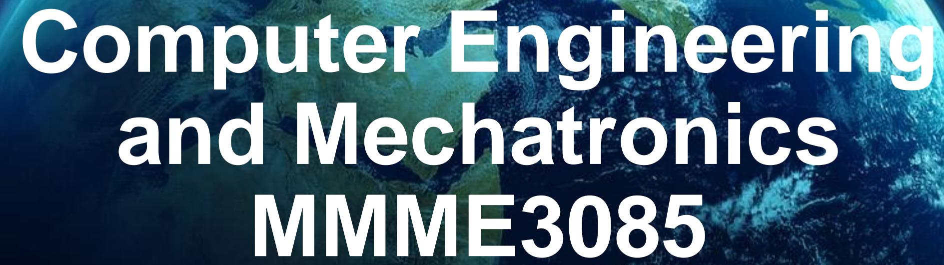
University of Nottingham
UK | CHINA | MALAYSIA



Computer Engineering and Mechatronics MMME3085

Dr Louise Brown

Lecture 1
Programming Recap

The *Famous* “Hello World” Program (1)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                      // Return from prog
}
```

These lines instruct the compiler to ‘include’ the contents of the files ‘stdio.h’ and ‘stdlib.h’

This is done by the compiler **before** the code is compiled

Note: Writing ‘Hello World’ is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* "Hello World" Program (2)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                          // Return from prog
}
```

<u>All</u> C code starts execution at the line

int main()

(regardless of where it is in your code)

Note: Writing 'Hello World' is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* “Hello World” Program (3)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                        // Return from prog
}
```

Brackets {} are used to block lines of code

Note: Writing ‘Hello World’ is a tradition in programming – there are whole web sites dedicated to it, e.g.
http://helloworldcollection.de/

The *Famous* “Hello World” Program (4)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                    // Return from prog
}
```

Comments are vital to good coding – they allow information to be included within code

Between /* and */
or following //

Note: Writing ‘Hello World’ is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* “Hello World” Program (5)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                     // Return from prog
}
```

printf is a <u>function</u> within C that allows us to write text to the display.

In C, the parameters for a function are placed within brackets ( ) , multiple parameters are comma separated.

Note: Writing ‘Hello World’ is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* “Hello World” Program (6)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;
}
```

Each statement in C is terminated with a semicolon ;

Note: Writing ‘Hello World’ is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* “Hello World” Program (7)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                          // Return from prog
}
```

The last statement

return 0;

Terminates the main() function – so ending the program

Note: Writing ‘Hello World’ is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

Chapter 5

Output

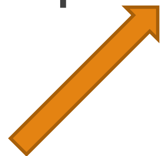
Displaying Variables (and text)

- The ‘general’ function in C we use do display output is

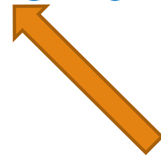
printf

- It is a function that can take one or more parameters
  - This is somewhat ‘unusual’ in programming in C where functions generally expect a fixed number of parameters.
- There must be at least one parameter – the text to display

```
printf("Hello world!");
```



Function: printf, used to output to the display



Parameter: The text to be displayed contained in double quotation marks

Formatting Characters

- There are some formatting options for things that we cannot ‘type’ into code (e.g. a ‘new line’)
- The two most common are
  - \n Insert a new line
  - \t Insert a TAB character
- There are more – take a look on-line!
- https://www.ibm.com/docs/en/rdfi/9.6.0?topic=set-escape-sequences

Displaying the contents of variables

Variable place holders – replaced (at run-time) with the contents of a variable

- %d Used to display an int (you can also use %i)
- %f Used to display a floating number
- %c Used to display a single character
- %s Used to display a string (of **char**acters)
- %x Used to display in hexadecimal
- %#x Used to display in hexadecimal with 0x in front of number

An example of formatting and place holders

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int a,b,c,sum;          /* Define variables */
    a = 1;                  /* Assign values */
    b = 2;
    c = 3;
    sum = a + b + c ;       /* Calculate sum & Display */

    printf ("\nThe sum of %d + %d + %d is %d \n", a, b, c, sum);

    return 0;               /* Return from prog */
}
```

LC5\printf_example.c

Tidying up output

We can ‘enhance’ the variable format string (%d, %f) to improve how we display numbers

Things that can be specified are

- The number of characters to used to display a value
- Where whitespace will be added
  - Before / after the text to be outputted

Note:

- For numbers if more characters are required than that ‘stated’ in the formatting string, the value is over-ridden
- For strings the output is truncated

Tidying up output (2)

