Elementi base

del linguaggio di programmazione

di Arduino

a cura dei proff.

Prof.ssa Tiziana Marsella

Prof. Romano Lombardi

dalla libera traduzione di:

arduino language reference

(per scopo puramente didattico)

Ambiente di programmazione: Arduino.exe

Il programma che viene usato per scrivere i programmi per Arduino si chiama arduino.exe ed è scaricabile dal sito http://arduino.cc/en/Main/Software. Sito ufficiale della piattaforma Arduino.

Per aprire il programma basta fare doppio click sull'icona oppure selezionare apri dal menù a tendina che si visualizza premendo il tasto destro sull'icona del programma.

Il programma si presenta con un'interfaccia grafica senza nome chiamata sketch che significa progetto come evidenziato dalla Figura1 che segue

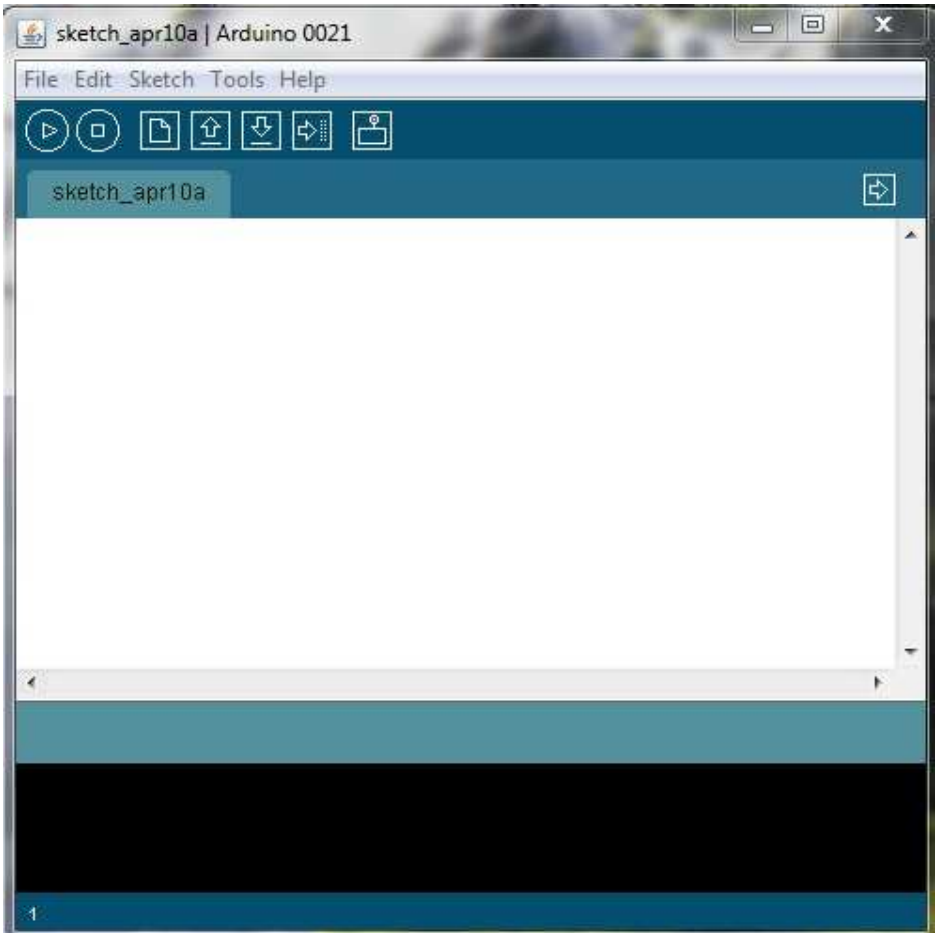

Figura 1: Esempio di sketch

Struttura di un programma

La struttura base del linguaggio di programmazione di Arduino si sviluppa sulla definizione di due funzioni: void setup() e void loop().

Queste due funzioni racchiuderanno le necessarie impostazioni per il funzionamento dei dispositivi collegati con Arduino e i blocchi di istruzioni per svolgere quanto richiesto.

void setup( )

La funzione **setup( )** è la prima ad essere chiamata quando parte uno sketch.

Viene utilizzata per inizializzare variabili, per impostare lo stato dei pin, per far partire le librerie da usare, per l'impostazione delle comunicazioni seriali.

La funzione di setup() sarà la prima ad essere eseguita dopo ogni accensione o reset di Arduino.

