INTRODUCCION A LA PROGRAMACION DEL ARDUINO

Referencia del lenguage C en Arduino *(Recopilado 17-6-19)*

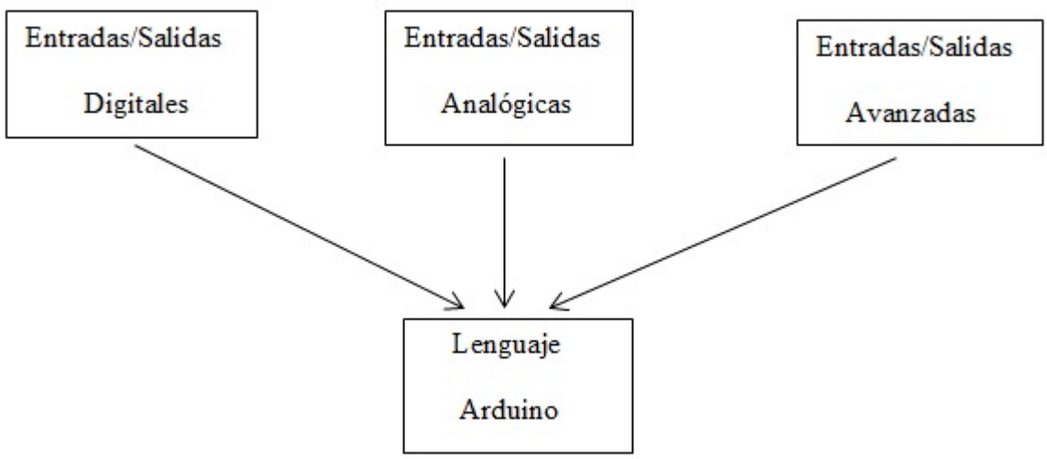
Si no has programado nunca no te preocupes: formas parte del 99.5% de la humanidad que nunca ha escrito una línea de código. El propósito de este tutorial es iniciarte en los rudimentos de la programación del Arduino de una forma sencilla y asequible. Recuerda que, como dijo Steve Jobs, “todo el mundo debiera saber programar porque programar te enseña a pensar”.

1. UN POCO DE HISTORIA

El lenguaje del Arduino está basado en el mítico lenguaje C. Si ya has trabajado en C este tutorial te parecerá un paseo. Si no, te basta con saber que C es el lenguaje en el que se ha desarrollado los sistemas operativos UNIX, Linux, y cientos de sistemas, programas y aplicaciones de ordenador. El lenguaje del Arduino es una versión reducida y mucho más sencilla de manejar que el lenguaje C. El objetivo de este lenguaje es que puedas programar de una manera intuitiva concentrándote en lo que quieres hacer más que en la manera de hacerlo.

Arduino fue desarrollado originalmente en el Interactive Design Institute en Ivrea (Italia) donde los estudiantes estaban familiarizados con un lenguaje llamado Processing. Este lenguaje estaba orientado a estudiantes de arte y diseño y utilizaba un entorno de desarrollo visual e intuitivo en el cual se basó el entorno de desarrollo del Arduino y su lenguaje de programación.

Trabajar con un Arduino consiste fundamentalmente en interactuar con los diferentes puertos de entrada y salida del Arduino. A fin de evitar al programador el engorro y la complejidad de programar estos puertos (ya sean analógicos, digitales o de cualquier otro tipo) el lenguaje de Arduino usa una serie de librerías (de las que no te tienes que preocupar ya que forman parte del lenguaje, ya las iremos viendo con detenimiento más adelante). Estas librerías te permiten programar los pins digitales como puertos de entrada o salida, leer entradas analógicas, controlar servos o encender y apagar motores de continua. La mayor parte de estas librerías de base (“core libraries”) forman parte de una macro librería llamada Wiring desarrollada por Hernando Barragán.



Cada vez que un nuevo puerto de entrada o salida es añadido al Arduino, un nuevo conjunto de librerías especializadas en ese puerto es suministrada y mantenida por los desarrolladores del nuevo puerto.

2. VARIABLES

Programar consiste básicamente en decirle a tu Arduino y a los actuadores que este controla desde sus puertos (o “shields”) lo que tiene que hacer (*o esperamos que haga, todos los programadores saben que estas son cosas frecuentemente diferentes*).

Un programa (o “sketch” en la jerga Arduino) consigue este objetivo fundamentalmente mediante el procesamiento más o menos complejo de datos y la transmisión de estos datos procesados a los actuadores. Lo que llamamos variables es simplemente una manera de codificar o representar estos datos dentro del sketch para facilitar su manipulación de cara a su transmisión hacia o desde los actuadores/sensores.

Desde un punto de vista práctico, podemos considerar las **variables** como los cajones de un escritorio, cada uno tiene una etiqueta describiendo el contenido y dentro de él se encuentra el valor de la variable (el contenido del cajón). Hay tantos tipos de variables como de datos: números de todo tipo representados de diferentes maneras (enteros, reales, binarios, decimales, hexadecimales, etc.), textos (de un solo o varios caracteres o líneas), matrices (arrays), constantes, etc.

2.1 TIPOS DE VARIABLES

El lenguaje de Arduino maneja los siguientes tipos de variables:

TIPO	DESCRIPCIÓN	EJEMPLO
void	Reservado para la declaración de funciones sin valor de retorno.	*void setup()void loop()*
byte	Un número entero del 0 al 255 codificado en un octeto o *byte* (8 bits)	*byte testVariable = 129;*
int	(*Int*eger=entero). Un número entero entre 32,767 y -32,768 codificado en dos octetos (16 bits)	*int testVariable = 28927;*
long	Un entero comprendido entre 2,147,483,647 y – 2,147,483,648 y codificado en 32 bits (equivalente a 4 bytes/octetos).	*long testVariable = 67876;*
float	Un número real (con decimales) almacenado en 4 bytes (es decir 32 bits) y comprendido entre 3.4028325E+38 y -3.4028325E+38	*float testVariable = 3.56;*
unsigned int	Un número natural (entero positivo) almacenado en 16 bits (2 bytes) y comprendido entre 0 y 65,545	*unsigned int testVariable = 38948;*
unsigned long	Un número natural (entero positivo) almacenado en 32 bits (4 bytes) y comprendido entre 0 y 4,294,967,296	*unsigned long testVariable = 657456;*
word	Lo mismo que *unsigned int*	*word testVariable = 51000;*
boolean	Una variable booleana que puede tener solamente dos valores: *true* (verdadero) o *false*	*boolean testVariable = true;*
char	Un carácter ASCII almacenado en 8 bits (un byte). Esto permite almacenar caracteres como valores numéricos(su código ASCII asociado). El código ASCII para el carácter ‘a’ es 97, si le añadimos 3 obtendríamos el código ASCII del carácter ‘d’	*char testVariable = ‘a’;* *char testvariable = 97;*
unsigned char	Este tipo de datos es idéntico al tipo *byte*explicado arriba. Se utiliza para codificar números de 0 hasta 255. Ocupa 1 byte de memoria.	*unsigned char testUnCh = 36;*

En el lenguaje de Arduino cuando queremos utilizar una variable primero hay que declarar el tipo de variable de la que se trata (por ejemplo '*int*' y luego el nombre que le queremos dar a esa variable ('testVariable' en los ejemplos de la tabla anterior).
Podemos dejar la variable sin inicializar (es decir, sin asignarle un valor de partida):

```
int comienzo;
```

o, si nos interesa asignarle un valor inicial:

```
int comienzo = 0;
```

Os aconsejamos inicializar siempre vuestras variables en el momento de declararlas. Esto os puede ayudar a depurar vuestros sketches y al mismo tiempo ahorra código. Asimismo, al declarar una nueva variable tratad de anticipar el uso que el sketch va a darle a esa variable y el rango de valores que va a tomar durante la ejecución (por ejemplo si va a sobrepasar el valor 32,000 interesa hacerla *long* en vez de *int*, o si va a tomar valores decimales entonces necesitaremos una *float*). De no hacerlo así podríamos encontrarnos con situaciones inesperadas durante la ejecución del sketch.
Ejemplo de uso de variables en un sketch:

Veamos el típico sketch que hace parpadear un LED activado a través de un pin:

```
int LEDpin = 6;    //la variable LEDpin se inicializa a 6,es decir vamos a activar el pin 6


void setup(){
    pinMode(LEDpin,OUTPUT);
   }
void loop(){
            digitalWrite(LEDpin,HIGH);
            delay (1000);
            digitalWrite(pinLED,LOW);
            delay (1000);
   }
```

El uso de variables nos permite reutilizar este código para otro pin con tan sólo cambiar la asignación inicial de la variable LEDpin.