- For integers we can specify the number of characters to use (space will be used to pad)
    - %6d Print as an integer with a width of at least 6 wide, whitespace added at the ‘front’
    - %-6d Print as an integer with a width of at least 6 wide, whitespace added at the ‘end’
- Reminder:
    - If more characters are actually needed (e.g. we specify 4 but the number to display is 123456 the format will be automatically overridden)

Tidying up output (3)

For floats we can specify the number of characters to use in total for the number as a whole (can be omitted) and the precision

- %4f Print as a floating point with a width of, at least, characters 4 wide (precision not specified)
- %.4f Print as a floating point with a precision of four characters after the decimal point
- %3.2f Print as a floating point at least 3 wide and a precision of 2DP

C5\formatting_numbers.c

There are a few others

- Some further examples

%e	64-bit floating-point number (double), printed in scientific notation using a lowercase e to introduce the exponent.
%E	64-bit floating-point number (double), printed in scientific notation using an uppercase E to introduce the exponent.

%x	Unsigned 32-bit integer (unsigned int), printed in hexadecimal using the digits 0–9 and lowercase a–f.
%X	Unsigned 32-bit integer (unsigned int), printed in hexadecimal using the digits 0–9 and uppercase A–F.

A quick on-line search for formatting options in C will give you a very long list of options!

Chapter 6

Operators in C

Operators (part 1)

- This is just the term we use to indicate that we plan to perform a mathematical or logical operation.
- C provides a range possible operations, listed below
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Bitwise Operators
  - Assignment Operators
  - Misc. Operator

Arithmetic Operators

- These are the basic one we use in everyday mathematics – every computing language provides them (almost all using the same symbols)
- If we assume A & B have previously been defined and that A=10 and B=3

Operator	Description	Example	
+	Addition	A + B	13
-	Subtraction	A – B	7
*	Multiply	A * B = 21	21
/	Divide	A / B	3.33 or 3 (dependant on variable types)
%	Modulus – the remainder after a division	A % B	1
++	Increment operator: adds 1 to the value	A++	11
--	Decrement operator: subtracts 1 from the value	A--	9

++A will increment A before it is used. A++ will use A with its current value and then increment (same for --)

C6\inc_dec_examples.c

Logical Operators

- We use these to combine relationships into more complex cases
- Assuming A = 1 and B =0

Operator	Description	Example	Result
&&	The is a Logical **AND** operator. If both the operands (values) are non-zero then the condition equates as TRUE	(A && B)	FALSE
||	The is a Logical **OR** operator. If either (or both) of the operands (values) are non-zero then the condition equates as TRUE	(A || B)	TRUE
!	This is the Logical **NOT** operator : this reverses the logical state of a condition	!(A == B)	TRUE

Note: There are also ‘Bitwise’ logical operations which operate on the individual bits – these are covered later

Relational Operators

- We use these to make comparisons between variables – the result is TRUE or FALSE
- If we assume A & B have previously been defined and that A=10 and B=3

Operator	Description	Example	Result
==	Checks if values are equal, if yes then TRUE else FALSE	(A == B)	FALSE
!=	Checks if values are NOT equal, if yes then TRUE else FALSE	(A != B)	TRUE
>	Checks if the left operand (value) is greater than the right	(A > B)	TRUE
<	Checks if the left operand (value) is less than the right	(A < B)	FALSE
>=	Checks if the left operand (value) is greater than or equal to the right	(A >= B)	TRUE
<=	Checks if the left operand (value) is less than or equal to the right	(A <= B)	FALSE

Bitwise Operators

- Unlike the LOGICAL operators which considered the values as zero or non-zero, bitwise operations act in the same way as you have been covering in digital electronics
- Assuming A = 60 decimal, 00111100 Binary   and B = 13 00001101 Binary

Operator	Description		Result (binary	Result (dec)
&	Performs a bitwise AND	0011 1100 & 0000 1101	0000 1100	12
|	Performs a bitwise OR	0011 1100 | 0000 1101	0011 1101	61

A & B:

0011 1100
0000 1101
0000 1100

A|B:

0011 1100
0000 1101
0011 1101

Bitwise Operators(2)

Assuming A = 60 decimal, 00111100 Binary and B = 13 00001101 Binary

Operator	Description		Result (binary)	Result (Dec)
&	Performs a bitwise AND	0011 1100 & 0000 1101	0000 1100	12
|	Performs a bitwise OR	0011 1100 | 0000 1101	0011 1101	61
^	Exclusive OR (XOR)	0011 1100 ^ 0000 1101	0011 0001	49
~	Ones complement ‘flips’ bits	~0011 1100	1100 0011	-61
<<	Left shift, moves bits left number of bits specified by the number on the right	0011 1100 << 2	1111 0000	240
>>	Left shift, moves bits right number of bits specified by the number on the right	0011 1100 >> 2	0000 1111	15

Assignment Operators

- These are, in some way, just ‘shorthand’ – you may like to use them but it is not essential.
- There are many of these, below are some of the more ‘common’ ones (others you can probably guess!)

