University of Glasgow

UESTC1008: Microelectronic Systems

Academic year 2019/2020 – Semester 2 – Lecture 3

Qammer H. Abbasi, Lei Zhang, Sajjad Hussain and Guodong Zhao
{qammer.abbasi,Lei.zhang,sajjad.Hussain, guodong.zhao}@glasgow.ac.uk

Embedded C vs. Regular C

"A good student never steal or cheat"

Embedded System

- When you press the button on your digital camera to take a photo, the microprocessor will perform the functions necessary to capture the image and store it
- The microprocessor's functions are controlled, guided and overseen by the embedded system software
- Just like your computer is controlled by the Operating System (like Windows or Macintosh), your camera is controlled by the embedded software
- The embedded software and embedded hardware form an embedded system

Embedded C

- Embedded C is the most popular embedded software language in the world
- Embedded C, even if it’s similar to C, and embedded languages in general requires a different kind of thought process to use
- Embedded systems, like cameras or TV boxes, are simple computers that are designed to perform a single specific task
- They are also designed to be efficient and cheap when performing their task

Embedded C

- Embedded systems are supposed to
  - use a low power to operate, and
  - be as cheap as possible
- As an embedded system programmer, you will have simple hardware to work with
- You will have very little RAM, ROM and very little processing power and stack space
- Your goal is to write programs that are able to leverage this limited processing power for maximum effect
- As an ordinary C programmer, you don't have as many constraints

Embedded C

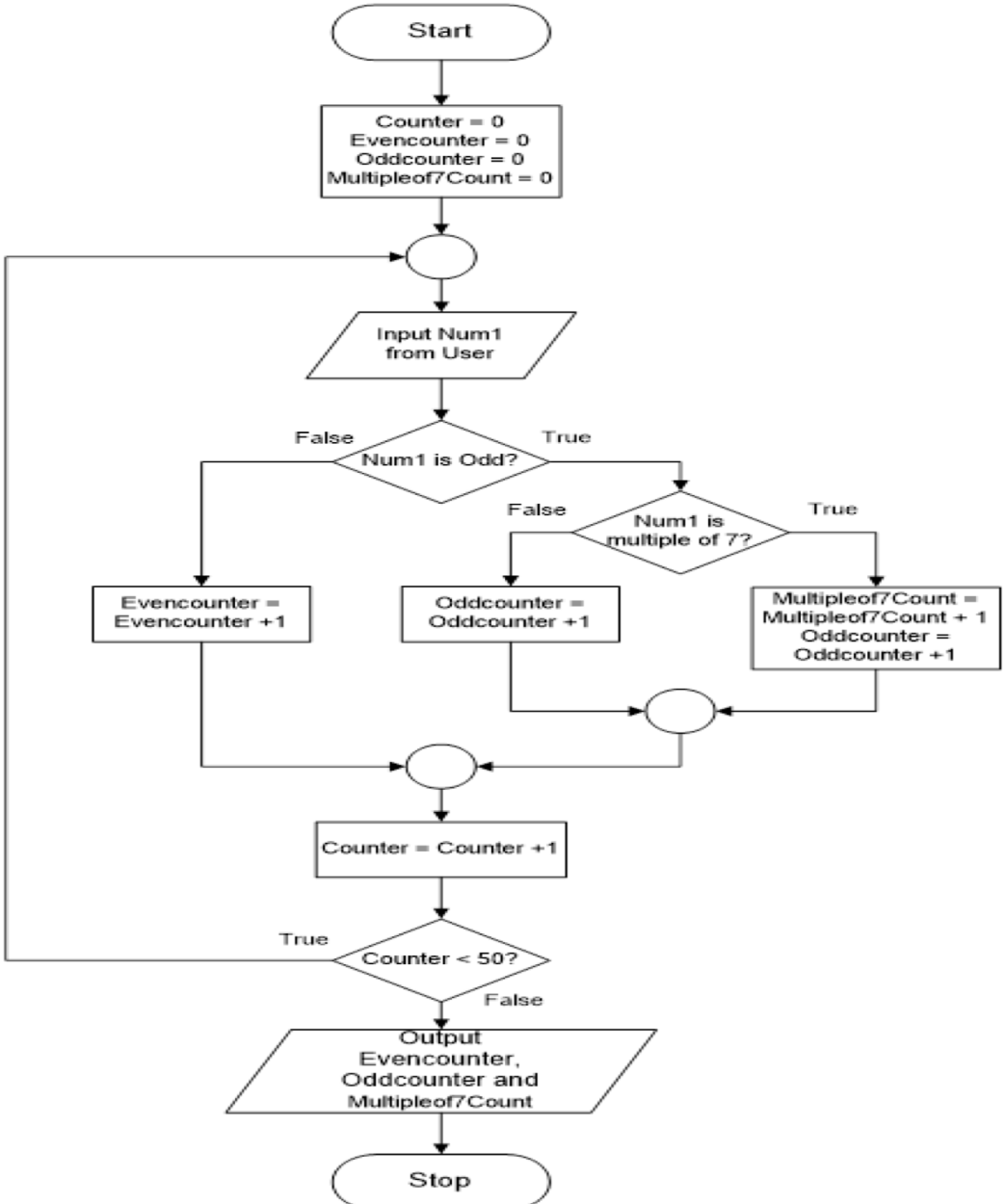
- Embedded C lies somewhere between being a high level language and a low level language
- Embedded C, unlike low level assembly languages, is portable
- It can run on a wide variety of processors, regardless of their architecture
- Unlike high level languages, Embedded C requires less resources to run and isn't as complex
- Some experts estimate that C is 20% more efficient than a modern language like C++
- Another advantage of Embedded C is that it is comparatively easy to debug

