“Wouldst thou,” – so the helmsman answered,
“Learn the secret of the sea?
Only those who brave its dangers
Comprehend its mystery.”

Henry Wadsworth Longfellow, 1850

“Wouldst thou,” – so the helmsman answered,

“Learn the secret of the **C**

Only those who brave its dangers

Comprehend its mystery.”

ENGD1025 Electronics CAE & Programme Fundamentals

C Programming Part 1

“Hello World”

Dr Mike Oliver & Dr Sridhar Govindarajan

Computer Programs

- Contain sequences of instructions
  - Statements
  - Decision making
  - Loops
- Instructions are executed sequentially.
- The computer will do exactly what you program it to do.
- When writing computer programs
  - Break large tasks down into small steps
  - Think logically

Example – Making a Pot of Tea

Sequence of steps:-

Check the water level of kettle

If water level is too low then
- Open Kettle lid
- Place kettle under tap
- Turn tap on
- While the water level is too low
  - Observe water level
- Turn tap off
- Close kettle lid

Connect the kettle to the mains
Turn on switch at mains socket

Depress switch on kettle

While switch is depressed
- Observe status of switch

Open tea pot lid
Place tea bags in tea pot
Pour boiling water into tea pot
Close tea pot lid

Why C?

- Industry standard language
- Ideal for low-level programming
- Good for your C.V.

Books

- “The C Programming Language”
  Kernighan and Ritchie

  A PDF copy may be found on the server in:-
  L:\Jordan
  Filename:
  c.pdf

- “Beginning C for Arduino”
  Jack Purdum

- Many other references.

Hello World

- The famous “Hello World” is one of the simplest programs around.
- Typically the first program encountered when learning to program.
- Don’t underestimate its importance:-
  - It is a working program that does something.
  - It uses some major language constructs

Hello World – Arduino

- The code below is one possible “Hello World” for the Arduino:-

```
// "Hello World" example for Arduino.


void setup()
{
    Serial.begin(115200);   // Starting the serial port at 115200 Baud.
    Serial.print("Hello world\n");
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

- Use the serial port monitor (Tools → Serial Monitor) to observe the output.
- Ensure the Baud rate matches that of the code (i.e. 115200 in this case)
- Have a go!

Arduino Program Structure

An Arduino program (sketch) must always contain two functions:-

- `setup()`
  - This is called once, every time when the program (sketch) starts
- `loop()`
  - Called after `setup()` has been executed.
  - `loop()` is repeatedly executed.

Comments

- The first block of code are simply comments.

```
// "Hello World" example for Arduino.
```

- Everything to the right of the `//` are ignored by the compiler.
- They are used to document code.

- Another example of a valid commenting technique is

```
/* Hello world program */
```

  where the comment is delimited by `/*` and `*/`

Breaking it down

- From a programmers perspective, code starts running from `setup()`
- Code appears in `setup()`. This is executed just once.
- In this example no code appears in `loop()`. `loop()` is run repeatedly.
- There is no screen. Data/messages can be transmitted via the serial port. Data can be received/displayed on the Arduino Serial Port Monitor.
- The first line configures serial data transfer at 115200 Baud (bits/second)

```
Serial.begin(115200);
```

Ensure this Baud rate matches that on the Serial Port Monitor!

- The second line writes a message to the serial port

```
Serial.print("Hello world\n");
```

No `main()`?

- All C programs must have a `main()` function
- Common misconception: The Arduino sketch does not contain a `main()` function therefore it isn't really C?

WRONG!

- The Arduino has an intrinsic `main()` function that is hidden from the programmer. It takes the general (simplified) format:-