Operator	Description	Example	‘Long hand’
=	Assigns the result of the RHS to the variable on the LHS	C = A + B	C = A + B
+=	Add and Assign combined.	B += A	B = B + A
-=	Subtract and Assign combined.	B -= A	B = B - A
*=	Multiply and Assign combined.	B *= A	B = B * A
/=	Divide and Assign combined	B /= A	B = B / A
%=	Divide and Assign combined	B %= A	B = B % A

Chapter 7

Input: Reading in information

Input and Output

- Most of the work we have done so far has resulted in values being displayed on the screen, where we have needed values for things to work we have ‘hard coded’ these into our code
- In practice we will however need to ‘request’ information from the user of our programs
  - We will need to check their input is ‘sensible’ – so often not the case!
- In C we have a few methods provided that allow us to have values inputted, like much in programming it is up to us to decide which is the most suitable approach
- We will look at a few methods
  - Reading in numerical values
  - Reading strings (the programming term for ‘text’)
  - Capturing a keypress (without the need for ‘return’ to be pressed)

Input: scanf – reading ‘real’ values (1)

- The command we will make use of to read input from the keyboard is **`scanf`** (defined in stdio.h)
- You are most of the way there already as scanf, the function we use, takes very much the same format as printf (the function we use to display variables on the screen).

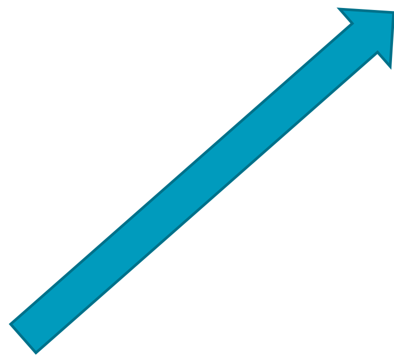
- First, a reminder…
  - Assuming a variable ‘a’ had been defined as an integer
  - To display it on the screen we would use

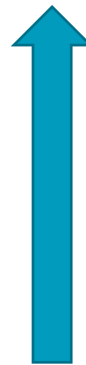
```
printf ("\nThe value of a is %d", a);
```

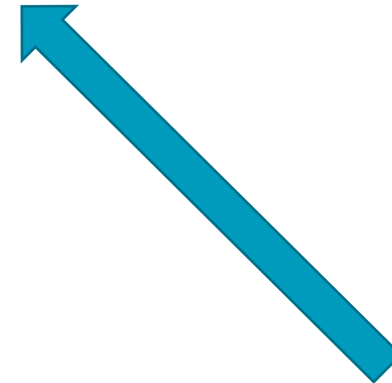
Input: scanf – reading ‘real’ values (2)

- To read a value entered at the keyboard into **a** we use

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







Scanf The function for reading from the keyboard into a variable	%d Indicates to scanf that an integer Is to be read	&a Read into the variable a You **MUST** put a **&** before the variable

Input: scanf – reading ‘real’ values (3)

- We can read in multiple values (even of mixed types)
  - we just use the correct formatting

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

Scanf
The function for reading
from the keyboard into a variable

%d %f
Indicates to scanf that an integer
then a float is to be read

&a, &b
Read into the variables a & b
You **MUST** put a **&** before each variable

scanf: Some examples in code

We will look at how we use for *scanf* in practice

C7\scanf_examples.c

Input and Output (part 1)

Scanf is a ‘general’ function that allows us to read in values to any C variable previous defined

It does however have a few limitations:

- The user must press ‘return’ to confirm input
- The function is ‘large’ (so may not fit into available program memory)
- It is not great for reading single characters (the ‘return’ causes problems)

There is an alternative method to capturing a keypress

- Also, as it is a very ‘low level’ function, it does not require much program memory

Input: getchar() - capturing a keypress

The function is **`getchar`**, define in stdio.h as

```
int getchar (void)
```

This is the simplest form of input

- we simply 'wait' for a key to be pressed and store this in a suitable variable
- Although an **int** is returned, we can store the result in a **char** as the value returned will always be in the range 0-255)