No te asustes si no entiendes todo en este código. De momento basta con que comprendas el uso de las variables. Todo lo demás lo iremos viendo en detalle más adelante. Nota que cada instrucción debe de terminarse con un punto y coma (;) de lo contrario el compilador no entendería que la instrucción ha acabado y nos daría un error.

2.2 ARRAYS

Un 'array' es una colección indexada (como un diccionario) de variables del mismo tipo. En el caso de un array el índice no es una palabra como en el diccionario, sino simplemente un número (que corresponde al número de orden de la variable concreta dentro del array). Puedes declarar un array de la siguiente manera:

```
int miLista[6];
```

Si deseas inicializarlo al mismo tiempo puedes hacerlo de la siguiente manera:

```
int miLista[6] ={1,2,3,4,5,6} ;
```

- Nota importante 1: ten en cuenta que el primer índice de un array es siempre 0 (no 1).
- Nota importante 2: ten especial cuidado de no tratar de acceder datos fuera del array (por ejemplo en en caso anterior *miLista [7]* ya que esto nos devolvería datos sin sentido de la memoria.)

Ejemplos de operaciones con arrays:

```
int minuevaLista[4] ={1,2,3,4} ;
nuevaVariable = minuevaLista[2];
minuevaLista[0] = 986;
nuevaVariable = minuevaLista[0];
```

La primera instrucción crea e inicializa el array con 4 valores. La segunda crea una nueva variable y le asigna el valor de la tercera variable del array que habíamos acabado de crear (el int 3). La tercera asigna a la

primera variable del array el valor entero (int) 986. Por último la cuarta instrucción asigna a la variable que habíamos creado en la segunda línea el valor almacenado en la primera variable del array (986).

2.3 STRINGS

Un string es una cadena (array) de caracteres. Los strings pueden ser declarados de dos maneras diferentes: como variables tipo '*char*' o como objetos pertenecientes a una clase más compleja llamados ´*Strings*´ de los que hablaremos al final de este capítulo. De momento nos ceñimos a las cadenas de caracteres definidas como variables '*char*'.

2.3.1 Strings como cadenas o arrays de caracteres:

Cuando una cadena de caracteres es almacenada en una variable tipo *char* normalmente se les hace terminar por el carácter nulo (ASCII 0). Esto se hace con el fin de permitir a las funciones que manipulan cadenas de caracteres en variables *char* reconocer el final de la cadena. Por lo tanto, cuando calcules el tamaño de una cadena dentro de un array *char* no te olvides de añadir una posición para el carácter de 'fin de cadena´ (ASCII 0).
Un array char puede ser declarado de diversas maneras:

```
char polichori[10];
char polichori[8]= {'C','h','o','r','i','z','o'} ;
char polichori[8]= {'C','h','o','r','i','z','o','\0'} ;
char polichori[]= "Chorizo";
char polichori[8]= "Chorizo";
char polichori[10]= "Chorizo";
```

En la primera línea creamos un *array* sin incializar. En la segunda lo dimensionamos e inicializamos y el sistema añade automáticamente un carácter nulo (ASCII 0 ó \0) al final del *array*. En la tercera añadimos el carácter de fin de *array* explícitamente. En la cuarta línea el *array* se fracciona en caracteres individuales y se dimensiona automáticamente. En la quinta el *array* se fracciona automáticamente y en la sexta dejamos espacios libres al final del *array* para nuevos datos.

2.3.2 Strings como objetos de una clase: la variable tipo '*String*':

Como hemos señalado antes podemos definir una variable de tipo String (con la "S" inicial en mayúscula), en vez del tipo "*char*" (carácter) que acabamos de ver. La ventaja de este tipo de variable es que (a pesar de ser más pesada para el sistema por su uso de la memoria y mayor complejidad de procesamiento) ofrece muchas más posibilidades al programador.
Al declarar una variable de tipo "*String*" podemos realizar sobre ella una serie de operaciones complejas de una manera muy sencilla: añadir caracteres a la variable, concatenar Strings, calcular su longitud, buscar y reemplazar substrings y mucho más.
Algunos ejemplos de cómo se declara y utiliza una variable del tipo ´String´:

```
String stringUno = "Hola String";         //declarando un String "constante"
String stringUno =  String('a');        // convirtiendo una constante tipo char en un String
```

En el módulo "Funciones Arduino" veremos como opera la función *String()* y como podemos realizar todas las operaciones con strings que hemos citado anteriormente

2.4 CONSTANTES

Algunas variables no cambian de valor durante la ejecución del sketch. En estos casos podemos añadir la palabra reservada '*const*' al comienzo de la declaración de la variable. Esto se utiliza típicamente para definir números de pin o constantes matemáticas (pi,e,etc…).

```
const int NoPinLed = 12;
const float pi = 3.1416;
```

Si tratas de asignar un valor a una constante más adelante en el sketch, el compilador te advertirá de este error mediante un mensaje.

El Arduino tiene una serie de palabras reservadas que son constantes:

- INPUT/OUTPUT (Entrada/Salida). Los pins digitales pueden ser configurados de ambos modos: como entrada (INPUT) o como salida (OUTPUT) mediante la función *pinMode()*: *pinMode(13, OUTPUT);* // Configura el pin 13 como salida digital.
- INPUT_PULLUP: Reservado como parámetro de la función *pinMode()* para el uso de resistencias pull-up integradas en el chip Atmega del Arduino.
- LED_BUILTIN: para el uso del Led de serie con el que viene equipado el Arduino (generalmente conectado al pin digital 13).
- TRUE/FALSE (Verdadero/Falso). Para el Arduino True (Verdadero) es cualquier valor que no es 0. False (Falso) es el valor 0.
- HIGH/LOW (Alto/Bajo). Es el valor lógico en una puerta digital: LOW es el valor 0 -0 Voltios- y HIGH es el valor 1 -5 Voltios-

Constantes tipo *int*:

Se trata de constantes usadas directamente en un sketch, como por ejemplo 123. Por defecto, estos números se tratan como enteros (*int*) pero esto puede cambiarse con los modificadores U y L (ver más abajo).
En general, se consideran las constantes enteras se formulan en base 10 (es decir, como números decimales). Sin embargo, se pueden utilizar notaciones especiales para expresar números en otras bases.