Embedded C

- Another major difference between Embedded C and Regular C is the absence of a convential operating system in Embedded Systems
- When you write a regular C program, you access it from within your operating system software, run it and then, when you're done, you exit back into your operating system
- With an Embedded C program, you have no operating system to fall back on!
- Your program will, for all intents and purposes, act like the operating system for the embedded device

Revision on Regular C

Identify the C operations in the flow chart



Identify the C operations in the C code

```
#include <stdio.h>

void main()
{
  int n,  num1;
  int Counter , Evencounter , Oddcounter , Multipleof7Count ;

  Counter = 0;
  Evencounter = 0;
  Oddcounter = 0;
  Multipleof7Count = 0;

  do
  {
     printf("Enter the number = ");
     scanf("%d", &num1);

     if(num1%2 == 1)
     {
        if(num1%7 == 0)
        {
           Multipleof7Count++;
           Oddcounter++;
        }
        else
        {
           Oddcounter++;
        }
     }
     else
     {
        Evencounter++;
     }
     Counter++;
   }while (Counter < 50);

   printf("Evencounter = %d, Oddcounter = %d, MultipleOf7Counter = %d \n",
Evencounter,Oddcounter,Multipleof7Count);

}
```

Comments

- Two ways of commenting are used
  - One is to place the comment between the markers

  /* and ***/**
  - Alternatively, use **//**
  - Comments are for **humans** to read. They are *ignored* by the compiler

```
/*A program which flashes mbed
LED1 on and off. */
```

```
#include “mbed.h”  //include the mbed header file as part of this program
```

Data Types: Character and Integer

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Data Type: Floating Point

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Arithmetic Operators

where A = 10 and B = 20

Operator	Description	Example
+	Adds two operands.	A + B = 30
–	Subtracts second operand from the first.	A – B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Relational Operators

