EG-151 Microcontrollers

Lecture 7: Introduction to Programming with C

Ben Clifford

b.r.clifford@swansea.ac.uk

11th November 2019

Part I

C Language

Functions, Statements, Variables, Data Types, Comments, Operators and Compilation

b.r.clifford@swansea.ac.uk

History of C Programming Language

- C was originally developed at Bell Labs by Dennis Ritchie in the early 1970s to make utilities running on Unix systems.

- During the 1980s, C gained popularity in microcomputer and microcontroller applications and in 1989 was standardised by the American National Standards Institute (ANSI) and was referred to as C89 or ANSI C.

- Subsequently in 1999, the language was standardised by the International Organization for Standardization (ISO) and referred to as C90. Since then there have been several iterations, each time introducing new features.(C99, C11 and most recently C18).

- Nowadays, C is one of the most widely used programming languages with compilers from various vendors available for most computer architectures and operating systems.

b.r.clifford@swansea.ac.uk

C Program Structure

- C is a high-level structured programming language and programs are built up from series of modules referred to as functions, where each function is designed to perform a specific task or set of tasks.
- Recall from lecture 3; A module or function is a self-contained block of program code which performs a specific set of actions and has a name by which it can be referred to and called up.
  - Each module receives data as an input, processes the data and returns a result.
  - In assembly language, modules were termed subroutines.

b.r.clifford@swansea.ac.uk

Functions in C

- In C, functions are given names that convey some idea of what the function does;

add_num

delay

random_number_generator

- Function names must begin with a letter or underscore character (_) and can contain letters, numbers and underscores.
  - Blank spaces are not permitted so typically an '_' is used in its place to separate words (*add_numbers*).
- C is a case sensitive language, so a function called 'delay' would be a different function to 'Delay' or 'DELAY'.
- Generally function names are written in all lower case letters.

b.r.clifford@swansea.ac.uk

Functions in C continued

- In C, there are 32 reserved keywords and these can not be used as function names.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

C language keywords

b.r.clifford@swansea.ac.uk

Functions in C continued

- Function names are followed by round brackets ‘( )’ which enclose the argument(s) to the function.
- These arguments are passed into the function when it is called.
- A function is executed by calling it somewhere in the program code.

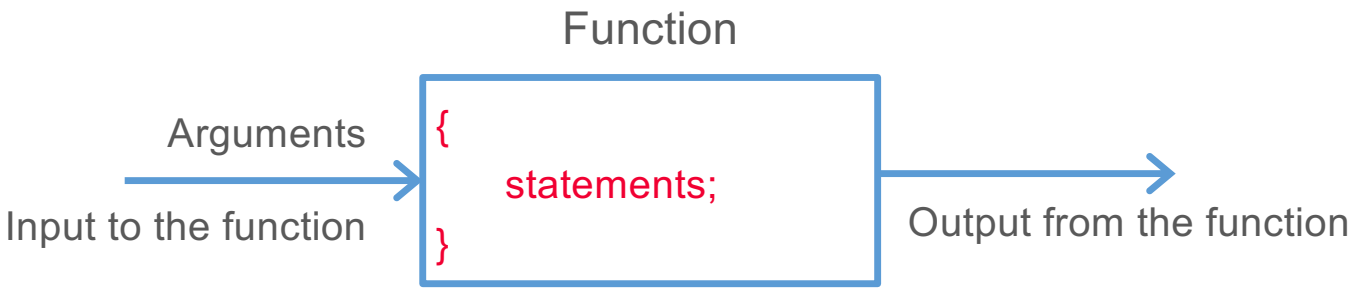
```
my_function (arguments);              //function call

type my_function (type, argument)   //function
{
    function body;
}
```


b.r.clifford@swansea.ac.uk

Statements in C

- C functions are made up from statements, each of which is terminated with a semicolon ‘ ; ’.
- The statements within a function are grouped by putting them between braces (parentheses) ‘{ }’

```
function_name ()          //function
{
    statement 1;
    statement 2;
}
```



b.r.clifford@swansea.ac.uk

Variables in C

- The term variable is used for a name which describes a memory address and follows the same naming convention as used for functions.

- We might have the variable name num1 to describe a particular memory address at which the first number is found and then num2 to describe a second memory address.

- In a function we may then have the following statements:

```
num1 = 20;
num2 = 30;
total = num1 + num2;
```

- Each of the statements in the example above tells the computer system to assign/store a value into the memory address described by the variable – assignment.