BASE	EJEMPLO	FORMATEADOR	COMENTARIO
10 (decimal)	123	ninguno	
2 (binario)	B1111011	prefijo 'B'	Máximo 8 bits (0 to 255), Solo caracteres 0-1 válidos
8 (octal)	0173	prefijo "0"	characters 0-7 valid
16 (hexadecimal)	0x7B	prefijo "0x"	characters 0-9, A-F, a-f valid

Decimal es base 10. Esta es la base en la que comúnmente trabajamos. Las constantes expresadas sin ningún prefijo se consideran como decimales.

Binario es base 2. Sólo los caracteres 0 y 1 son válidos.

Ejemplo:

```
B101    // equivale al decimal 5:  ((1 * 2^2) + (0 * 2^1) + 1)
```

El formato binario (B) solo funcionan con on bytes (8 bits) entre 0 (B0) y 255 (B11111111). Si necesitas utilizar un entero (*int* de 16 bits) en forma binaria, lo piuedes hacer con un procedimiento en dos pasos como el siguiente:
*miInt = (B11001100 * 256) + B10101010;   // B11001100 es el byte de orden superior*
Números octales están representados en base ocho. Sólo los caracteres del 0 al 7 son válidos. Valores octales se identifican con el prefijo "0" (cero).

Ejemplo:

```
0101    // Equivale al número decimal  65:  ((1 * 8^2) + (0 * 8^1) + 1)
```

Nota:

Es posible inducer en error al compilador incluyendo por accidente un cero como prefijo de una constante que el compilador va a interpreter como un número octal.

Hexadecimal (o hex) se refiere a numeros en base 16. Los caracteres válidos para estos numeros son las cifras del 0 al 9 y las letras A a la F; A equivale a 10, B a 1 y así hasta la F (15). Valores Hex values se identifican con el prefijo "0x". Las letras de la A a la F pueden ser escritas tanto en mayúsculas como en minúsculas (a-f).

Ejemplo:

```
0x101   // Equivale al número decimal 257 decimal   ((1 * 16^2) + (0 * 16^1) + 1)
```

Formateadores U y L:

Generalmente, una constant entera es tratada como un *int* con las consiguientes limitaciones en cuanto a valores. Para especificar que la constant entera tiene otro tipo hay que añadirle el sufijo:

- 'u' o 'U' para que el formato de la constante sea "*unsigned*". Ejemplo: 33u
- 'l' o 'L' para que el formato de la constante sea "*long*". Ejemplo: 100000L
- 'ul' o 'UL' para que el formato de la constante sea "*unsigned long*". Ejemplo: 32767ul

Constantes tipo coma flotante (*float*):

Del mismo modo que las constantes enteras, las constantes en coma flotante se usan para hacer el código más legible. Estas constantes se reemplazan durante la compilación por el valor asignado en la expresión declarativa.

Ejemplo:

n = .005;

Las constantes en coma flotante pueden también ser expresadas en una variedad de notaciones científicas. Ambos caracteres 'E' y 'e' pueden ser utilizados como indicadores de exponente.

CONSTANTE EN COMA FLOTANTE	EQUIVALE A	Y TAMBIÉN A
10.0	10	
2.34E5	$2.34 + 10^5$	234.000
67e-12	$67,0 * 10^{-12}$	. 000000000067

2.5 ÁMBITO DE LAS VARIABLES

2.5.1 Variables locales y globales

En todos los lenguajes estructurados como el del Arduino las variables tienen una propiedad llamada "ámbito" ("scope" en inglés). Esta propiedad se refiere al hecho de que el valor de la variable puede ser accedido o no en unas partes del sketch según cómo y dónde esta variable haya sido definida.

Aquellas variables que hayan sido definidas fuera de las funciones *setup* y *loop* (veremos que representan estas funciones más tarde cuando hablemos de las funciones y las estructuras) se denominan variables *globales* y su valor puede ser accedido desde cualquier punto del programa o *sketch*.
Por el contrario, aquellas variables definidas dentro de una función se llaman variables locales y tienen un ámbito (scope) local: sus valores son accesibles solamente desde dentro de la función en la que han sido declaradas.

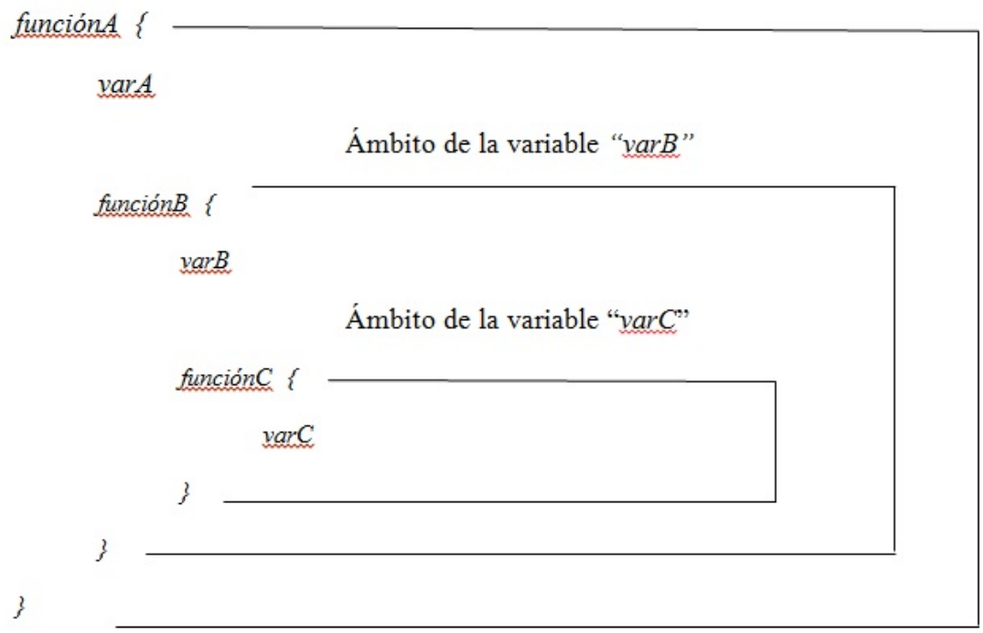
El siguiente sketch muestra el uso de variables globales y locales:

```
int pinNoLedGlobal = 12;     (pinNoLedGlobal definida como variable global)
void Setup       {
       pinMode(pinNoLedGlobal, OUTPUT);
}
void loop         {
       int pinNoLedLocal =13;   //pinNoLedLocal definida como variable local dentro de loop
pinMode(pinNoLedLocal, OUTPUT);
       digitalWrite(pinNoLedGlobal, HIGH);
       digitalWrite(pinNoLedLocal, LOW);
}
```

A medida que tus programas (sketches) crecen en tamaño y complejidad, el uso de variables locales es aconsejado porque evita confusiones entre variables con el mismo nombre.

En este punto podéis preguntaros que sucede con el ámbito de las variables en el caso de variables declaradas dentro de funciones anidadas (una función llamada desde otra función, que a su vez ha sido llamada desde otra función). Lógicamente, él ámbito de aquellas variables definidas en las funciones exteriores se extiende a las funciones interiores, pero no viceversa.

La siguiente figura ilustra este concepto:



El valor de la variable "*varA*" puede ser accedido desde las funciones "*funciónA*", "*funciónB*" y "*funciónC*" es decir, es una variable global. En cambio las variables "*varB*" y "*varC*" son variables locales. El valor de la variable "*varB*" sólo puede ser accedido desde las funciones "*funciónB*" y "*funciónC*", mientras que la variable local "*varC*" sólo puede ser leída desde la función "*funciónC*".