where A = 10 and B = 20

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

where A = 1 and B = 0

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Logical Operators

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

Assignment Operators

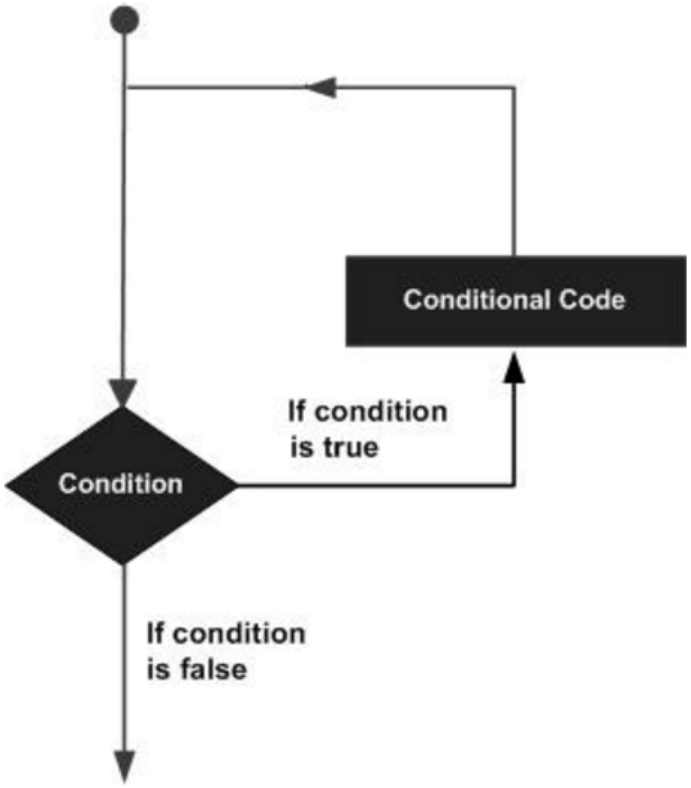
Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A

Assignment Operators (continued)

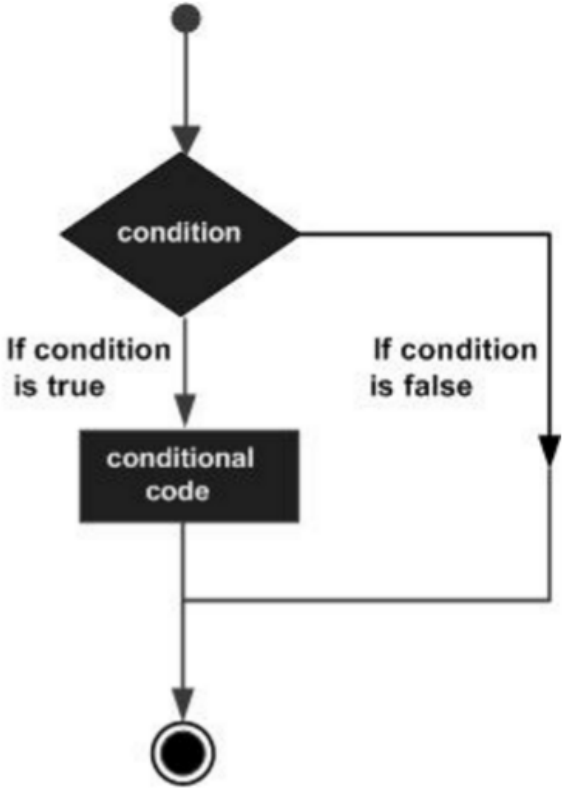
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Conditional Statements

- Loop



- Single Decisions



Loops

Loop Type & Description
while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop It is more like a while statement, except that it tests the condition at the end of the loop body.
nested loops You can use one or more loops inside any other while, for, or do..while loop.

For and *While* Loops

- The ***for*** statement

```
for(expr1; expr2; expr3) statement
```

```
expr1;

while(expr2)
{
statement
expr3;
}
```

- The ***while*** statement

```
while(expression)
statement
```

Infinite Loop

- When the conditional statement is empty, the loop will run forever (or until you turn the power off or hit the Ctrl + C keys).

```c
#include <stdio.h>

int main () {

   for( ; ; ) {
      printf("This loop will run forever.\n");
   }

   return 0;
}
```

Single Decision

Statement & Description
if statement An **if statement** consists of a boolean expression followed by one or more statements.
if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
nested switch statements You can use one **switch** statement inside another **switch** statement(s).

if - else

- The ***if-else*** statement is used to make decisions.

```
if (expression)
statement_1
else
statement_2
```