- Assignment statements use the equals (=) sign to assign the value on the right of it to the variable on the left.

b.r.clifford@swansea.ac.uk

Variable Types in C

- Before storing a value in a variable, the type of the data to be stored must be defined.
  - This is done using a declaration statement and is usually placed at the start of code (global variables).
- To declare a variable, the type is inserted before the variable name.
- Type declarations within a function appear immediately after the opening brace and are called local variables.
- So in the case of the previous example, before we can store values in the variables, num1, num2 and total, we must first declare the type of data that is going to be stored in them:

```
int num1;
int num2;
int total;
```

```
int num1, num2, total;
```


b.r.clifford@swansea.ac.uk

Variable Types in C continued

The C language supports 2 different forms of data types:

1. Primary data types:
   - These are fundamental data types in C namely integer(int), floating point(float), character(char) and void.

2. Derived data types:
   - Derived data types are nothing but primary datatypes but a little twisted or grouped together and include array, structure, union and pointer.

Note: All primary data types may be changed with the -T: Flexible Type Management compiler option.
All primary types (except char) are without a signed/unsigned qualifier, and their default values are signed (e.g. int is the same as signed int).

b.r.clifford@swansea.ac.uk

Integer Variable Types

Integer type variables are used to store whole numbers.

Type	Default Size (bytes)	Range		Formats available with the –T option
		Min	Max	
int or signed int	2	-32,768	32767	8-bit, 16-bit, 32-bit
unsigned int	2	0	65535	8-bit, 16-bit, 32-bit
short int or signed short int	2	-32.768	32,767	8-bit, 16-bit, 32-bit
unsigned short int	2	0	65,535	8-bit, 16-bit, 32-bit
long int or signed long int	4	-2,147,483,648	2,147,483,647	8-bit, 16-bit, 32-bit
unsigned long int	4	0	4,294,967,295	8-bit, 16-bit, 32-bit

More information about the data types available and use of the –T option for the HCS08 series of microcontroller can be found on page 464 of the HC(S)08 Compiler Manual.

b.r.clifford@swansea.ac.uk

Floating Point Variable Types

Floating point type variables are used to store real numbers (includes fractional part, e.g. 1.234).

Type	Default Size (bytes)	Range		Formats available with the –T option
		Min	Max	
float	4	$1.17E^{-38}$	$3.40E^{38}$	32-bit, 64-bit
double	8	$2.22E^{-308}$	$1.79E^{308}$	32-bit, 64-bit

More information about the data types available and use of the –T option for the HCS08 series of microcontroller can be found on page 464 of the HC(S)08 Compiler Manual.

b.r.clifford@swansea.ac.uk

Character Variable Types

char type variables are used to store characters.

Type	Default Size (bytes)	Range		Formats available with the –T option
		Min	Max	
char or unsigned char	1	0	255	8-bit, 16-bit, 32-bit
signed char	1	-128	127	8-bit, 16-bit, 32-bit

More information about the data types available and use of the –T option for the HCS08 series of microcontroller can be found on page 464 of the HC(S)08 Compiler Manual.

b.r.clifford@swansea.ac.uk

Void Variable Type

Type *void* means no value.

This is used to specify the type of functions which returns nothing or take no arguments.

b.r.clifford@swansea.ac.uk

Types with the –T option

- Since memory space on microcontrollers is typically small, when writing microcontroller application code it is best practice to avoid using the basic data formats – char, int, short, long and in their place use types specified with the –T option, allowing the programmer to specify the size of each data type. For example:

  Use *uint8_t* in place of *unsinged int*.

- Here we define an 8-bit unsigned integer type (uint8_t) in place of the default 16-bit unsigned integer (unsigned int).

- Most –T options are available in the newer versions of the C language library and are included in the stdint.h header file

- The compiler we use is based on an older version of the standard C language library (C89) and does not have these types included as standard or support inclusion of the stdint.h library which contains them in the newer standard C language library.
  - In order to use these types we must first define them – typedef unsigned char uint8_t

b.r.clifford@swansea.ac.uk

Arrays

An array is a collection of data storage locations each one having the same data type, but all referenced through a single name.

The size of the array is indicated between square brackets [ ] immediately after the name and is required in the declaration:

```
float temperature[10];          /* type array_name[size] */
```

Elements in an array are accessed using an index number: the first index number is 0, the second is 1 and so on up to n-1 where n is the size of the array.