Sintassi

```
void setup( ){
        //  istruzioni varie;
}
```

Esempio

```
int pulsante=3;
void setup( ){
        Serial.begin(9600);
        pinMode(pulsante,OUTPUT);
}
```

in questo esempio si impostata la velocità di comunicazione seriale con il computer a 9600 bit per secondo (baud) che serve per poter visualizzare sul PC tramite il Serial Monitor (funzionante solo quando Arduino è collegato al computer) l'esito di operazioni volute e il pin 3 impostato come OUTPUT.

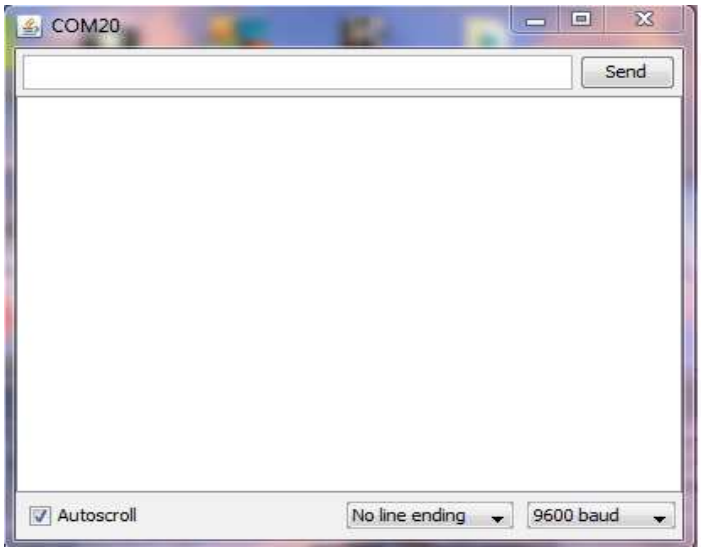

Figura 2: Serial Monitor

void loop( )

Dopo la creazione della funzione setup(), che inizializza e imposta i valori iniziali, la funzione loop() fa proprio quanto suggerisce il proprio nome eseguendo ciclicamente il programma definito al suo interno.

Permette l'esecuzione del programma, interagisce con la scheda Arduino.

Sintassi

```
void loop( ){
        // istruzioni da ripetere in modo ricorsivo;
}
```

Esempio

```
void loop( ){
        digitalWrite(3, HIGH);    // metti il pin 3 allo stato alto
        delay(1000);              // mantieni questo stato per 1 secondo
        digitalWrite(3, LOW);     // metti il pin 3 allo stato basso
        delay(1000);              // mantieni questo stato per un secondo
}
```

Sintassi

; *punto e virgola*

Il punto e virgola (**;**) deve essere inserito alla fine di un'istruzione e per separare gli elementi del programma.

Il ; è anche utilizzato per separare gli elementi di un ciclo for.

Dimenticare ; al termine di un'istruzione darà errore di compilazione. L'errore nella digitazione del testo è normale, trovare un errore è complicato. Quando si rileva un errore nella compilazione prima tra tutte le cose conviene verificare se c'è stata l'omissione del ;