2.5.2 Variables estáticas

La palabra reservada (keyword) "*static*" se usa para crear variables que son visibles únicamente para una function. La diferencia con las variables locales es que éstas son creadas y destruidas cada vez que se llama a una función. Las variables definidas como "*static*" persisten cuando la function ha terminado y conservan los datos almacenados en ellas disponibles para la próxima vez que se llame a esa función. Estas variables se crean e inicializan solamente la primera vez que la función que las crea es llamada.
Ejemplo:

```
/* RandomWalk
* Paul Badger 2007
* RandomWalk wanders up and down randomly between two
* endpoints. The maximum move in one loop is governed by
* the parameter "stepsize".
* A static variable is moved up and down a random amount.
* This technique is also known as "pink noise" and "drunken walk".
*/
#define randomWalkLowRange -20
#define randomWalkHighRange 20
int stepsize;

int thisTime;
int total;

void setup()
{  Serial.begin(9600);
}

void loop()
{        //  tetst randomWalk function
```

```
     stepsize = 5;
     thisTime = randomWalk(stepsize);
     Serial.println(thisTime);
     delay(10);
}

int randomWalk(int moveSize){
  static int  place;     // variable to store value in random walk - declared static so that it
stores
                         // values in between function calls, but no other functions can change its
value
  place = place + (random(-moveSize, moveSize + 1));
  if (place < randomWalkLowRange)
   {  // check lower and upper limits
      place = place + (randomWalkLowRange - place); // reflect number back in positive direction
   }
  else if(place > randomWalkHighRange){
       place = place - (place - randomWalkHighRange);     // reflect number back in negative
direction
  }
return place;
}
```

2.5.3 Variables volátiles

Cuando declaramos una variable como "*volatile*" lo que estamos haciendo es instruyendo al compilador a cargar siempre la variable desde la memoria RAM y no desde uno de los registros internos del procesador. La razón por la que hacemos esto es que, bajo ciertas condiciones, el valor de una variable almacenada en los registros internos puede sufrir alteraciones. En Arduino el único sitio en el que esto puede suceder es en las las partes de código asociado con interrupciones (*interrup service routines*).
Ejemplo

```
// conmuta un LED cuando el pin de interrupción cambia de estado
int pin = 13;
volatile int state = LOW;
void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}

void loop()
{
  digitalWrite(pin, state);
}

void blink()
{  state = !state;
}
```

2.6 MARCADORES SINTÁCTICOS

Antes de proseguir vamos a explicar algunos marcadores que hemos visto en ejemplos anteriores sin explicarlos debidamente:

2.6.1 Fin de instrucción: ";"

Habrás observado que todas las declaraciones e instrucciones terminan en un punto y coma:

```
int a = 13;
```

Esta es una convención sintáctica que conviene no olvidar si no queremos tener que vérnoslas con un impenetrable y aparentemente ilógico error del compilador.

2.6.2 Llaves (“*curly brackets*”) “{}”

Las llaves de apertura y cierre son una parte importantísima de la sintaxis del lenguaje de programación de C. Se utilizan en diversas estructuras (que veremos a continuación) y son frecuentemente una fuente de dolores de cabeza para los principiantes.

La llave de apertura “{” debe de ser complementada siempre por una llave de cierre “}”. El IDE de Arduino IDE (entorno de desarrollo integrado) incluye una función que nos ayuda a comprobar este equilibrio entre llaves de apertura y de cierre. Simplemente con seleccionar una llave de apertura/cierre, el IDE nos indicará donde está la llave de cierre/apertura que “casa” con la seleccionada (“*logical companion*”). No te fíes demasiado de esta funcionalidad del IDE ya que a veces se puede equivocar y señalar una “compañera” equivocada.
Dado que el uso de las llaves es tan variado y extendido en C, se recomienda escribir la llave de cierre justo después de la de apertura para ir “rellenando” el interior con código y evitar así olvidarnos de escribir la llave de cierre al final (piensa que probablemente vas a escribir bucles, funciones, etc. dentro de estas llaves que llevaran a su vez llaves de apertura y cierre. De esta manera evitas olvidos que te llevarían a errores bastante impenetrables del compilador que pueden a veces ser difíciles de localizar en un programa grande. Por otra parte, ten en cuenta que las llaves de apertura y cierre son el marcador sintáctico más importante del lenguaje y su uso en una posición incorrecta pueden llevarte a una ejecución del programa totalmente diferente de la que habías previsto.

Un pequeño resumen de los usos más frecuentes de las llaves de apertura y cierre:

En funciones:

```
void myfunction(datatype argument){
    instruccion(es)
  }
```

En bucles:

```
while (boolean expression)
  {
     instruccion(es)
  }
  do
  {
     instruccion(es)
  } while (boolean expression);

  for (initialisation; termination condition; incrementing expr)
  {
     instruccion(es)
  }
```

En instrucciones condicionales:

```
if (boolean expression)
  {
      instruccion(es)
  }
else if (boolean expression)
  {
     instruccion(es)
  }
  else
  {    instruccion(es)
  }
```

2.6.3 Comentarios dentro de un programa (documentando nuestros programas)

Habrás notado que muy frecuentemente añadimos comentarios al código de un sketch o programa con el efecto de hacerlo más legible, fácil de entender (y mantener) por otro programador, en definitiva, mejor documentado.

Generalmente los comentarios se utilizan para informar sobre la manera en que trabaja el programa (a otros programadores, o recordase a sí mismo cuando tienes que modificar un programa que escribiste hace tiempo) Los comentarios son ignorados por el compilador y no se exportan al procesador, así que no toman ningún espacio en el microprocesador ni en la memoria de ejecución. Dependiendo de su extensión, hay dos maneras diferentes de formular comentarios:

```
x = 5;     // Esto es un comentario de una sola línea.
           // Todo lo que hay desde la doble barra inclinada
           //y hasta el final de la línea es un comentario.
/* esto es un comentario de bloque –que ocupa varias líneas –Se suele usar para comentar un bloque
de código (en vez de una sola instrucción como el comentario de línea de arriba */
```

Nota: se pueden incluir comentarios de línea dentro de un comentario de bloque, pero no viceversa. Hay que tener mucho cuidado y cerrar el comentario de bloque con un */.

De lo contrario el compilador producirá un mensaje de error.

Poner entre comentarios de bloque (/*…*/) una bloque entero de código que da problemas es una buena idea de cara a identificar el elemento que da problemas sin tener que borrar (y reescribir) todo el código del bloque en cuestión.

2.6.4 #*define*: definiendo valores constantes

El *#define* es un componente útil de C que permite que el programador dé un nombre a un valor constante antes de que se compile el programa. Las constantes definidas en Arduino no usan memoria del programa en el microprocesador. El compilador substituirá referencias a estas constantes por el valor definido durante la compilación.
Sin embargo, esto puede tener algunos efectos secundarios indeseados. Por ejemplo, el valor de una constante que ha sido declarada con *#define*será incluido automáticamente en todos los nombres de constantes o variables que incluyan este nombre como parte de su propio nombre. Veamos un ejemplo de esto último:

```
#define PinLed 3
const PinLedRojo = 8;
```

El compilador sustituirá el nombre de la constante *PinLedRojo* por *3Rojo*.
La sintaxis del #define es:

```
#define Nombreconstante  valor
```

(nota que el # debe de ir pegado al *define* y es obligatorio)
Example

```
#define ledPin 3
```

- Nota importante: Recuerda que NO hay punto y coma tras la instrucción *#define*. Si lo pones el compilador te agrederá con una serie de mensajes ininteligibles mensajes de error.
- #define PinLed 4; // ¡erroneo!!. ¡No pongas el ; al final!

- Del mismo modo incluir un signo igual tras el nombre de la constante conseguirá también irritar igualmente al compilador.

```
#define PinLed =5  //¡Error!!. ¡No pongas el signo igual!
```

2.6.5 #*include*: Incluyendo bibliotecas externas.