To read and write to the array we use the array name and the index position we want to access, for example to write to the first two elements in the array:

```
temperature[0] = 20.2;          /* write to the first position in the array */
temperature[1] = 20.5;          /* write to the second position in the array */
```

and to read from them:

```
a = temperature[0];             /* read from the first position in the array*/
b = temperature[1];             /* read from the second position in the array*/
```


b.r.clifford@swansea.ac.uk

More variable declaration examples

```
typedef unsinged char uint8_t;
uint8_t my_variable;

int my_array[10];

char a_letter;
a_letter = 'x';

#define pi 3.1415;
```

```
float my_array[5] = {20.2, 21.4, 20.9,19.8, 20.4}
```


b.r.clifford@swansea.ac.uk

Operators

- Similarly to assembly language statements/instructions are made up of an operator and a number of operands, the operators in C can be split into the same four categories we saw with assembly mnemonics based on the operation that they perform:
  - Data Transfer
  - Arithmetic
  - Logical
  - Program control

- In ‘C’ the data transfer operations are mostly covered by assignment operations without ever having to deal directly with the accumulator and other registers.
  - A = 10, B = 20, Total_Answer = A + B, PTFD = Total_Answer, etc.

b.r.clifford@swansea.ac.uk

Arithmetic Operators

- Arithmetic operations follow the BODMAS (BIDMAS) rules

Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Increment	++
Decrement	--

Examples

assume a – c are defined as integer type variables

```
a = 2 * 5 + 6 / 3;          a = 12;

b = (3.14 * r * r);         /* brackets can be used to improve readability */

c = c++;                    /* c = current value of c + 1 */
```


b.r.clifford@swansea.ac.uk

Arithmetic Operators continued

assume d – h are defined as integer type variables and i as a floating point number

```
d = 5/3;        d = 1     /* gives the real part of the answer */

e = 5%3;        e = 2     /* gives the fractional part */

f = 6%3;        f = 0     /* gives 0 as there is no fractional part */

g = 5.5/2;      g=2

h = 5.5%2;                /* invalid – will not compile */

i = 5.5/2;      i = 2.75  /* will evaluate correctly if i is defined as a floating point number */
```

Same rules apply to multiplication operations

b.r.clifford@swansea.ac.uk

Logical Operators

And	&&
Or	||
Not	!

if ((voltage > 10) && (current < 20))

if the voltage is greater than 10 and the current is less than 20 the condition is true and the result will be 1, otherwise it is 0.

If((voltage > 10) || (current > 20))

if the voltage is greater than 10 or the current is greater than 20 the condition is true and the result will be 1, otherwise it is 0.

c = 0;

!c = 1

The unary operator (!) is usually used to turn true into false and vice versa.

b.r.clifford@swansea.ac.uk

Logical Operators - Bitwise

AND	&
OR	|
XOR	^
NOT	~
Shift Right	>>
Shift Left	<<

- In addition to the logical operators designed to evaluate multiple conditions that were discussed on the previous slide, there are bitwise logical operators which work in the same way as the logic operators in assembly language.

```
  1010 0011
& 1101 0101
  1000 0001

  1010 0011
| 1101 0101
  1111 0111

~ 1010 0011
  0101 1100
```

e.g. portA = portA | 0xF7; /* used to mask off bits */

b.r.clifford@swansea.ac.uk

Relational Operators

The final group is relational operators which are similar to the program control operations we used in assembly language.

Is equal to	==
Is not equal to	!=
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=

Is x equal to 2 would be written as (x == 2)

Important: don't confuse the double equals sign with the single one used for assignment operations.

(x = 2) is different from (x == 2)

Is x greater than 10 if(x > 10)

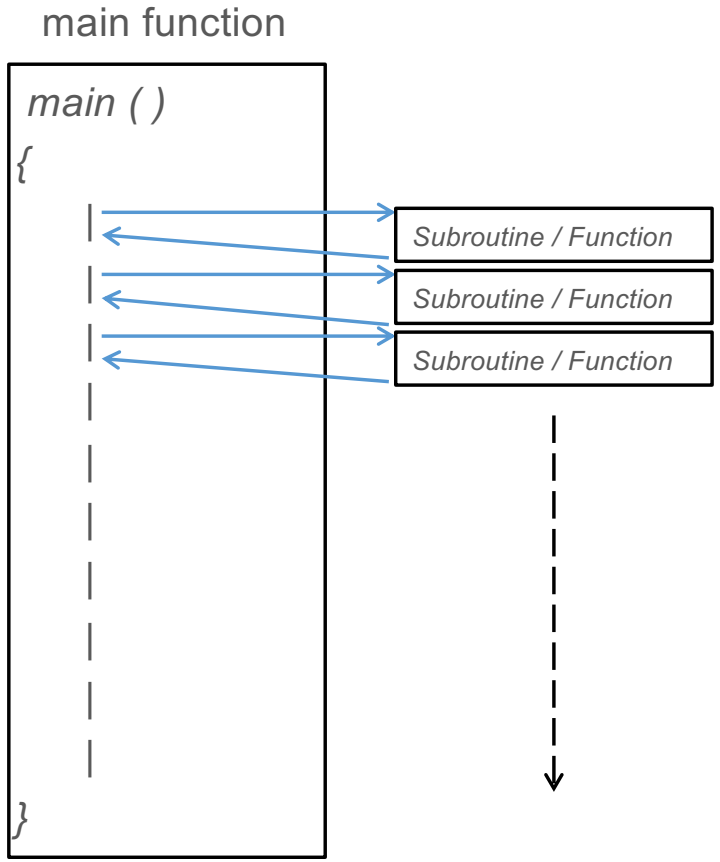
Is y less than 10 if(y < 10)