Switch

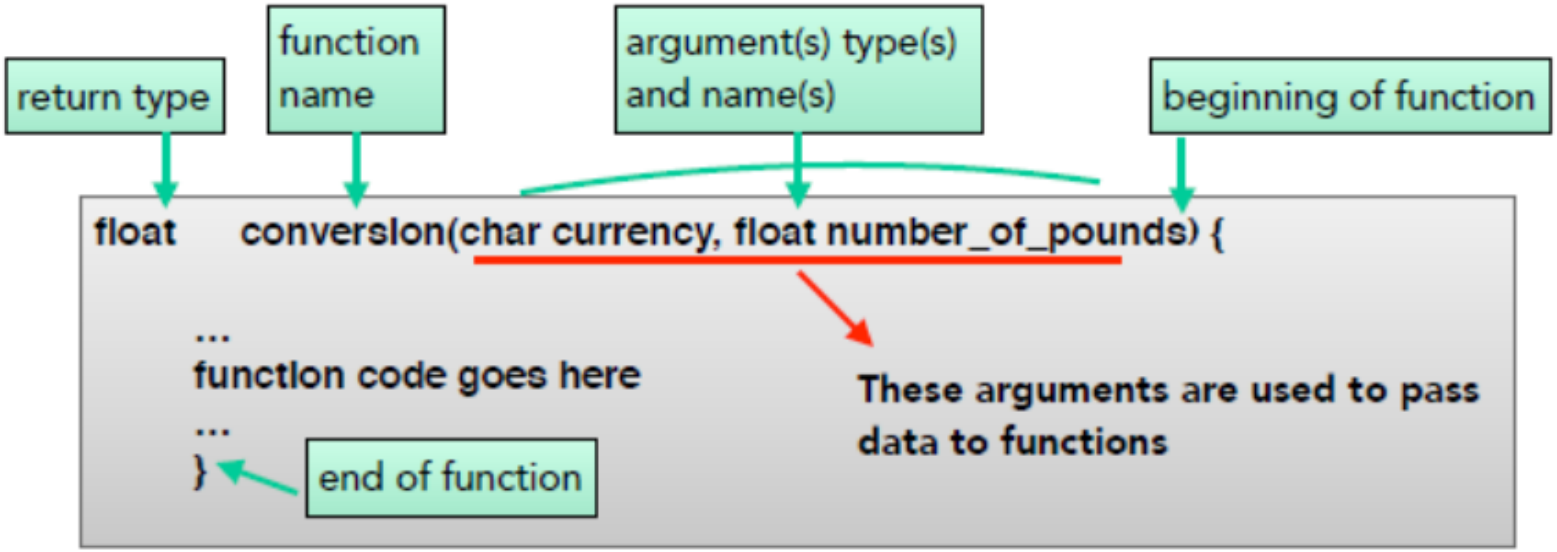
- The ***switch*** statement is special multi-way decision maker that tests whether an expression matches one of a number of constant values, and branches accordingly

```
switch ( i )
{
case 0: display = 1; break;
case 1: display = 2; break;
}
```