La instrucción "*#include*" se utiliza para incluir bibliotecas externas en un sketch. Esto da el programador el acceso a un gran número de bibliotecas estándar del lenguaje C (una librería es fundamentalmente un grupo de funciones ya desarrolladas), asi como de un muchas bibliotecas escritas especialmente para Arduino. Puedes encontrar la página principal de referencia para las bibliotecas de C del AVR (el AVR es una referencia a los microprocesadores de Atmel en los cuales se basa el Arduino) en la siguiente dirección:

http://www.nongnu.org/avr-libc/user-manual/modules.html
Observa que el *#include*, del mismo similar que el *#define*, no admite punto y coma al final.
El ejemplo expuesto a continuación incluye una biblioteca utilizada para poner datos en la memoria flash en vez de usar la RAM. Esto ahorra espacio en la RAM que puede ser usado como memoria dinámica y hace que las operaciones de búsqueda en grandes tablas mucho más prácticas y rápidas.

```
#include <avr/pgmspace.h>
```

```
prog_uint16_t myConstants[] PROGMEM = {0, 21140, 702  , 9128,  0, 25764, 8456,
0,0,0,0,0,0,0,0,29810,8968,29762,29762,4500};
```

2.7 OPERADORES ARITMÉTICOS

Arduino maneja los siguientes operadores aritméticos:

= operador de asignación

Almacena el valor a la derecha signo de igualdad en la variable a la izquierda del signo de igualdad.

Es importante notar que este operador de asignación del lenguaje de programación de C tiene un significado diferente del que se le da comúnmente en álgebra donde indica una ecuación o una igualdad.

Ejemplo:

```
int sensVal;                // declara una variable int llamada  sensVal
sensVal = analogRead(0);  // almacena en sensVal (tras digitalizarlo) el voltaje  medido en el pin
analógico 0
```

- Notas importantes: La variable en el lado izquierdo del operador de asignación (=) necesita poder almacenar el valor que se le asigna. Si la variable no es lo suficientemente grande (*int* en vez de *long int*, por ejemplo), el valor almacenado en la variable será incorrecto.
- No confunda al operador de asignación [=] (un único signo de igualdad) con el operador de comparación [==] (dos signos de igualdad), que evalúa si dos expresiones son iguales.

Adición, substracción, multiplicación, y división

Estos operadores devuelven la suma, la diferencia, el producto, o el cociente (respectivamente) de los dos operandos. La operación se lleva a cabo usando el tipo de datos de los operandos, así pues, por ejemplo, 9/4 da 2 puesto que 9 y 4 son *ints*.
Esto también significa que la operación puede desbordar si el resultado es más grande que el que se pueda almacenar en el tipo de datos (por ejemplo, añadiendo 1 a un *int* con el valor 32.767 da -32.768). Si los operandos son de diversos tipos, el tipo “más grande” se utiliza para el cálculo.
Si uno de los números (operandos) es del tipo *float* o de tipo *double*, el cálculo ser realizará en coma flotante.
Ejemplos

```
y = y + 3;
x = x - 7;
i = j * 6;
r = r / 5;
```

Sintaxis

```
resultado = valor1 + valor2;
resultado = valor1 - valor2;
resultado = valor1 * valor2;
resultado = valor1 / valor2;
```

Parametros:

```
valor1: cualquier variable o constante


valor2: cualquier variable o constante
```

Notas importantes:

- Ten en cuenta que las operaciones entre constantes de tipo *int* se almacenan por defecto en una variable tipo *int*, de manera que en algunos cálculos entre constantes (por ejemplo 60* 1000) el resultado puede desbordar la capacidad de almacenamiento de una variable *int* y devolver un resultado negativo.
- Elije variables de tamaño suficientemente grande para que puedan almacenar los resultados de los cálculos por grandes que estos sean.

- Para cálculos con fracciones usa variables en coma flotante (*float*), pero sea consciente de sus desventajas: utilización de más memoria, lentitud de cálculo.
- Utilice la función de conversión *myFloat* para convertir un tipo variable a otro de manera rápida.

% (módulo):

El operador **%** (módulo) calcula el resto cuando un número entero es dividido por otro. Es útil para guardar una variable dentro de una gama particular (por ejemplo, el tamaño de un array). Su sintaxis es la siguiente:
resultado = dividendo % divisor
Ejemplos:

```
x = 7 % 5;   // x contiene 2
x = 9 % 5;   // x contiene 4
x = 5 % 5;   // x contiene 0
x = 4 % 5;   // x contiene 4
```

Ejemplo en un sketch:

```
/* modifica posición a posición los valores de un array volviendo a empezar en la primera posición
(0) cuando se halla llegado a la última (9) */

int values[10];
int i = 0;

void setup() {}
void loop()
{
  values[i] = analogRead(0);
  i = (i + 1) % 10;   // modulo operator rolls over variable
}
```

- El operador modulo no funciona con variables tipo *float*.

2.8 OPERADORES COMPUESTOS

++ (incremento) / — (decremento)

Estos operadores se utilizan respectivamente para incrementar o decrementar un variable.

Sintaxis

```
x++;  // incrementa  una unidad y retorna el antiguo valor de x
++x;  // incrementa x una unidad y retorna el nuevo valor de x
x-- ;   // decrementa x una unidad y retorna el antiguo valor de x
--x ;   // decrementa x una unidad y retorna el nuevo valor de x
```

Ejemplos:

```
x = 2;
y = ++x;      // x contiene ahora el valor 3, y contiene 3
y = x--;      // x contiene ahora el valor 2, y sigue conteniendo 3
```

Operadores compuestos += , -= , *= , /=

Estos operadores realizan una operación matemática en una variable con otra constante o variable. Estas notaciones son simplemente una simplificación práctica de una sintaxis más compleja tal y como se muestra a continuación:

Sintaxis:

```
x += y;   // equivale a la expresión x = x + y;
x -= y;   // equivale a la expresión x = x - y;
x *= y;   // equivale a la expresión x = x * y;
```

```
x /= y;   // equivale a la expresión x = x / y;
```

Parametros:

x: cualquier tipo de variable

y: cualquier tipo de variable o constante

Ejemplos:

```
x = 2;
x += 4;      // x contiene 6
x -= 3;      // x contiene 3
x *= 10;     // x contiene 30
x /= 2;      // x contiene 15
```

2.9 UTILIDADES

2.9.1 El operador *sizeof*

Este operador nos permite conocer el número de bytes de una variable o el número de bytes ocupados por un array.

Sintaxis:

```
sizeof(variable)
```

Parámetros:

Variable: cualquier tipo de variable o array (int, float, byte)

Ejemplo:

Este operador es muy práctico para trabajar con arrays (por ejemplo strings de texto) cuando queremos cambiar el tamaño del array sin alterar otras partes del programa.