b.r.clifford@swansea.ac.uk

The “main” function

- C programs are built up from functions each designed to perform a specific task or set of tasks.
- In C, we use a function called *main( )* to combine all of these modules/functions in the required order.
- Every program must have a function called *main( )* in which the other functions are called and execution begins with the first statement in the *main( )* function.

```
/* Calculate cost of meal in a restaurant */
main( )
{
        input_cost();
        tip();
        total_charge();
}
```



Comments in C

- In the same way that we used comments in assembly to add descriptions to blocks and lines of code we can also add these in ‘C’

- In the ‘C’ language comments are surrounded by /* comment */ and can span multiple lines or start with // if they are on a single line.

```
A = 10;        // This line sets A equal to 10

/* This comment
spans multiple lines */
```


b.r.clifford@swansea.ac.uk

Functions Revisited – defining and declaring

- The purpose of a function is to receive some data, operate on that data and then return data back to the main function.
- A function is called using its name followed by brackets containing the data to be operated on.

function_name(data to be passed);

- If the function is going to return data then the return type must be specified before the function name.

int function_name(data to be passed);

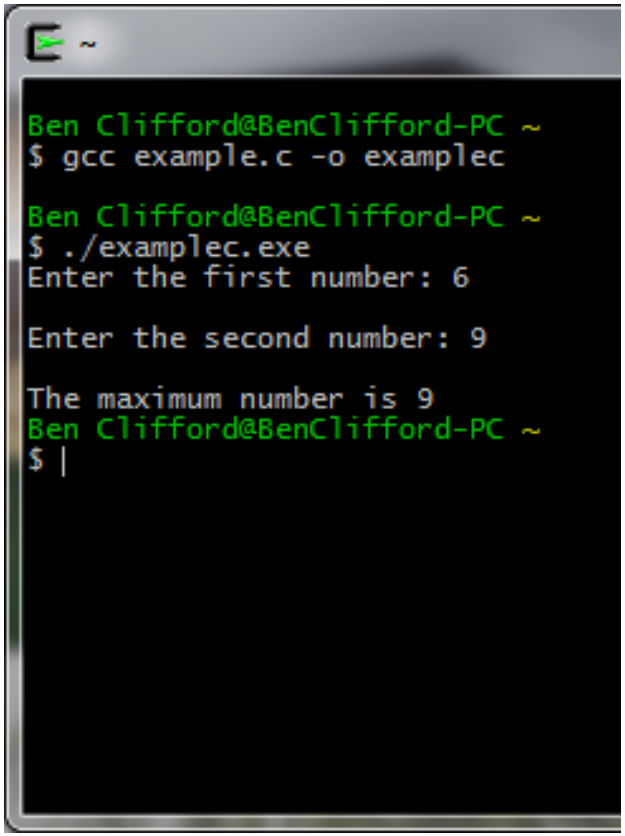
- Before the function can be called in code it should be defined at the start of code (but can be defined in place).

int function_name (int x, int y, int z);

b.r.clifford@swansea.ac.uk

A short example

```c
#include <stdio.h>

int main(void)
{
    int num1, num2, maxnum;          /* Variable definitions */
    int findmax(int, int);           /* findmax function definition */

    printf("Enter the first number: ");
    scanf("%u",&num1);

    printf("\nEnter the second number: ");
    scanf("%u",&num2);

    maxnum = findmax(num1,num2);     /* Call findmax function */

    printf("\nThe maximum number is %u",maxnum);
}

int findmax(int x, int y)            /* Start of findmax function */
{
    int maxnum;
    if(x >= y)
    {
        maxnum = x;
    }
    else
    {
        maxnum = y;
    }
    return(maxnum);                  /* Return maxnum to the main function */
}
```