```
void main(void)
{
    setup();
    while(1){
        loop();
    }
}
```

ENGD1025 Electronics CAE & Programme Fundamentals

C Programming Part 2

Variable types

Dr Mike Oliver & Dr Sridhar Govindarajan

Overview

- Variable Types
  - Integers
  - Floating point
  - User-defined variables
  - Declaring variables

Variable Types - Integer

In C, there are two standard types used for representing integers

- `char`
  - Stands for character
  - Length: 1 byte
  - Represents integer values in the range of -128 to 127

- `int`
  - Stands for integer
  - Strictly speaking, length of `int` is larger than a `char`
  - Length varies from platform-to-platform (typically 4 bytes for a PC, 2 bytes for a microcontroller).

Variable Types - Integer

To avoid ambiguity there are two derivative types of integer:

`short`	Typically 2-bytes wide
`long`	Typically 4-bytes wide

All these data types are signed and will accept positive and negative values. They have unsigned equivalents (that take positive values only); the type name is preceded by the keyword `unsigned`.

For example `unsigned char` is an unsigned value that can represent values from 0 to 255.

Variable Types – Floating Point

In C, there are two types used for representing floating-point (real) value:-

- `float`
  - Standard type for representing floating-point values
  - 4 bytes wide.

- `double`
  - Derivative type used to represent floating-point numbers with double precision.
  - In theory, represents a much wider range of values than `float`.
  - Some platforms do not properly implement `double`; (On the Arduino platform, `double` masquerades as `float`.)

Variable Types – User Defined

- In C, it is possible to create user-defined data types
- This is achieved using the `typedef` keyword with the following general syntax:-

```
typedef original_variable_type user_variable_type;
```

- For example, to define a user-defined variable called ‘byte_t’ which is essentially an unsigned character:

```
typedef unsigned char byte_t;
```

- From this user definition, `byte_t` is of type `unsigned char`.

Declaring Variables

- To declare a variable in C, the following general syntax is observed:-

```
variable_type variable_name;
```

- For example, declare a character variable called `i`

```
char i;
```

- For example, declare an integer variable called `j`

```
int j;
```

- For example, declare a floating-point variable called `f`

```
float f;
```

- For example, declare a user-defined variable called `byte_val` that represents a byte:-

```
typedef unsigned char byte_t;
byte_t byte_val;
```

Declaring Variables

- It is possible to declare several variables (of the same type) in a single declaration. For example, declare integers x, y and z.

```
int x, y, z;
```

- It is possible to initialize a variable upon declaration. For example declare an integer value count to have an initial value of 10.

```
int count = 10;
```

- Rules for variable names:-
  - Variable name must start with a letter or underscore (_).
  - Variable names can contain a mixture of letters (upper and lower case), numbers and underscores.
  - Sensible approach to give your variables meaningful names.

ENGD1025 Electronics CAE & Programme Fundamentals

C Programming Part 3

Type casting, Arrays and Strings

Dr Mike Oliver & Dr Sridhar Govindarajan

Overview

- Type casting
  - Converting between types
- Arrays
- Strings

Type Casting

- Method of converting from one type to another.
- When performing a type-cast, place the required type name in parentheses before the quantity that needs converting.
- Consider the following example.

```
int a = 24;
float f;
```

- Suppose the value of `a` needs to be stored in `f`.
- As a and f are different types, type casts are necessary.
- Integer to floating point conversion

```
f = (float)a;
```

Type Casting

Example: Converting 8-bit character via type-casting to an integer.

```
char c = 12;
int i;
```

- Casting `c` from a `char` to an `int` :