El programa del ejemplo imprime carácter a carácter un string de texto. Prueba el siguiente programa cambiando la frase:

```
char myStr[] = "esto es una prueba";
int i;

void setup(){
   Serial.begin(9600);
}

void loop() {
   for (i = 0; i < sizeof(myStr) - 1; i++){
      Serial.print(i, DEC);
      Serial.print(" = ");
      Serial.write(myStr[i]);
      Serial.println();
   }
   delay(5000); // ralentiza el programa…
}
```

2.9.2 El operador PROGMEM

Este operador almacena datos en la memoria flash en vez de hacerlo en la RAM del sistema. Este operador PROGMEM debe de ser usado únicamente con los tipos de datos definidos en la librería '*pgmspace.h*'. Este operador ordena al compilador que almacene los datos especificados en la línea del operador en la memoria flash en vez de hacerlo en la RAM del sistema donde iría normalmente.
El operador PROGMEM forma parte de la librería '*pgmspace.h*', por lo tanto debes incluir esta librería al comienzo de tu sketch:

```
#include <avr/pgmspace.h>
```

Sintaxis:

- *dataType variableName[] PROGMEM = {dataInt0, dataInt1, dataInt3…};*
- *program memory dataType –cualquier tipo de variable de los incluidos abajo (program memory data types)*
- *variableName – el nombre del array que quieres almacenar en memoria flash.*
- Hay que tener en cuenta que, dado que el operador utilidad PROGMEM es un modificador variable, no hay una regla fija sobre donde colocarla dentro de la línea de especificaciones. La experiencia muestra que las dos siguientes ubicaciones de PROGMEN tienden a funcionar bien:

```
dataType variableName[] PROGMEM = {};   //esta ubicación de PROGMEN funciona.
PROGMEM  dataType  variableName[] = {}; // y esta también
```

Pero no esta otra:

```
dataType PROGMEM variableName[] = {};   // ni lo intentes…
```

PROGMEM puede ser usado con una simple variable, sin embargo, debido a la complejidad de su uso, sólo merece la pena usarla cuando quieres almacenar en memoria flash un bloque relativamente grande de datos (generalmente un array).

El uso del operador PROGMEM implica dos operaciones bien diferenciadas: almacenamiento en memoria flash y lectura usando métodos (funciones) específicos definidos en la librería '*pgmspace.h*' para devolver los datos a la memoria RAM del sistema afín de poder usarlos dentro del sketch.
Tal y como hemos mencionado antes, es esencial usar los tipos de datos especificados en *pgmspace.h*. Por alguna extraña razón, el compilador no admite los tipos de datos ordinarios. Detallamos debajo la lista completa de variables ("*program memory data types*"). Parece ser que la "*program memory*" no se lleva nada bien con los números en coma flotante, así que mejor evitarlos.

```
prog_char      - 'char' con signo (1 byte) -127 a 128
prog_uchar     - 'char' sin signo (unsigned) (1 byte) 0 a 255
prog_int16_t   - ' int' con signo (2 bytes) -32,767 a 32,768
prog_uint16_t  - 'int' sin signo (2 bytes) 0 a 65,535
prog_int32_t   - 'long' con signo (4 bytes) -2,147,483,648 a * 2,147,483,647.
prog_uint32_t  - long´' sin signo (4 bytes) 0 a 4,294,967,295
```

Ejemplo

Este ejemplo muestra como leer y escribir variables tipo unsigned char´ (1 byte) e '*int*' en PROGMEM.

```
#include <avr/pgmspace.h>  // abrir librería 'pgmspace.h'
// almacenar en PROGMEM algunos 'unsigned ints
PROGMEM  prog_uint16_t charSet[]  = { 65000, 32796, 16843, 10, 11234};

// save some chars
prog_uchar signMessage[] PROGMEM  = {"EL CHORIZO NINJA. EL NUEVO ÉXITO DEL CINE ESPAÑOL"};
unsigned int displayInt;
int k;    // contador
char myChar;

// leer un 'int' de dos bytes de la PROGMEM y almacenarlo en "displayint"
 displayInt = pgm_read_word_near(charSet + k)

// leer un 'chart' de un byte de la PROGMEM y almacenarlo en "myChart"
myChar =  pgm_read_byte_near(signMessage + k);
```

TOMANDO DECISIONES: LAS ESTRUCTURAS DE CONTROL

Programar consiste básicamente en decirle a nuestro ordenador (el Arduino) que es lo que queremos que haga cuando se confronte con una o varias opciones. En otras palabras, queremos que el Arduino pueda tomar decisiones en base a los datos disponibles en ese momento (el valor de las variables).

Estas decisiones se toman en base a el resultado de uno o varios tests lógicos ("*boolean tests*" en inglés). El resultado de estos tests lógicos será *true* o *false* (verdadero/falso) y determinará la decisión elegida.
Ejemplos de tests lógicos:

```
60 < 120; true

30 > 28; true
```

```
45<= 32; false


20 == 21; false
```

La siguiente tabla muestra los operadores lógicos y su descripción:

OPERADOR	DESCRIPCIÓN	OPERADOR	DESCRIPCIÓN
>	Mayor que	<	Menor que
>=	Mayor o igual que	<=	Menor o igual que
==	Igual que	!=	Diferente

Es importante notar que == no es lo mismo que =. == es el operador lógico de igualdad y devuelve el valor *true* o *false* mientras que = se usa para asignar valores a variables o arrays.
Los tests lógicos tienen lugar dentro de las llamadas "estructuras de control" ("control statements" en inglés). Estas estructuras son las que deciden el camino seguido por el sketch en cada momento. Por ejemplo pueden verificar el nivel de tensión (HIGH/LOW) en la entrada de cierto pin y en base a el valor leído activar (o no) un led conectado a otro pin.

A continuación vamos a explicar las estructuras de control utilizadas en el lenguaje del Arduino.

3.1 If, else, else if

La primera estructura de control que vamos a considerar es el operador "if" ("si" en español). Este operador verifica simplemente si un test lógico es cierto o falso (es decir, si devuelve el valor *true* o *false*) y en función de esto realiza (o no) una serie de acciones.
En términos prácticos el "*if*" actúa de la siguiente manera:

```
if (test lógico) {
   // realiza una serie de acciones si el test lógico resulta ser verdadero ("true")
}
```

Por ejemplo:

```
if (varA < varB) {
   digitalWrite(PinLedRojo, HIGH);
}
```

En el caso de arriba el sketch compara el calor de las variables "*varA*" y "*varB*". Si el valor de *varA* es inferior al de *varB*, el código contenido en el bloque del *if* (entre las llaves "*{" y "}"*) será ejecutado a continuación. En caso contrario, este código queda sinejecutar y se pasa a la instrucción siguiente después del bloque *if*.
El operador "*if*" puede ser complementado con el operador "*else"* (sino…). El bloque de acciones asociadas al "*else*" son ejecutadas si el test lógico del "*if*" dió "*false"* como resultado. Esto funcionaría de la siguiente manera:

```
if (test lógico) {
   // realiza una serie de acciones si el test lógico devuelve verdadero ("true")
}
else {
    // realiza una serie de acciones si el test lógico devuelve falso ("true")
}
```

Veamos el siguiente ejemplo práctico:

```
if (varA < varB) {
    digitalWrite(PinLedRojo, HIGH);
}
else {
    digitalWrite(PinLedRojo,LOW);
}
```

En este caso, si el valor de *varA* es mayor o igual que el de *varB* el led conectado al pin (*PinLedRojo*) será apagado.

Podemos complicar esta estructura de control un poco más con la inclusión de una condición dentro del *if:* el *"else if".* El ejemplo siguiente muestra el uso de esta nueva condición:

```
if (varA < varB) {
    digitalWrite(PinLedRojo, HIGH);
}
else if (varA == varB) {
    digitalWrite(PinLedVerde, HIGH);
}
else {
    digitalWrite(PinLedRojo,LOW);
}
```

Como veis el "*else if"* nos permite afinar el control dentro del "*if*". En efecto, con el "else if" hemos introducido un test lógico dentro del "if": si el valor de las dos variables ("varA" y "varB") es idéntico podemos ejecutar un bloque alternativo de acciones. En este caso activamos el led verde.
Es importante notar que solamente un bloque de acciones es ejecutado dentro de un "*if*". De hecho podemos incluir varios "*else if*" dentro de un "*if*" para realizar un control mucho más fino dentro del sketch. Sin embargo, esto no sería una solución muy elegante (ni de fácil lectura dentro del sketch). Por eso, para este tipo de tomas de decisión donde nos enfrentamos a muchas opciones diferentes, la estructura de control ideal es el "*switch case*" que veremos a continuación.