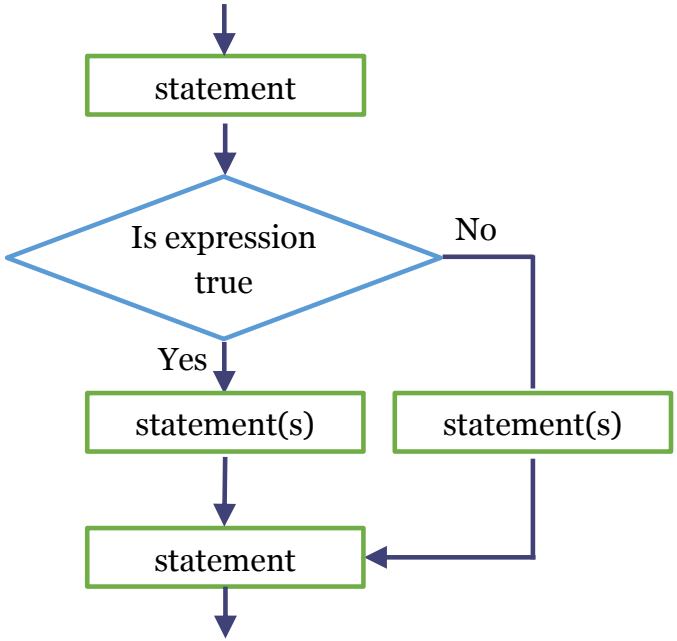


b.r.clifford@swansea.ac.uk

Flow Control – The if Statement

- The *if* statement allows branching within code. It can be used to check if a particular condition has been met.
- If the decision evaluates to true then a statement is executed, if not true then the statement is not executed and the program will branch to the next statement or check the next condition.
- The syntax of the *if* statement is:

```
if (variable_1 == 1 ) {
        statements;
}
else if( variable_1 > 1 ){
        statements;
}
else {
        statements;
}
```

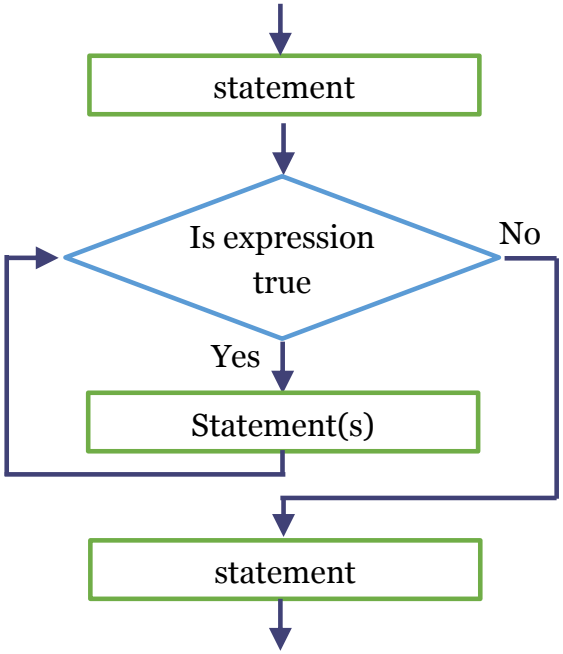


b.r.clifford@swansea.ac.uk

Flow Control – The while Statement

- In order to write a function that loops, i.e. execution of a sequence of statements until a particular condition is met, a *while* statement is used.
- The while statement allows for a loop to be continuously repeated as long as the condition is true.
- The syntax of the *while* statement is:

```
while (condition 1 < condition 2)
{
        statements;
}
```



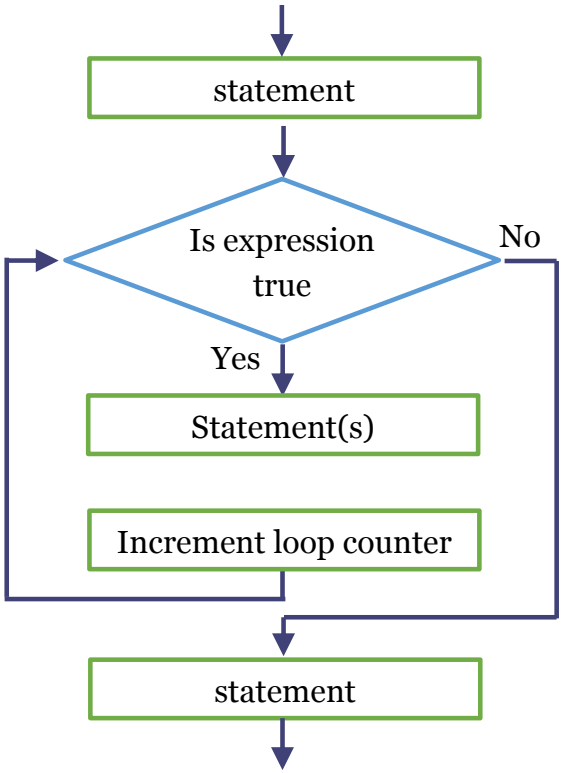
b.r.clifford@swansea.ac.uk

Flow Control – The *for* Statement

- Another way of writing a *while* statement is to use a *for* loop.
- The term loop is used for the execution of a sequence of statements until a particular condition is met.
- The syntax of the *for* statement is:

```
for (x = 0; x<10, x++)
{
        statements;
}
```