where **i** is an integer

Functions

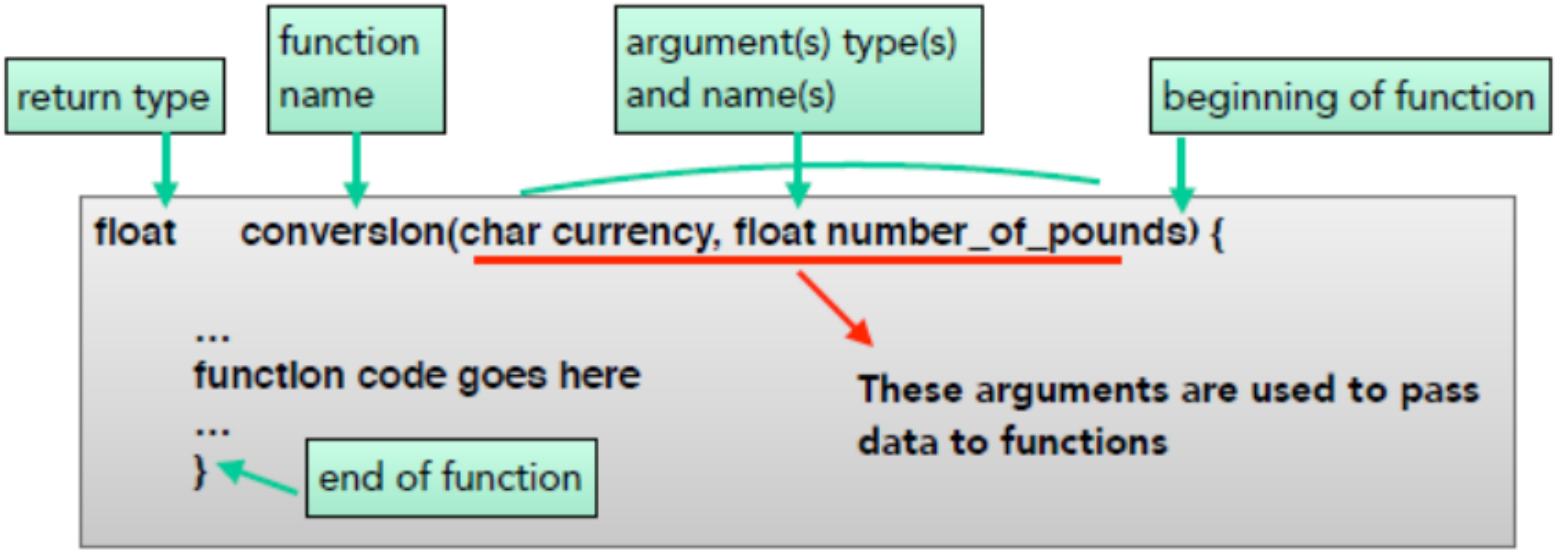
- Function Definitions



- Only one return variable is allowed
- The final statement of the function may be a return, which will specify the value returned to the calling program

Functions

- Function Definitions



- The main function
  - Program execution starts at the beginning of main() and ends at the end of main()
  - Other functions may be written outside main(), and called from within it.

Main Function and Void Return

- Some functions perform the desired operations without returning a value
  - In this case, the **return type** is **void**

- The main function
  - Program execution starts at the beginning of main() and ends at the end of main()
  - Other functions may be written outside main(), and called from within it

Delay Function

- How to **define** a delay function using e.g., for loop

```
void delay(int y)
{
int i = 0;
for(i = 0; i<10000*y; i++)
{
}
}
```

- How to **call** the delay function

```
delay(3);
```

```
delay() = 3;
```

Arrays

- An array is a set of data elements, each of which has the same type
- The declaration of an array

```
int a_1[10];
```

Name of the array:	**a_1**
Number of the elements:	**10**
Elements of the array:	a_1[0], a_1[1], … a_1[9]
Data type of its elements:	**integer**

Structure

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

//A struct declaration defines a type

```
struct size{
int w; int h;
};
```

size: a structure tag (optional)
members: w, h

//Declare instances of the structure struct{int w; int h} size1, size2;
/* If the declaration is tagged, however, the tag can be used in definition of instances of the structure. */
struct size sz;

//Access the members
structure-name.member (e.g., **sz.w**): "." is structure member operator

Structure can Mix Data Types

Datatype	C VARIABLE		C ARRAY		C STRUCTURE	
	Syntax	Example	Syntax	Example	Syntax	Example

int	int a	a = 20	int a[3]	a[0] = 10
				a[1] = 20
				a[2] = 30
				a[3] = '\0'
					struct student
					{
					int a;
					char b[10];
					}
						a = 10
						b = "Hello"

char	char b	b='Z'	char b[10]	b="Hello"		

Typedef

- C provides a facility called typedef for creating new data type names
  - e.g., typedef int Length;

```
typedef struct size{
        int w;
int h;
} Treepoint;
```

This creates a new type keyword called Treepoint (a structure)

- A typedef declaration does not create a new type in any sense; it merely adds a new name for some existing type.

Compiler Directives

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol
  - #define – This macro defines constant value and can be any of the basic data types.
  - #include <file_name> – The source code of the file “file_name” is included in the main C program where “#include <file_name>”

Embedded C for mbed

Compiler Directive — #define directive

- #define directive
  - Define a *symbolic name* or *symbolic constant* to be a particular string of characters.