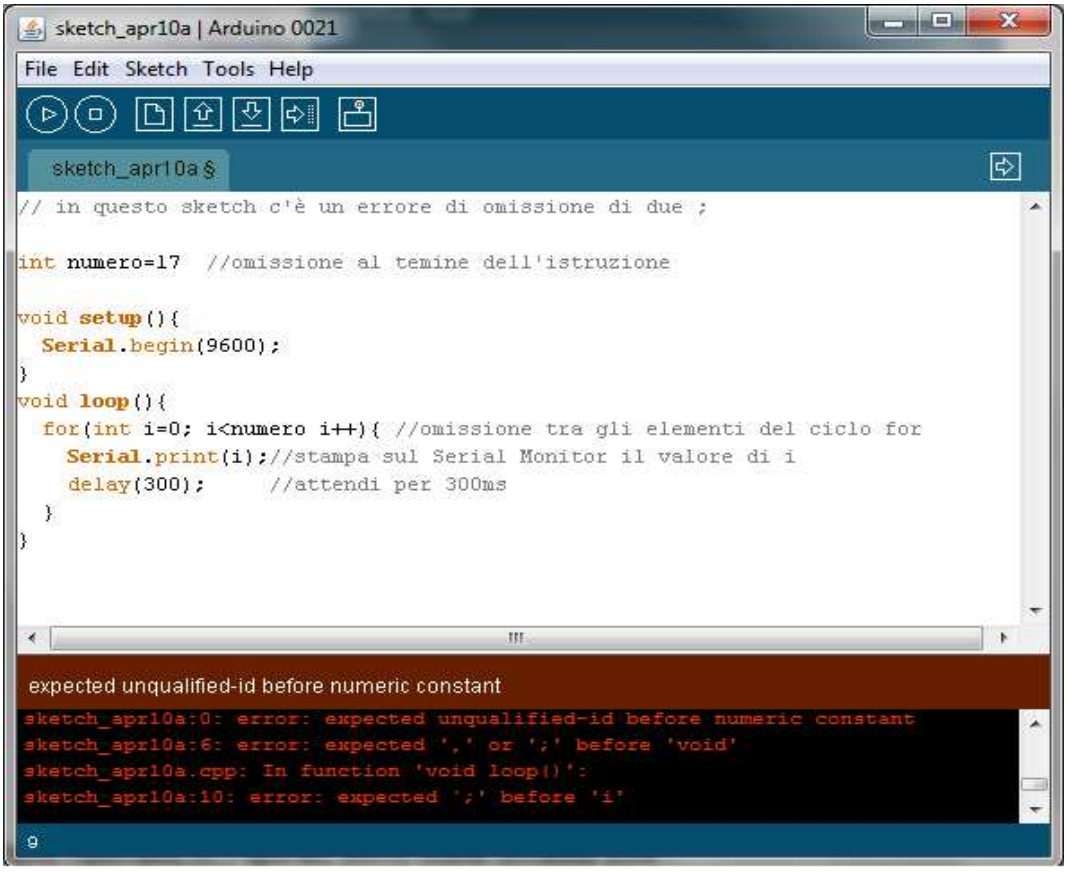

Figura 3: Errore di compilazione per mancanza di ;

Nell'esempio riportato in Figura2 sono state omessi due ;

Nella pagina dello sketch, in basso e con caratteri di color rosso su sfondo nero, vengono evidenziati entrambe le omissioni.

Esempio

int valore=13; // *; al termine di un'istruzione*

for(int i=0; i<14; i++){ … } // *; come separatore degli elementi di un ciclo for*

Le parentesi graffe { }

Un blocco di istruzioni è un gruppo di istruzioni racchiuso entro parentesi graffe { }.

La differenza tra blocco di istruzioni e gruppo di istruzioni è che il blocco crea un ambito locale per le istruzioni che contiene per cui in un blocco si posso dichiarare e impiegare variabili locali che cessano di esistere al termine dell'esecuzione del blocco stesso.

Ogni istruzione interna al blocco stesso viene eseguita in sequenza.

Le parentesi graffe { } quindi definiscono l'inizio e la fine di un blocco di istruzioni e funzioni.

Ad una parentesi che si apre deve corrisponderne la chiusura. Lo sbilanciamento delle parentesi è spesso motivo di criticità del codice di un programma grande e quindi errore di compilazione del codice scritto.

L'ambiente di sviluppo della programmazione di Arduino include caratteristiche specifiche di ricerca di bilanciamento delle parentesi. Se si seleziona una parentesi o si clicca su un punto immediatamente successivo alla parentesi, l'ambiente di sviluppo individua la parentesi abbinata.

Quando il cursore si ferma subito una parentesi viene evidenziata la parentesi corrispondente.

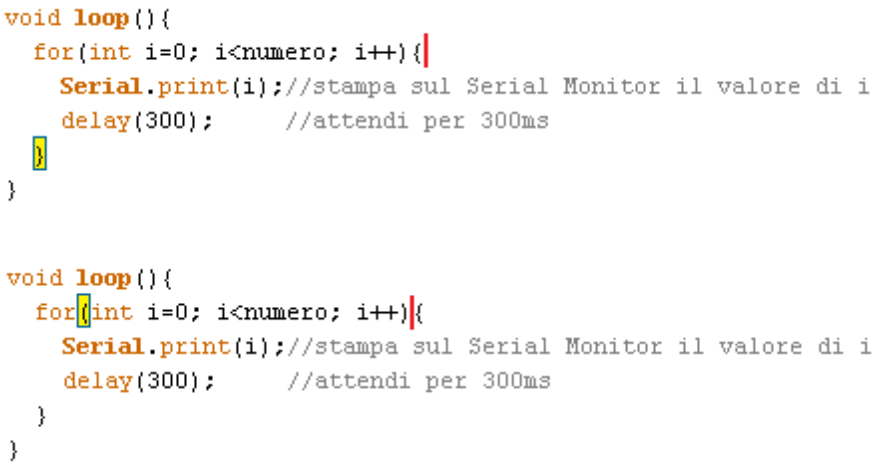

Figura 4: Corrispondenza cursore parentesi

Come si vede dalla Figura in corrispondenza del cursore, evidenziato in rosso, corrisponde la parentesi racchiusa nel rettangoletto evidenziato in giallo

Commenti

I commenti rappresentano del testo che viene ignorato dal programma e vengono utilizzati per descrivere il codice o per aiutare a comprendere parti di programma.