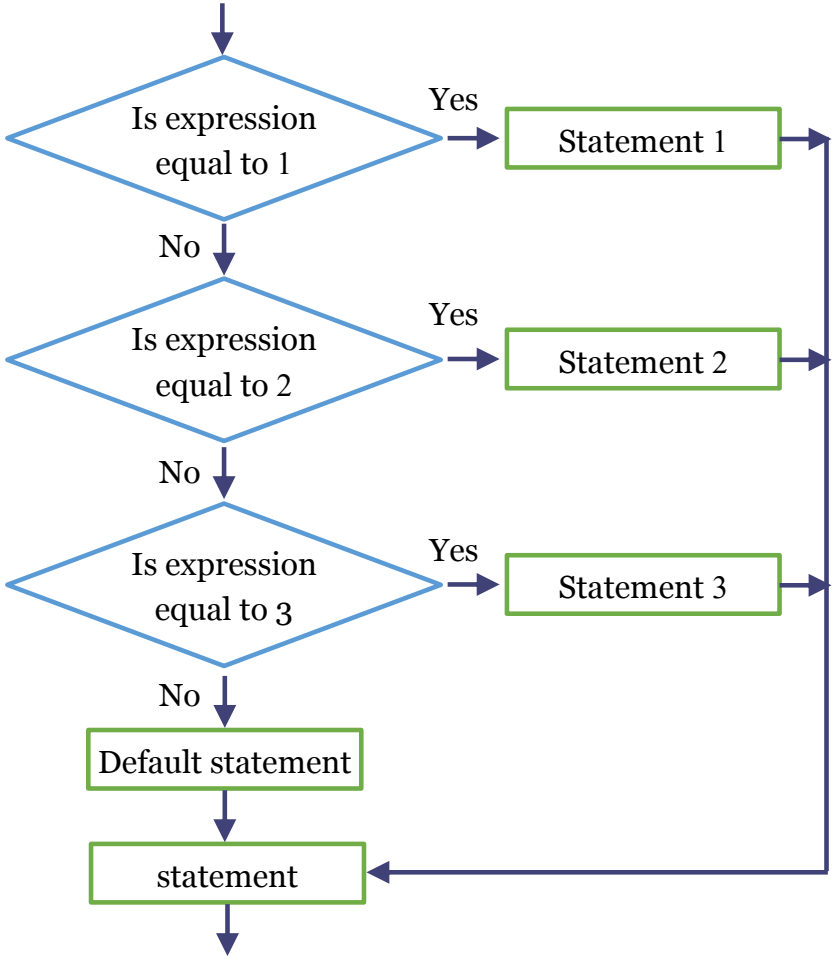
- The variable x starts at 0 and is incremented after completing the statements within, while x is less than 10.



b.r.clifford@swansea.ac.uk

Flow Control – The switch Statement

- The switch statement allows selection between several possible options.
- The test condition is referred to as the expression and the choices are referred to as the cases
- Each switch expression will have a default choice for when no choices match the expression.



b.r.clifford@swansea.ac.uk

Flow Control – The switch Statement

The syntax of the *switch* statement is:

```
switch(expression)
{
        case 1:
        statements;
        break;

        case 2:
        statements;
        break;
          ⋮
        default:
        statements;
}
```

```
char grade = 'B'

switch(student_grade)
{
        case 'A' :
          printf("Excellent!" );
          break;

        case 'B':
          printf("Well Done!" );
          break;


        default:
          printf("Invalid grade\n" );
}
```


b.r.clifford@swansea.ac.uk

Pre-processor

- The pre-processor is part of the compilation process and runs before the code is compiled, performing a similar function to the assembler when processing assembly directives.

- The pre-processor looks for lines of code beginning with a # and evaluates them before compilation.
  - There are also a number of predefined macros which can be called and these are surrounded by "__" characters.

- The most commonly used pre-processor directives, and also the ones we will be using, are #include and #define.