```
i = (int)c;
```

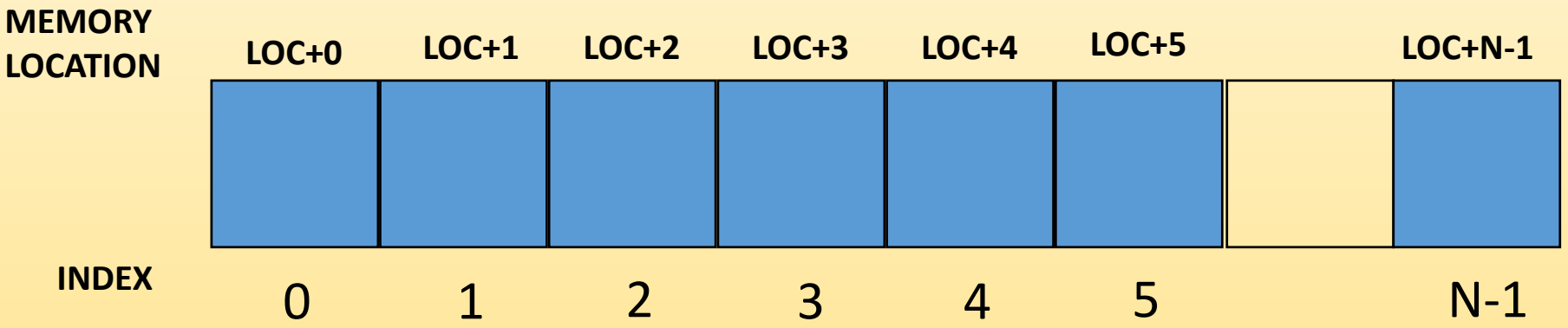
- Some care required when converting larger types to smaller type (for example `long` to `char`) to prevent loss of data due to truncation.

Arrays

- Collection of items stored at contiguous memory locations.
  - Items in an array must be of the same type.
  - Possible to have array of `float` or array of `int`, but not an array of both.
- Application for arrays:-
  - Storage of 'x' and 'y' data for a graph.
  - One array could store time data, the another could store voltage data.

Arrays

- Consider an N-element array of bytes:-



- Suppose the array is stored from the position LOC in memory.
- The first element would be stored at position (LOC).
- The second element would be stored at position (LOC+1).
- The third element would be stored at position (LOC+2).
- The N^{th} element would be stored at position (LOC+N-1).

Arrays

- The relative memory location of each element is dependent upon the size of the type.
- For an array of 4-byte long values:-
  - the first element starts at position (LOC),
  - the second stored from (LOC+4),
  - the third stored from (LOC+8), etc.

- Arrays in C use zero-based indexing. An array starts at position 0.
  - The first element has an index of 0,
  - the second has an index of 1,
  - the third has an index of 2 and
  - the N^{th} has an index of N-1.

Declaring an Array

- An array can be declared in the following generic way:

```
type array_name[size_of_array];
```

- e.g. declaring an integer array called *my_array* containing 5 elements:-

```
int my_array[5];
```

- A value can be assigned to any element in an array using the following generic technique:-

```
array_name[element_number] = value;
```

Populating an Array

- Example: populating the first five elements of an array (indices 0 to 4) with the values 10, 12, 16, 22 and 30.

```
my_array[0] = 10;  // Assigning value 10 to 1st element
my_array[1] = 12;  // Assigning value 12 to 2nd element
my_array[2] = 16;  // Assigning value 16 to 3rd element
my_array[3] = 22;  // Assigning value 22 to 4th element
my_array[4] = 30;  // Assigning value 30 to 5th element
```

- Example: obtaining the third element of an array:

```
Serial.print("The third element is %d\n", my_array[2]);
```

- Populating an array upon declaration:-

```
int my_array[5] = {10, 12, 16, 22, 30};
```

Warning – Keep within Limits!

- Care must be taken when using arrays – keep within the limits of the array.
- C won't throw a run-time error if you go beyond the limits of an array.
- What happens when you go beyond the limits of the array? Consider the 5-element array in previous example.
- Attempt to read element [5] (i.e. the 6th element). This lies outside the bounds of the array. The data being read could be another variable and could be interpreted as garbage.
- Attempt to write element[5]. This could overwrite another variable in memory. In some programming languages the program will terminate gracefully with a run-time error. However, C will not stop a program from writing beyond the limits of the array, overwriting other variables and potentially causing catastrophic effects.
- *You have been warned!*

Strings

- Strings are special cases of arrays – character arrays - arrays of type `char`.
- Example: declaring a string of six characters; call it `my_string`.

```
char my_string[6];
```

- Writing the word "Hello" to the string on a character-by-character basis using the same technique used for setting elements in an array.