Let's take a look at this in action

C7\getchar_example.c

Input: getch() - capturing a keypress

`getchar()` does have a problem; it needs a ‘return’ which can cause problems later

An alternative (non-standard, but works on a PC) is to use

```
int getch (void)
```

To use this you need to add `#include <conio.h>` to your code

Let’s take a look at this in action

LC7\getch.c

Strings: A special case

- A **string** is an array (covered later) of individual characters that we treat as a single piece of text
- To define a string we simply create an array of chars large enough to hold our ‘text’.
  - We do this by adding `[n]` after the variable name,
  - Where `‘n’` is the maximum number of characters for our string + 1 (to allow for the end of string marker), e.g.

```
char Surname[51]
```

    - Provides a variable that can hold a string of up to 50 characters

- We can then read/write this using the formatting character %s

Strings: A practical approach (1)

The following shows how we can create a string, read into it and print out the contents

```
int main()
{
    char MyName[50];
    printf ("\nWhat is your name? ");
    scanf ("%s",MyName);
    printf ("\nHello %s", MyName);
    return 0;
}
```

Note: We drop the **&** here as scanf knows from the %s we are reading into an array

LC7\scanf_string.c

Strings: A practical approach (2)

`scanf` is ok for strings but has problems if we include a space (it stops reading from that point.

A better function to use is `gets()`

```
int main()
{
    char MyName[50];
    printf ("\nWhat is your name? ");
    gets (MyName);
    printf ("\nHello %s", MyName);
    return 0;
}
```

Note: We drop the **&** here as gets knows we are reading into an array

LC7\gets_string.c

Strings: A better approach

Even better coding is to use a slightly more advanced approach which allows us to specify the maximum numbers of characters that can be read (any beyond this are discarded).

Limit the number of characters to read

```
int main()
{
    char MyName[50];
    printf ("\nWhat is your name? ");
    fgets (MyName, 50, stdin);
    printf ("\nHello %s", MyName);
    return 0;
}
```

fgets is a function to read from a file device

stdin is the standard input (keyboard)

LC7\fgets_string.c

Lab 1 Programming Assignment

Introduction

Lab 1 Programming Assignment

Submission: 3pm, Thursday 26th October 2023

Files on Moodle:

- Project brief
- Mark scheme
- Flowchart templates
- Sensor and encoder skeleton codes

By the end to today's lecture you should be able to complete the encoder program.
Next week we will cover the material needed for the sensor program (functions).

Chapter 8

Program Flow in Code

Decisions: if

In coding, there are times when we need to follow a different path dependent on the current state of (say) a variable

We will know these possible routes from the design of our code (and possibly, any associated flow charts)

From this knowledge we can define conditions which will be true/false

We when implement these conditions in code – leading to a ‘program flow’

The ‘**if**’ Statement

```
if( expression )
    statement_to_execute_if_true;
```

Note:
We can also have a block of code in {} controlled by the if statement

- Condition to be tested is in parenthesis and is made up using ***relational operators (and, if required, logical operators)***
- If non-zero (TRUE), the statement is executed

Decisions: if

Quick in-class exercise

Assuming x & y have been defined:

write `if` statements that equate non-zero (true) if

- x > y
- x <> 3 and y > 7
- x <= 7 and (x + y) > 15
- x = y or x = 7

Decisions: if (Solutions)

Solutions:

x > y

```
if ( x > y )
```

x <> 3 and y > 7

```
if ( (x != 3) && ( y > 7 ) )
```

x <= 7 and (x + y) > 15

```
if ( ( x  <= 7 ) && ( ( x + y ) > 15 ) )
```

x = y or x = 7

```
if  (( x == y ) || ( x == 7 )
```

Decisions: if - in practice

- We will look at some examples using
  - == Equal to (NB **TWO** EQUAL SIGNS)
  - != Not equal to
  - > Greater than
  - < Less than
  - >= Greater than or equal to
  - <= Less than or equal to

LC8\if_examples.c

A WARNING !!!

Watch out when using ‘==‘ and ‘=‘ with **if**

```
if ( a = 1 )
     statement;
```

Sets a equal to 1 (which returns true) so the statement is executed

```
if ( a == 1 )
     statement;
```

Executes the statement ONLY if the value of a is 1

THIS IS A VERY COMMON MISTAKE TO MAKE !