#include
#define
#undef
#if
#elif
#endif
#ifdef
#ifndef
#error
__FILE__
__LINE__
__DATE__
__TIME__
__TIMESTAMP__
macro operator
macro operator

b.r.clifford@swansea.ac.uk

Pre-processor directives

The #include directive is used to include header files which contain declarations of existing and frequently used functions that can be substituted into your program.

#include <header_file.h>

/*This variant is used for system header files. The pre-processor searches for a file named in a standard list of system directories*/

#include “my_library.h”

/*This variant is used for header files of your own program. It searches for the file name first in the current directory, then in the same directories used for system header files. These files may contain custom function definitions, common functions, secondary data type definitions, etc.”

The #define directive is used to define a macro – when the macro name appears in code it will be replaced with the definition stated. For sample:

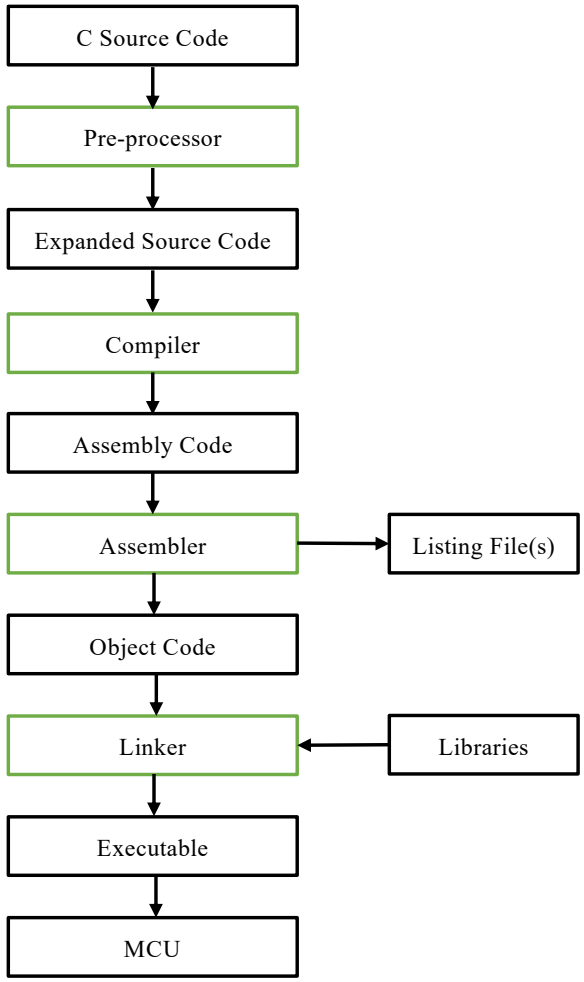
```
#define pi 3.14
#define cube(x) (x)*(x)*(x)
```


b.r.clifford@swansea.ac.uk

Compiler

- A compiler is a special program that processes statements written in a particular programming language and translates them into another language or in some cases directly into machine code.

- When executing, the compiler first parses (analyses) all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.



b.r.clifford@swansea.ac.uk

Part II

C Language Examples

b.r.clifford@swansea.ac.uk

Assembly to C

```
MAIN_LOOP:  CLRA
            LDX     #1
            STX     TEMP

COMP:       CPX     #11
            BEQ     END
            ADD     TEMP
            INCX
            STX     TEMP
            JMP     COMP

END:        STA     $0070

            BRA     MAIN_LOOP
```

```c
/*****************************************************/
/* This demonstration program sums the numbers 1 to 10 */
/* and puts the result into memory.                  */
/*                                                   */
/* This program will be used in EG-151               */
/* Author: Ben Clifford                              */
/* Date: 05/09/19                                    */
/*****************************************************/


#include <hidef.h>      /* for EnableInterrupts macro          */
#include "derivative.h" /* include peripheral declarations     */

typedef unsigned char uint8_t;


/******** Variables ********/

uint8_t i, sum, result;

/***************************/

void main(void)
{
    SOPT   = 0x00;      /* System option register.This line of
                           code disables the watch dog by
                           setting the computer operating
                           properly check to off               */
    ICGC1 = 0x74;       /* set up the clock                    */


    for(;;)
    {
        sum = 0;        /* initialise sum to 0                 */
        i = 1;          /* initialise i to 1 first value to sum*/

        while(i <= 10)  /* The sum is calculated and updated   */
        {               /* in this while loop                  */
            sum = sum + i;
            i = i + 1;  /* increment the loop counter          */
        }

        result = sum;   /* store the result (sum) into the
                           variable result                     */
    }   //end of for

}   // end of main
```