```
my_string[0] = 'H';
my_string[1] = 'e';
my_string[2] = 'l';
my_string[3] = 'l';
my_string[4] = 'o';
my_string[5] = '\0';
```

- Each character is delimited by single quotes.
- The string ends with the null character `\0`. This null termination denotes the end of the string and must be explicitly included.

Initializing Strings on Declaration

Initializing strings on a character-by-character basis:-

```
char my_string[6] = {'H','e','l','l','o','\0'};
```

and

```
char my_string[] = {'H','e','l','l','o','\0'};
```

Other methods of initializing strings during declaration include:-

```
char my_string[6] = "Hello";
```

and

```
char my_string[] = "Hello";
```

- Strings are delimited by double-quotes and are implicitly terminated.
- All these methods are only valid during declaration.

ENGD1025 Electronics CAE & Programme Fundamentals

C Programming Part 4

Operators and Program Flow Control

Dr Mike Oliver & Dr Sridhar Govindarajan

Overview

- Operators
  - L- and R-values
  - Mathematical operators
  - Logical Operators
- Program Flow Control
  - Conditional operators
  - Decision making
  - Repetition

OPERATORS

L & R values

L-value

- The entity on the left of an assignment.
- This refers to a memory location.
- An expression that qualifies as a L-value may appear in the left-hand or right-hand of an assignment.

R-value

- The value on the right of an assignment.
- This is a data value stored in memory.
- It cannot have a value assigned to it.
- An expression that qualifies as an R-value may ***only*** appear in the right-hand of an assignment.

L & R Values

Examples of valid assignments:-

```
int a = 20;
int b = a;
c = b + a;
```

Examples of invalid assignments:-

```
10 = 20;          X
30 = b;           X
a + b = c;        X
```

Also consider:-

```
const float pi = 3.1415926536;
```

- Here pi can only appear once in the assignment as an L-value.
- Once the constant is assigned, it cannot be used as an L-value again.

Mathematical Operators (Integer)

- Consider the following integers:-

```
int a = 22;
int b = 7;
int c;
```

Addition: **+ operator**

- e.g. `c = a + b;`
- In this case, the value 29 would be assigned to `c`.

Subtraction:- operator

- e.g. `c = a - b;`
- In this case, the value 15 would be assigned to `c`.

Mathematical Operators

- **Multiplication** *** Operator**
  - e.g. `c = a * b;`
  - In this case, the value 154 is assigned to `c`.

- **Division:** **/ and % Operators**
  - Integer division is more involved.
  - The / operator performs an integer division resulting in an integer quotient.
  - The remainder is found using the % (modulo) operator. For example

```
int quotient, remainder;
quotient = a / b;
remainder = a % b;
```

Result is 3

Result is 1

- Caution! Integer values occupy a finite wordlength.
  - Due to this, the results of mathematical operations could overflow the limits of the integer leading to erroneous results.

Mathematical Operators (Floating Point)

- Consider the following variables:-

```
float d = 22.0;
float e = 7.0;
float f;
```

Addition: **+ Operator**

- e.g. `f = d + e;`
- Here, the value 29.0 would be assigned to `f`.

Subtraction: **- Operator**

- e.g. `f = d - e;`
- Here, the value 15.0 would be assigned to `f`.

Mathematical Operators (Floating Point)

Multiplication: *** Operator**

- e.g. `f = d * e;`
- Here, the value 154.0 would be assigned to `f`.

Division: **/ Operator**

- e.g. `f = d / e;`
- Here, the value 3.142857 would be assigned to `f`.

Mathematical Operators (Floating Point)

- Take care when using integers in floating point calculations. Consider:

```
f = 22 / 7;
```

- Here, the value 22 is an integer, the value 7 is an integer and 22 / 7 is considered an integer division, resulting in the value 3.
- To ensure a floating point calculation takes place, at least one of the values must be a floating point value. For example:

```
f = 22.0 / 7;
```
or
```
f = 22 / 7.0;
```
or
```
f = 22.0 / 7.0;
```

gives the desired result of 3.142857.

Bitwise Operators

- Typically used to manipulate bits in a byte.
- Consider the following definitions and declarations:-

```
typedef unsigned char byte_t;
byte_t a = 128;   // 0x80 in hex ==> 10000000 in binary
byte_t b = 1;     // 0x01 in hex ==> 00000001 in binary
byte_t c = 204;   // 0xCC in hex ==> 11001100 in binary
byte_t d = 85;    // 0x55 in hex ==> 01010101 in binary
byte_t e;
```

Bitwise Operators

Left shift <<

- This shifts a value *n*-places to the left.
- Each left shift is effectively a multiplication by 2.
- Any bits that 'overflow' the word will be lost.
- e.g: shifting `b` 1 place to the left:-