Blocco di commenti / .... */*

Un blocco di commenti inizia con **/*** e termina con i simboli ***/**

Sintassi

/

Tra questi simboli posso commentare il codice che sviluppo senza che in fase di compilazione venga occupata memoria. Questi commenti vengono ignorati.

**/*

Linea di commento //

Quando si vuole commentare solo una linea si utilizzano le doppie **//**

// *linea di commento*

Le linee singole di commento vengono spesso usate dopo un'istruzione per fornire più informazioni relativamente a quanto serva l'istruzione per poterlo ricordare anche in seguito.

Funzioni

Una funzione è un blocco di codice che viene eseguito nel momento in cui essa viene chiamata.

Dichiarazione di funzione

La funzione è dichiarata tramite la specificazione del tipo di funzione.

Questo è il valore che restituisce (return) la funzione stessa. Se non deve essere restituito alcun valore la funzione sarà del tipo “void”.

Dopo la dichiarazione del tipo si dichiara il nome e si fa seguire le parentesi tonde entro cui vanno inseriti i parametri che devono essere passati alla funzione se necessari.

Sintassi

```
tipo nomeFunzione(parametro1, parametro2, … , parametro n){
        istruzioni da eseguire;
}
```

Esempio di una funzione che effettua la somma tra due numeri interi num1 e num2

```
int x=somma(4,6);   // esempio di chiamata di una funzione denominata somma
int somma(int num1, int num2){
        int numeroSomma=num1+num2;
        return numeroSomma;
}
```

dove:

int specifica il tipo di funzione;

somma è il nome della funzione

num1 e **num2** sono i parametri che vengono passati alla funzione stessa

numeroSomma è il valore di tipo int che la funzione restituisce in caso di chiamata.

Se la funzione non deve restituire alcun valore sarà dichiarata di tipo ***void*** come accade per le funzioni setup() e loop().

La presenza dei parametri non è obbligatoria: la chiamata ad una funzione si può fare anche senza parametri.

Le costanti

Le costanti sono variabili predefinite nel linguaggio per Arduino e vengono utilizzate per rendere il programma più semplice da leggere.

Le costanti sono classificate in diversi gruppi.

Costanti booleane

Le costanti booleane definiscono i livelli logici di vero (**true**) e falso (**false**).

false è più semplice da utilizzare in quanto rappresenta lo zero (0) ;

true invece rappresenta una condizione di non-zero (non solo significato di “uno”) per cui anche -1, 2,-200 (valori diversi da 0) sono definite tutte condizioni di vero in senso booleano.

Importante: true e false sono scritte in minuscolo.

Costanti INPUT e OUTPUT

Queste costanti definiscono il comportamento da associare ai pin quali: INPUT e OUTPUT

Un pin digitale può essere configurato sia come INPUT che come OUTPUT. Scegliere la modalità INPUT o OUTPUT significa scegliere il comportamento del pin.

INPUT

Arduino configura un pin come INPUT con pinMode(), mettendo quel pin in una configurazione di alta impedenza.

In questa condizione il pin si presta meglio a leggere un sensore, un valore di tensione, che ad alimentare un circuito (per es con led).

OUTPUT

Un pin è impostato come OUTPUT quando Arduino lo pone in condizione di bassa impedenza tramite il pinMode().Questo significa che questo pin può fornire una certa quantità di corrente.

L'ATMEGA può fornire o assorbire da un circuito esterno una corrente fino a 40mA.

I pin configurati come output possono essere danneggiati o distrutti se cortocircuitati verso massa o verso la linea di alimentazione a 5V.

La quantità di corrente fornita dai pin dell'ATMEGA non è abbastanza potente per poter alimentare relè e motori per cui vengono richiesti circuiti di interfaccia.

Costanti HIGH e LOW

Costanti che definiscono il livello di un pin: **HIGH e LOW**

HIGH

Ha diversi significati a seconda se il pin è settato come INPUT o OUTPUT

Quando un pin è settato in modalità **INPUT** tramite pinMode e legge con il comando digitalRead un valore superiore a 3V il microcontrollore restituirà un valore HIGH.