b.r.clifford@swansea.ac.uk

Increment/Decrement a Counter

```
COUNTER=0;

for(;;)
{
    if (PTDD_PTDD2==0)
    {
        COUNTER=COUNTER--;
        SHORT_DELAY(0xFF);
        PTFD=COUNTER;
    }
    else
    {
        COUNTER=COUNTER++;
        SHORT_DELAY(0xFF);
        PTFD=COUNTER;
    }
}
```

```
                CLR     COUNTER

COUNT_LOOP:     LDA     COUNTER
                STA     PTFD
                LDA     PTDD
                AND     #BIT2
                BEQ     DECREMENT
                INC     COUNTER
                BRA     END_LOOP

DECREMENT:      DEC     COUNTER

END_LOOP        JSR     SHORT_DELAY
                BRA     COUNT_LOOP
```


b.r.clifford@swansea.ac.uk

Sum Student Numbers

```
#define STUDENT_COUNT  7
muint8 student_number_1[STUDENT_COUNT] = {1,2,3,4,5,6};
muint8 total_answer;
int counter1,counter2;

for(;;)
{
    counter1 = 0;
    Total_Answer = 0;
    while (counter1 < Student_Count)
    {
        Total_Answer = Total_Answer + student_Number_1[counter1];
        counter1 = counter1 + 1;
    }
    PTFD = Total_Answer;
 }
```

```
                LDHX    #TABLE
                CLRA

NEXT_NUMBER:    ADD     0,X
                AIX     #01
                CPHX    #END_TABLE
                BNE     NEXT_NUMBER
                STA     PTFD

LOOP:           BRA     LOOP

TABLE           FCB     1,2,3,4,5,6
END_TABLE       FCB     0
```


b.r.clifford@swansea.ac.uk

Display Result of ADC Conversion

```
for(;;)
{
    ADC1SC1=8;
    while(ADC1SC1_COCO==0)
    {
    }
    PTFD=ADC1RL;
}
```

```
START_ADC:  LDA   #%00001000
            STA   ADC1SC1

WAIT_ADC:   TST   ADC1SC1
            BPL   WAIT_ADC

            LDA   ADC1RL
            STA   PTFD

            BRA   START_ADC
```


b.r.clifford@swansea.ac.uk

How simple can ‘C’ programs be

```c
/*************************************************************/
/* This demonstration program shows the state of the         */
/* rocker switches by setting the corresponding LED          */
/* on Port F.                                                */
/*                                                           */
/* This program will be used in EG-151                       */
/* Author: Ben Clifford                                      */
/* Date: 05/09/19                                            */
/*************************************************************/


#include <hidef.h>      /* for EnableInterrupts macro                */
#include "derivative.h" /* include peripheral declarations           */

#define leds PTFD       /* LED BAR is connected to Port F            */
#define switches PTAD   /* Rocker Switches are connected to port A   */

typedef unsigned char uint8_t;

/******** Variables ********/

/***************************/

void main(void)
{
    SOPT   = 0x00;      /* System option register.This line of code
                           disables the watch dog by setting the
                           computer operating properly check to off */
    ICGC1 = 0x74;       /* set up the clock                          */

    PTADD = 0x00;       /* define all bits of Port A as inputs.      */
    PTAPE = 0xFF;       /* Enable pullups on Port A.                 */
    PTFDD = 0xFF;       /* define all bits of Port F as ouputs.      */



    for(;;)
    {

        leds = switches;/* send the data on Port A to Port F         */

    }   //end of for

}   // end of main
```

b.r.clifford@swansea.ac.uk

Part III

Exercise 4

11/11/19 – 18/11/19

b.r.clifford@swansea.ac.uk

Aim

- The aim for this week to write and commission the assembly language program that was written in exercise 2 in ‘C’ language.

b.r.clifford@swansea.ac.uk

Starting Point

```
for(;;) {  /* loop forever */

    //student number addition code starts here //

    counter1 = 0;     //initialize counter to zero at the beginning

    total_answer = 0; //initialize total answer to zero at the beginning

    while (counter1 < STUDENT_COUNT)
    {

        total_answer = total_answer + student_number_1[counter1++];  // add the first student number to total answer and
                                                                     // increment counter by 1
    }

    PTFD = total_answer;//send the final answer to the LEDs

    //student number addition code ends here//
}
```


b.r.clifford@swansea.ac.uk

Assessment Criteria

The assessment criteria for exercise 3 will be:

1. Modify the starting program to use a ‘for loop’ in place of the while loop to iterate through the student number.

2. Modify the C program to output the sum of one student number when pressing SW3, and the other student number by pressing SW4 on the demo board.

3. Ability to answer questions on how the programme works, key statements and decisions.

b.r.clifford@swansea.ac.uk