```
e = b << 1;
```

The result stored in e will be 00000010 (i.e. 2).

- e.g. shifting `b` 6 places to the left:-

```
e = b << 6;
```

The result stored in e will be 01000000 (i.e. 64).

Bitwise Operators

Right shift >>

- This shifts a value *n*-places to the right.
- Each right shift is effectively a division by 2.
- Any bits that 'underflow' the word will be lost.
- For example: shifting `a` 2 places to the right:-

```
e = a >> 2;
```

  The result stored in `e` will be 00100000 (i.e. 32).

- For example: to shift `a` 5 places to the right can be performed with:-

```
e = a >> 5;
```

  The result stored in `e` will be 00000100 (i.e. 4).

Bitwise Operators

Bitwise Negation ~

- All bits in a word are inverted, or negated. i.e. each 1 becomes a 0, and each 0 becomes a 1.
- For example:

```
e = ~c;
```

results in 00110011 = 51 being stored in `e`.

Bitwise Operators

- **Bitwise AND &**
- With the bitwise AND operation between two words, corresponding bits are logically ANDed together .
- For example:

```
e = c & d;
```

results in:-

```
  11001100
& 01010101
  --------
  01000100
  --------
```

So the value 01000100 = 68 will be stored in `e`.

Bitwise Operators

Bitwise OR |

- With the bitwise OR operation between two words, corresponding bits are logically ORed together.
- For example:

```
e = c | d;
```

results in:-

```
  11001100
| 01010101
  --------
  11011101
  --------
```

- So the value 11011101 = 221 will be stored in `e`.

Bitwise Operators

Bitwise Exclusive OR (XOR) ^

- With the bitwise Exclusive OR operation between two words, corresponding bits are logically XORed together.
- For example:

```
e = c ^ d;
```

results in:-

```
  11001100
^ 01010101
  --------
  10011001
  --------
```

- So the value 10011001 = 153 will be stored in `e`.

Conditional (Boolean) Operators

- The course of how a program flows depends on results of decisions.
  - A decision will result in 1 if the condition being tested is true.
  - A decision will result in 0 if the condition being tested is false.
- For illustration two integers `a` and `b` will be considered.

Equality ==

- Expression `(a == b)` evaluates to 1 if `a` equals `b`
- Note the double-equals sign. A **common mistake** is to evaluate `(a = b)`.
  - This assigns the value of b to a; the result is 0 if b is 0, or 1 if b is non-zero.

Inequality !=

- Expression `(a != b)` evaluates to 1 if `a` does not equal `b`.

Conditional (Boolean) Operators

- **Less than <**

  Expression `(a < b)` evaluates to 1 if `a` is less than `b`.

- **Greater than >**

  Expression `(a > b)` evaluates to 1 if `a` is greater than `b`.

- **Less than or Equal to <=**

  Expression `(a <= b)` evaluates to 1 if `a` is less than or equal to `b`.

- **Greater than or Equal to >=**

  Expression `(a >= b)` evaluates to 1 if `a` is greater than or equal to `b`.

Conditional (Boolean) Operators

- **Logical AND &&**

  Expression `((a == 3) && (b == 4))`

  evaluates to 1 if `a` is 3 and `b` is 4.

- **Logical OR ||**

  Expression `((a == 3) || (b == 4))`

  evaluates to 1 if `a` is 3 or `b` is 4.

  Also evaluates to 1 if `a` is 3 and `b` is 4.

- **Logical NOT !**

  Expression `!((a == 3) && (b == 4))`

  evaluates to 0 if `a` is 3 and `b` is 4.

PROGRAM FLOW CONTROL

Decision Making (1) - `if`

- The `if` statement is used to determine whether or not a section of code is to be executed.

Simple Decision Making

In its simplest general form:-

```
if (condition is met)
    execute code
```

For example:

```
if (a > b)
    Serial.print("a is greater than b\n");
```

Decision Making (1) - `if`

Single Statement vs Compound Statement

- The following example is fine if just one C-statement needs executing:-

```
if (a > b)
    Serial.println("a is greater than b\n");
```

- If multiple statements need executing, a *compound statement* should be used:-

```
if (a > b)
{
    Serial.println("a is greater than b\n");
    Serial.println("i.e. a > b\n");
}
```

Opening Brace

Indent

Closing Brace

Decision Making (1) - `if`

Either / Or Decisions (if - else)

- The if statement can be extended to execute one block of code or another.