For example: #define PI 3.14

In LPC17xx.h file:

```
typedef struct
{
….
} GPIO_TypeDef;

#define GPIO0_BASE  constant_value
```

```
#define GPIO0 ((GPIO_TypeDef *) GPIO0_BASE)
```

Compiler directives are messages to the compiler Compiler directives all start with a hash, #

Compiler Directive — #include directive

- #include directive
  - The #include directive directly inserts another file into the file that invokes the directive.
  - e.g., **#include <>** /*used to enclose files held in a directory different from the current working directory*/
  - **#include “mbed.h”** /*Used to contain a file located within the current working directory */

An example of mbed Program

```
/*Example: A program which flashes mbed LED1 on and off. Demonstrating use of
digital output and wait functions. */

#include "mbed.h" //include the mbed header file as part of this program

//program variable myled is created, and linked with mbed LED1
DigitalOut myled(LED1);

    int main() {              //the function starts here

        while(1)              // a continuous loop is created
        {
        myled = 1;    //switch the LED on, by setting the output to logic 1
         wait(0.2);     //wait 0.2 seconds — wait function is from the mbed library
         myled = 0;   //switch the LED off, by setting the output to logic 0
         wait(0.2);     //wait 0.2 seconds
        }
    }
```

An example of mbed Program

```
/*Program Example 3.1: Demonstrates use of while loops. No external connection required
*/
#include "mbed.h"
DigitalOut myled(LED1);
DigitalOut yourled(LED4);

int main() {
  char i=0;          //declare variable i, and set to 0
  while(1){          //start endless loop
  while(i<10) {      //start first conditional while loop
    myled = 1;
    wait(0.2);
    myled = 0;
    wait(0.2);
    i = i+1;         //increment i
  }                  //end of first conditional while loop
  while(i>0)  {      //start second conditional loop
    yourled = 1;
    wait(0.2);
    yourled = 0;
    wait(0.2);
    i = i-1;
  }
    }                //end infinite loop block
}                    //end of main
```

Understanding the mbed API

```
//program variable myled is created, and linked with mbed LED1
DigitalOut myled(LED1);
myled = 1; // this is not a normal = operator
```

*) If you check mbed.h file, you will see this: #include “DigitalOut.h”
*) In DigitalOut.h file, you will know DigitalOut is defined as a class.

Function	Usage
`DigitalOut`	Create a DigitalOut connected to the specified pin
`write`	Set the output, specified as 0 or 1 (int)
`read`	Return the output setting, represented as 0 or 1 (int)
`operator=`	A shorthand for write
`operator int()`	A shorthand for read

Member function of Class **DigitalOut**

how to use member functions: e.g., myled.read()

myled = 1 **is the same as** myled.write(1)

mbed Peripheral components are defined as classes.

Running Programs on mbed Board

- Write C code in Keil uVision or online compiler
  - generate .cpp file
- Compile using uVision or online compiler
  - generate .bin file (machine code)
- Download the machine code
  - Copy .bin to mbed board
- Test on the mbed board

API

- An application programming interface (API) is a set of subroutine definitions, protocols, and tools for building application software
- In general terms, it's a set of clearly defined methods of communication between various software components
- A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer

(Source: https://en.wikipedia.org/wiki/Application_programming_interface)

mbed API

- API documentation is a quick and concise reference containing what you need to know to use a library or work with a program
- It details functions, classes, return types, and more
- In mbed, API documentation for programs and libraries is fully supported both within the Compiler and in the code listings on the public site
- mbed API Link: https://developer.mbed.org/handbook/API-Documentation#extra-features

SDK

- A Software Development Kit (SDK) is a package of pre-written code that developers can re-use in order to minimize the amount of unique code that they need to develop themselves
- SDKs can help to prevent unnecessary duplication of effort in a development community

mbed SDK

- The mbed Software Development Kit (SDK) is a C/C++ microcontroller software platform relied upon by tens of thousands of developers to build projects fast
- The mbed SDK has been designed to provide enough hardware abstraction to be intuitive and concise, yet powerful enough to build complex projects.
- mbed SDK Link: https://developer.mbed.org/handbook/mbed-SDK

Summary

- Embedded C vs normal C
- What will we study in next lecture.