3.2 Eligiendo entre múltiples opciones: Switch case

La estrutura "*switch case*" es la opción ideal cuando tenemos que elegir entre un número más o menos elevado de opciones. De algún modo es como el panel de botones en un ascensor dentro de unos grandes almacenes (si pulsas 1 vas a la planta deportes, 2 a la de moda, 3 a la de electrodomésticos, etc…).
Esta estructura funciona de la siguiente manera:

```
switch (variable){
   case valor1:
   // instrucciones ejecutadas cuando el valor de la variable == valor1
   break;
case valor2:
   // instrucciones ejecutadas cuando el valor de la variable == valor2
   break;
case valor3:
   // instrucciones ejecutadas cuando el valor de la variable == valor1
   break;
.....

default:
   // instrucciones ejecutadas en cualquier otro caso
   break;
}
```

El "*break*" al final de cada bloque de instrucciones dentro de cada "*case*" es opcional. De no ponerlo, tras haber ejecutado ese bloque se seguirían analizando los demás casos hasta el final. Por tanto, es conveniente incluir la instrucción "*break*" al final de cada bloque si queremos impedir que el sketch se siga ejecutando dentro del "*switch*" (esto hará nuestro sketch más eficiente).
La instrucción "*default*" es también opcional. Se utiliza cuando queremos que se realicen una serie de acciones concretas en caso de que ninguno de los casos anteriores haya sido activado.

3.3 Los operadores lógicos booleanos.

Los operadores lógicos booleanos son la sal de la programación. Sin ellos no podríamos realizar programas de una cierta complejidad. Por esto es importante que entendamos bien cómo funcionan. Arduino usa los tres operadores lógicos más conocidos: AND, OR y NOT pero los representa de una manera diferente:

- AND se representa como &&
- OR se representa como **||**
- NOT como !

Veamos cómo funcionan:
AND

```
if (varA > varB && varC > varD) {
   digitalWrite(pinLedRojo, HIGH);
}
```

Si el valor de la variable varA es mayor que el de varB **Y** el valor de la variable varC es mayor que el de varD, ejecutar las instrucciones del bloque: encender el led rojo.

OR

```
if (varA > varB || varC > varD) {
   digitalWrite(pinLedVerde, HIGH);
}
```

Si el valor de la variable varA es mayor que el de varB **O** el valor de la variable varC es mayor que el de varD, ejecutar las instrucciones del bloque: encender el led verde.

NOT

```
if (!botonPulsado) {
   digitalWrite(pinLedAzul, HIGH);
}
```

Si el botón no está pulsado (NOT botonPulsado == true) encender el led azul.

3.4 Operadores lógicos binarios

A medida que progreses en tus sketches para Arduino verás que necesitas acceder al contenido de algunas variables a nivel binario (bit a bit). En algunos casos tendrás que leer información de sensores que sólo se puede encontrar a este nivel, en otros se tratará de programar un dispositivo o una acción.

La operación lógica más sencilla a nivel de bit es el NOT. Como sin duda sabes, un NOT aplicado sobre un bit simplemente cambia su polaridad (es decir, su valor pasa de ser el contrario del que tenía antes: convierte un 0 en un 1 y viceversa).

Un ejemplillo: (recuerda que en Arduino el NOT a nivel binario se escribe ~)

```
int x = 42; // En binario esto es 101010
int y = ~x; // y == 010101
```

Por otra parte los operadores lógicos AND, OR y XOR, (&, | y ^ en el lenguaje Arduino) funcionan del siguiente modo a nivel de bit:

A	B	A AND B (A&B)	A OR B (A | B)	A XOR B(A^B)
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Estos operadores se usan a veces para *enmascarar* ciertos bits en un número. Por ejemplo para extraer ciertos bits bits de un número. Si quieres extraer los dos bits de menor peso de un número puedes hacerlo así:

```
 int x = 42; // En binario esto es 101010
int y = x & 0x03; // y == 2 == B10
```

Puedes también activar (set) o borrar (clear) uno o varios bits en un número usando el operador OR. El siguiente ejemplo activa el quinto bit de x con independencia del valor que este bit tuviera antes.

```
int x = 42; // En binario esto es 101010
int y = x | 0x10; // y == 58 == B111010
```

“<<“ y “>>“

<< y >> son los operadores de desplazamiento binario (bit shift operators) que te permiten desplazar los bits de una variable a una cierta posición antes de trabajar con ellos. El primer operador desplaza los bits hacia la izquierda y el segundo a la derecha como muestran los siguientes ejemplos.

```
int x = 42; // En binario esto es 101010
int y = x << 1; // y == 84 == B1010100
int z = x >> 2; // z == 10 == B1010
```

La segunda instrucción (int y = x << 1;) copia el valor de x en y para luego desplazar sus bits (los de y) una (1) posición hacia la izquierda rellenado con un cero a la derecha. La tercera instrucción copia el valor de x en z para luego desplazar sus bits (los de z) dos (2) posiciones a la derecha rellenando con ceros a la izquierda.

Ha de notarse que la variable x original permanece inalterada. Estos shifts (desplazamientos) tienen lugar sobre las variables y, z.

Hay que evitar las operaciones de desplazamiento con números con signo ya que pueden producir resultados impredecibles. Por otra parte, no hay que confundir los operadores.

Mucho cuidado con confundir los operadores lógicos booleanos (&&, ||) con los binarios (&, |). Los operadores booleanos no trabajan a nivel binario (sólo a nivel ‘true’ o ‘false’).

4. ¡A TRABAJAR! LAS ESTRUCTURAS EN BUCLE (*LOOPS*)

Te estarás preguntando cuando le vamos a poner a trabajar al Arduino en serio. Ya hemos llegado. Las estructuras en bucle (“loops” en inglés) realizan una tarea de manera repetitiva hasta que ésta se ha completado. Los puedes visualizar como una lavadora que da vueltas y vueltas a la ropa realizando una serie de acciones hasta que está limpia.

Arduino maneja tres estructuras en bucle diferentes: “*for*”, “*while*” y “*do while*”.

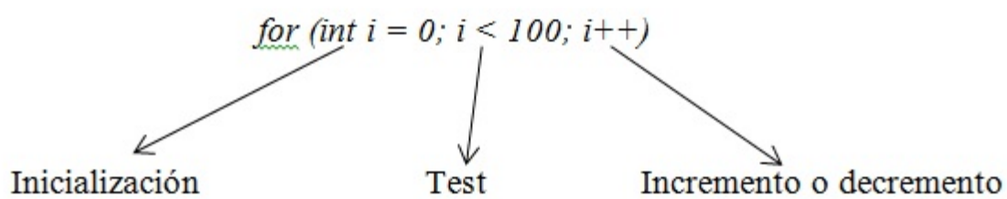
4.1 El bucle “*for*”

Este tipo de bucles se utiliza para ejecutar un bloque de código un cierto número de veces. En general se usan con un contador incremental que va aumentando hasta alcanzar un valor prefijado momento en el que el bucle se da por terminado.