```
if (condition is met)
    execute code block 1
else
    execute code block 2
```

- For example:

```
if (a == 0)
{
    Serial.println("a is zero\n");
}
else
{
    Serial.println("a is non-zero\n");
}
```

Decision Making (1) - `if`

Multiple Decisions (if - else if - else)

- The `if` statement can be extended to execute one of many blocks of code by using `else-if`. As an example:-

```
if (a > 0)
{
    Serial.println("a is positive\n");
}
else if (a == 0)
{
    Serial.println("a is zero\n");
}
else
{
    Serial.println("a is negative\n");
}
```

Use as many `else if` statements as required

Decision Making (2) - `switch`

- The `switch` statement can be neater than multiple `else-if` statements, e.g.

```
switch (variable)
{
case outcome 1:
    code block 1 to be executed;
    break;
case outcome 2:
    code block 2 to be executed;
    break;
case outcome n:
    code block n to be executed;
    break;
default:
    code block n+1 to be executed;
    break;
}
```

Decision Making (2) - `switch`

- Suppose the value of the variable matches outcome 2, then code block 2 will be executed.
- The end of the code block must be terminated with a `break` statement otherwise, the execution will drop to the code associated with the next `case`.
- If the value of the variable does not match any of the outcomes, the `default` code will be executed.

Decision Making (2) - `switch`

- It is also possible to have multiple outcomes associated with the execution of a specific block of code.

```
switch(number)
{
case 1:
case 3:
case 5:
case 7:
case 9:
    Serial.println("Digit is odd\n");
    break;
case 0:
case 2:
case 4:
case 6:
case 8:
    Serial.println("Digit is even\n");
    break;
default:
    Serial.println("Number is not a single digit\n");
}
```

Loops (1) - `while`

- The while loop runs a section of code repeatedly whilst a condition is met

```
while (condition is met)
    execute code;
```

- e.g. Counting from 0 to 100 using a `while` loop:

```
int i = 0;
while (i < 100)
{
    Serial.println(i, DEC);
    i++;
}
```

- The `i++` statement increments the value of `i`. It literally means `i = i + 1`

Loops (1) - `while`

Jumping out of a loop - `break`

- A loop can be terminated using the break statement. e.g. terminate loop when `i` reaches 42.

```
if (i == 42)
{
    break;
}
```

Prematurely commencing a new iteration - `continue`.

- Using the `continue` statement, a new iteration of the loop can be started before the previous iteration is complete. e.g. start a new loop iteration once `i` reaches 42.

```
if (i == 42)
{
    continue;
}
```

Loops (1) - `while`

do-while()

- The do-while structure is an extension of the while loop. It takes the general form:-

```
do
{
    execute code;
} while (condition is met);
```

- Example: an earlier example written as a `do-while` loop:

```
int i = 0;
do
{
    Serial.println(i, DEC);
    i++;
} while (i < 100);
```

- Note: `do-while` executes its associated code at least once.

Loops (2) - `for`

- The `for` loop runs a section of code multiple times.
- Generally used where some code needs running for a defined number of times.
- General form:

```
for (initialization statement(s); condition statement; increment statement)
    execute code;
```

- Example of a `for` loop that is being used to populate to arrays: X and Y.

```
int X[10], Y[10];
int i;
for (i = 0; i < 10; i++)
{
    X[i] = i;
    Y[i] = i * i;
}
```

Loops (2) - `for`

break and continue.

- The `break` and `continue` statements also apply to `for` loops.

Infinite Loops

- Infinite loops can be implemented using a `for`-loop:-

```
for(;;)
{
    // Code for execution
}
```

- Here, there is no start statement, no condition, and no increment. Therefore this effects an infinite loop.

- A `while` statement can also effect an infinite loop:-