C8\if_equals_example.c

Examples of if

- Single Statement (already seen)

```
if ( a == 5 )
     printf  ("a is equal to 5");
```

For multiple statements lines, use {}

```
if ( a > b )
{
    temp = b;
    b = a;
    a = temp;
}
```

Note: You can (and may wish to) use {} even for single statements

Else - Used to expand capabilities of *if* to include the 'not true' case

If we need statements to be executed if the condition is not met, we add an else e.g.

```
if ( a == 5 )
    printf("a is equal to 5\n");
else
    printf("a is not equal to 5\n");
```

Note:

We use {} if multiple lines of code are required to be controlled by if/else

Simple example (1)

```
if ( x == 2 )
{
   printf ("The value of x was 2\n");
   printf ("I will now do something\n");
}
else
   printf ("Not 2, so I will do nothing ");
```

Note:

As the ‘else’ is only controlling one line we can omit the { }

You may however wish to place even a single line of code inside { }

else / if

We can ‘chain’ if, else if & else should it be necessary

```
if ( a == 1 )
   printf ("A = 1\n");
else if ( a == 2 )
   printf ("A = 2\n")
else
   printf("A is neither 1 or 2\n");
```

Simple example (2)

```
if ( x == 2 )
{
   printf ("\nThe value of x was 2");
   printf ("\nI will now do something");
 }
else if ( x == 3 )
{
   printf ("\nThe value of x was 3");
   printf ("\nI will now do something else");
 }
else
   printf ("Neither 2 or 3; I will do nothing");
```

C8\if_else_if_else.c

More complex *if* conditions

So far we have had ‘single’ expressions,

We use LOGICAL operators (chapter 6) to construct more complex conditions

or	||
and	&&
not	!

Combine these within ***if*** statements, e.g.

```
if ( ( age > 5 )  && ( age < 16 ))
     printf ("You should be in school ! \n");
```

C8\complex_if.c

Use of single variables in if conditions

You may see statements within an 'if' statement which use a single variable, or a single variable combined with a 'not' operator:

Equivalent to:

```
if (index)
{
      printf( "index has a value – not 0 \n";
}
```

Test is true for any non-zero value of index.

```
if (index != 0 )
```

```
if (!index)
{
      printf( "index is 0 \n";
}
```

Test is true for only when index is 0

```
if (index == 0 )
```

switch – case construct

Equivalent to `if, else if .. else`

Makes for easier reading of code

Allows for different ‘cases’ to produce the same result

Use **`break`** once a to drop-out of **`case`**

- (causes control to pass to the statement following the innermost enclosing **`while, do, for`** or **`switch`**)

switch vs if / else if / else

if, else if .. else	switch

```
if ( c == 'a' )
{
   printf("Hi");
}
else if ( c == 'b' )
{
   printf("Bye");
}
else
{
   printf("Err")
}
```

```
switch (c)
{
    case 'a' :  printf("Hi");
                break;
    case 'b' :  printf("Bye");
                break;
    default :   printf("Err");
                break;
}
```

Note: {} are optional here as only one line of code of controlled by each condition

Multiple Cases within Switch

if, else if .. else

```
if ( c == 'a' || c == 'A' )
{
   printf("Hi");
}
else if ( c == 'b' || c == 'B' )
{
   printf("Bye");
}
else
{
    printf("Err")
}
```

switch

```
switch (c)          /* Switching on c (a char) */
{
    case 'a':                /* Case 'a' or 'A'  */
    case 'A':  printf("Hi");
              break;

    case 'b':                /* Case 'b' or 'B' */
    case 'B':  printf("Bye");
               break;

    default :  printf("Err"); /* Default action */
              break;
}                        /* End of switch */
```

Applies to any integral type

```
int a;                              /* Define an int */
scanf ("%d",&a);                    /* Get value */
switch (a)                          /* Start of switch */
{
    case 1:   printf("Hi");         /* Case 1 */
              break;

    case 2:   printf("Bye");        /* Case 2 */
              break;

    default :printf("Err");         /* Default */
              break;
}  /* End of switch */
```

C8\switch_1.c

A reminder for numbers (when characters!).

```
char c;                            /* Define a char */
c=getch();                         /* Get value */
switch (c)                         /* Start of switch */
{
    case '1':    printf("Hi");     /* Case '1' */
                 break;

    case '2':    printf("Bye");    /* Case '2' */
                 break;

    default :    printf("Err");    /* Default */
                 break;
}  /* End of switch */
```

C8\switch_2.c

A clever trick – omit the break (if you mean to!)

```
char c;                                 /* Define a char */
c=getch();                              /* Get value */
switch (c)                              /* Start of switch */
{
    case '1':    printf("Hi");          /* Case '1' */

    case '2':    printf("Bye");         /* Case '2' */
                 break;

    default :    printf("Err");         /* Default */
                 break;
}  /* End of switch */
```

C8\switch_3.c