El ejemplo siguiente muestra en típico bucle “*for*” que imprime el valor del contador *i* de 0 hasta 99 hasta que se apaga el Arduino.

```
void setup()
{
   Serial.begin(9600);
}
void loop {
   for (int i = 0; i < 100; i++){
       Serial.println(i);
   }
}
```

Expliquemos cómo funciona el bucle “*for*” basados en el ejemplo anterior:



La variable *i* es inicializada con el valor 0. Al final de cada bucle la variable se incrementa en 1 (*i*++ es una manera abreviada de codificar *i = i + 1*).
El código en el interior del bucle se ejecuta una vez tras otra hasta que alcanza el valor 100. En ese punto el bucle finaliza y recomienza volviendo a poner la variable *i* a 0.

La primera línea del bucle es la instrucción “*for*”. Esta instrucción tiene siempre tres partes: inicialización, test y el incremento o decremento de la variable de control o contador.
La inicialización sólo sucede una vez al comienzo del bucle. El test se realiza cada vez que el bucle se ejecuta. Si el resultado del test es “verdadero” (*true*), el bloque de código se ejecuta y el valor del contador se incrementa (++) o decrementa (–) tal como está especificado en la tercera parte del “*for*”. El bloque de código (o rutina) continuará ejecutándose hasta que el resultado del test sea “*false*” (es decir, cuando el contador *i* haya alcanzado el valor 100).

4.2 El bucle “*while*”

Muchas veces la condición para que el bucle siga ejecutándose es mucho más compleja que la estudiada en el caso anterior. En estos casos utilizaremos el bucle “*while*”. Este tipo de bucle testea una expresión contenida entre paréntesis y ejecuta el bloque de código contenido entre llaves (“*curly braquets*” en inglés) hasta que la expresión es falsa.

```
while (expression) {
   // do statements
}
```

El código del bucle “*for*” que hemos explicado antes se puede reescribir como un bucle “*while*” de la siguiente manera:

```
void setup() {
   Serial.begin(9600);
}


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

Puedes considerar un bucle “*for*” como un caso particular del bucle “*while*”. El bucle ”*while*” se usa cuando no estamos seguros de cuantas veces se va a ejecutar el bucle. El bucle “*while*” se suele usar para testear la entrada de sensores o botones.
La siguiente rutina testea el valor de un sensor:

```
int sensorValue = 0;
while (sensorValue < 2000 {
    sensorValue = analogRead(analogPin);
}
```

El código entre llaves se ejecutará ininterrumpidamente mientras el valor de *sensorValue* sea inferior a 2000. Cuando programes bucles “*while*” asegúrate de que el código del bucle cambia el valor de la variable test (*sensorValue*). Si no te arriesgas a que el bucle se ejecute indefinidamente (*infinite loop*).

4.3 El bucle “*do while*”

Este tipo de bucle es algo menos usual que el bucle “*while*”. La diferencia con el “*while*” es que el “*do while*” testea la expresión test al final del bucle (es decir después de ejecutar el bucle. De esta manera nos aseguramos que el bucle se ejecuta al menos una vez. El bucle “*do while*” se usa típicamente para leer datos de un fichero

```
do {
    // bloque de código (rutina)
   } while (expresión);
```

Al igual que en el caso anterior, el bloque de código comprendido entre las llaves del bucle se ejecutará hasta que la expresión resulte ser falsa.

4.4 *Continue*. Evitar la ejecución de parte del bloque de código en el bucle.

A veces dentro de un bucle (*for*, *do* o *while*) queremos que sólo parte del bloque de código se ejecute antes de iniciar la siguiente iteración. La instrucción “*continue*” nos permite saltar el resto del bloque de código y volver a la expresión de control del bucle para proceder con las siguientes iteraciones.
Veamos un ejemplo:

```
for (x = 0; x < 255; x ++)
   {
      if (x > 40 && x < 120){  // saltar valores de x en la ejecución del bucle
```

```
            continue;
        }
    digitalWrite(PWMpin, x);
    delay(50);
}
```

En este ejemplo vemos cómo conseguir que se evite la ejecución de la función *digitalWrite()* cuando *x* esté comprendido entre 40 y 120.

4.5 *Goto*. Escapando de un bucle

A veces sucede que se nos "enreda" un poco el código con bucles anidados en otros bucles que están dentro de otros bucles…. y no sabemos cómo salir de este laberinto de bucles. En este tipo de situaciones a veces una instrucción "de escape" puede sernos de gran ayuda.

La instrucción *goto* permite transferir el control a un punto especificado mediante una etiqueta (por ejemplo fuera de la maraña de bucles en la nos hemos metido). Esta es la sintaxis del *goto*:

```
etiqueta:
…
goto etiqueta; // transfiere la ejecución del programa a la etiqueta
```

Normalmente se desaconseja el uso del *goto* en programación ya que el uso del *goto* puede llevarnos a la creación de programas con un flujo de ejecución indefinido, que puede dificultar mucho la detección y eliminación errores (debugging).
Veamos un ejemplo de utilización del *goto:*

```
for(byte r = 0; r < 255; r++){
    for(byte g = 255; g > -1; g--){
        for(byte b = 0; b < 255; b++){
            if (analogRead(0) > 250){ goto escape;}
               // more statements ...
        }
    }
}
escape:
```

En este ejemplo podemos salir "bruscamente" de ese grupo de tres bucles anidados en caso de que la lectura de un pin analógico sea mayor que 250. En ese caso, el flujo de ejecución sería transferido fuera de los tres bucles a la etiqueta "*escape*".

FUNCIONES

Hemos visto ya de pasada las dos funciones o rutinas principales de un sketch: *setup()* y *loop()*. Ambas se declaran de tipo *void*, lo cual significa que no retornan ningún valor. Arduino simplifica muchas tareas proporcionando funciones que permiten controlar entradas y salidas tanto analógicas como digitales, funciones matemáticas y trigonométricas, temporales, etc.
Tu puedes también escribir tus propias funciones cuando necesites programar tareas o cálculos repetitivos que vas a usar varias veces en tu sketch (o en futuros sketches). La primera cosa que tienes que hacer es declarar el tipo de función (*int*, si retorna un entero, *float* si es un número en coma flotante, etc…).
Te recomendamos dar a la función un nombre que tenga que ver con lo que hace para que te resulte fácil recordarla (o identificarla cuando la veas escrita en uno de tus sketches). A continuación del nombre de la función (y sin dejar espacios) van los parámetros de la función entre paréntesis. Y ya sólo queda lo más difícil: escribir el código de la función entre llaves:

```
type functionName(parameters){
    // el código de la function va aqui.
}
```

Recuerda que si la función no retorna un valor hay que definirla como de tipo "*void*" (vacio). Para devolver el control a sketch la función usa la instrucción *return* seguida por el nombre de la variable que se quiere devolver y luego un punto y coma (;).
La siguiente rutina muestra una función que convierte una temperatura expresada en grados Fahrenheit a Celsius (también conocidos como grados centígrados):

```
float calcTemp(float fahrenheit){
float celsius;
celsius = (fahrenheit - 32)/ 1.8;
```

```
return celsius;
}
```

La function se declara de tipo *float* y se le pasa un parámetro (también de tipo *float*) que representa los grados Fahrenheit medidos por un sensor. Dentro de la función se declara la variable Celsius (de tipo *float)* que va a contener la medida de la temperatura pasada en Fahrenheit tras su conversión a Celsius (imagino que la fórmula de conversión te resulta familiar). La instrucción *return* devuelve el valor en Celsius de la temperatura pasada en Farenheit. Esta función puede ser usada como parte de un sistema para registrar las temperaturas captadas por un sensor a lo largo del tiempo.
Como puedes ver, el uso de funciones es la manera ideal de simplificar tu código y abordar tareas repetitivas.