```
while(1);    // This condition is always true
```

ENGD1025 Electronics CAE & Programming Fundamentals

C Programming Part 5

Functions

Dr Mike Oliver & Dr Sridhar Govindarajan

Overview

- Functions
  - Introduction to Functions
  - Developing functions
  - void Functions
  - Passing parameters by value and reference

Introduction to Functions

- Functions allow programs to be broken-down into small manageable chunks.
- Functions can be reused within a program.
- **Generally**, functions accept a list of parameters and return a value.

Developing an Example Function

- Example: write a function to calculate voltage given current and resistance values (i.e. using Ohm's Law).
- First step: define a *prototype* for the function.
  - Inform the compiler about the function: name, parameters and return value.
  - Give function prototype a sensible, meaningful name (e.g. `CalcVoltage`)
  - Define the arguments (parameters)

    e.g. “current” and “resistance (Use `float` as they are unlikely to be integers)
- Sensibly, the result of an Ohm's Law calculation should also be floating point; the function should return `float`.
- Finally, the prototype is terminated by a semicolon. Hence:

```
float CalcVoltage(float current, float resistance);
```

Developing an Example Function

- Second step: Once prototype is defined, implement the function.
  - Copy the prototype.
  - Replace the semicolon with pair of curly braces (body of function resides between these braces)
  - Implement the code (e.g. in this case, perform the calculations)
  - Use `return` statement to cause function to exit and return a value (e.g. in this case voltage)
  - Result is below:-

```
float CalcVoltage(float current, float resistance)
{
    float voltage = current * resistance);
    return voltage;
}
```

Developing an Example Function

- Final step: Test and use the new function.
- In this example, the user enters current (I) and resistance (R) values.
- The function `CalcVoltage()` is *called* using the values `I` and `R`.
  - The value of `I` is copied into current, and
  - The value of `R` is copied into resistance

  When the function exits, the calculated value of voltage is returned then stored in `V`.
- When the function exits, the calculated value of voltage is returned then stored in V.

```
float I = 0.002;
float R = 470;
float V;

V = CalcVoltage(I, R);
Serial.print("The voltage (in Volts) is ");

Serial.println(V);
```

`void` Functions

- In terms of mathematics, a function takes a value, process it and generates another value.
- In C it is possible to have functions that do not return a value.
  - These are known as void functions.
  - Their return type is `void`.
- Example: a very simple function that does not return a value:-

```
void Hello()
{
    Serial.println("Hello world");
}
```

- Calling this function, i.e.

```
Hello();
```

generates a "Hello world" message. No value is returned.

Passing Parameters By Value

- In C data can be passed into functions using **value parameters**.
- A value, or a **copy** of a variable is passed into the function.
- Example: Consider the following function called `power`. This takes a value called `number` and raises it to the power given by the value of `exponent`. The result is returned.

```
float power(float number, int exponent)
{
    float result = 1;
    while (exponent >= 1)
    {
        result = result * number;
        exponent--;
    }
    return result;
}
```

Passing Parameters By Value

- The function can be called from another part of the program. Consider:

```
float res;
float num = 3.0;
int exp = 4;
res = power(num, exp);
```

- A *copy* of the value `num` (i.e. 3.0) will be passed into number.
- A *copy* of the value `exp` (i.e. 4) will be passed into exponent.
- The function will execute and return the value 81 into res.

Passing Parameters By Reference

- To pass an actual variable into a function by reference, the **address** of the object is passed as the actual parameter (using the address operator &) .
- The corresponding formal parameter is declared as a **pointer**.
- Inside the function, the name of the formal parameter is dereferenced and prefixed with the indirection operator * to access the memory allocated to the object.
- Example: the following function swaps the contents of a pair of integer (int) operators:-

```
void swap_int(int *value1, int *value2)
{
    int temp_value;

    temp_value = *value1;
   *value1 = *value2;
    *value2 = temp_value;
}
```

Passing Parameters By Reference

This example shows how the function would be called:-

```
int x = 4;
int y = 7;
swap_int(&x, &y);
```

- Address operator `&` passes the addresses of `x` and `y` to the function `swap_int()` as opposed to a copy of their values. The function can then access `x` and `y`.
- `*value1` accesses the memory allocated to variable `x`,
- `*value2` accesses the memory allocated to variable `y`.