Un pin può essere settato come INPUT con il pinMode, in seguito reso alto HIGH con digitalWrite, questo imposterà la resistenza interna di pullup a 20K che porterà il pin allo stato HIGH a meno che esso non venga costretto a LOW da qualche circuito esterno.

Quando un pin è impostato come OUTPUT con il comando pinMode e viene settato a valore HIGH con digitalWrite, il pin si troverà al valore di 5V. Esso diventa una sorgente di corrente sia verso un carico tipo LED con resistori verso massa che verso un pin, impostato come OUTPUT e settato basso.

LOW

Quando un pin è impostato come INPUT tramite pinMode() il microcontrollore legge valore LOW, tramite readAnalog, se la tensione è inferiore ai 2V.

Quando un pin invece è impostato in modalità OUTPUT con pinMode() e impostato a valore LOW con digitalWrite, il pin è posto a 0V. In questa modalità il pin assorbe corrente ad es. da un circuito esterno o da un pin impostato come OUTPUT ma HIGH.

Costanti a numero interi (Integer)

Le costanti intere sono numeri usati direttamente in uno sketch come ad es. 123. Per default questi numeri son trattati come di tipo int ma possono essere cambiati con i modificatori U e L (vedi seguito).Normalmente le costanti di tipo integer sono considerate di base 10, ma speciali formattatori possono modificarne la notazione.

Base	**esempio**	**formattatore**	**commento**
10 (decimale)	123		nessuno
2 (binario)da 0 a 255	B11001101	prefisso B	lavora solo con 8 caratteri 0 e 1
8 (ottale)	O173	prefisso O	caratteri da 0 a 7
16(esadecimale)	0x7B	prefisso 0x	caratteri 0-9, A-F,a-f

Il formattatore binario lavora solo sui caratteri 0 e 1 e con un massimo di 8. Se invece è necessario avere un input a 16bit in binario si può seguire la procedura in due step:

myInt(**B11001100***256)+B10101010 dove **B11001100** è la parte alta del numero
la moltiplicazione per 256 significa traslare tutto di 8 posti.

Formattatori U e L

Per default una costante intera è trattata come un numero di tipo int con l'attenta limitazione nei valori. Per specificare una costante di tipo Integer con un altro tipo di dato, si prosegue in questo modo:

33U o **33u** forza la costante in un formato di tipo int senza segno (unsigned)

100000l o **100000L** forza la costante nel formato di tipo long

32767UL forza il numero nel formato di tipo long senza segno

Costante di tipo floating point (notazione numeri a virgola mobile)

Anche questi costanti vengono determinate nella fase di compile time e non di run.

Le costanti di tipo floating point possono essere espresse in diverse modalità scientifiche

10.0	10	
2.34E5	2.34*10^5	234000
67e-12	67.0*10^-12	.000000000067

Le variabili

Dichiarazione di una variabile

Le variabili permettono di assegnare un nome e memorizzare un valore numerico da utilizzare per scopi successivi nel corso del programma.

Le variabili sono numeri che possono cambiare a differenza delle costanti che invece non possono mai cambiare.

Conviene scegliere il nome della variabile in modo descrittivo in modo da rendere il codice più semplice da ricostruire.

Tutte le variabili devono essere dichiarate prima del loro utilizzo.

Dichiarare una variabile significa definire il tipo di valore, come ad es. int, float, long, assegnarle un nome e assegnarle opzionalmente un valore iniziale.

Il valore della variabile può essere cambiato in ogni momento si necessita mediante operazioni aritmetiche o altri diversi tipi di assegnamenti.

Esempio: il seguente codice dichiara una variabile inputVar e le assegna il valore analogico letto dal pin 2.

```
int inputVar=0;              //inizializza la variabile di nome inputVar
inputVar=analogRead(2);      // assegna alla variabile su inizializzata il valore
                             // letto dal pin analogico 2 variandone il contenuto
```

inputVar è sempre la stessa variabile.

La variabile si può assegnare, riassegnare, si può testarne il valore e utilizzare questo valore direttamente.

Esempio

```
if(inputVar<199)        // testa se il valore è minore di 100
{
      inputVar=20;      // se la condizione è vera assegna un verto valore alla
                        // variabile
}
delay(inputVar);        // usa la variabile direttamente come argomento del
                        // comando delay
```

Una variabile può essere dichiarata e inizializzata prima del setup(), localmente all'interno di una funzione, e definita anche in un blocco di istruzioni come in un ciclo in questo caso la variabile si chiama “variabile locale”.

Variabile globale

Una variabile **globale** è una variabile che può essere vista e usata da ogni istruzione e funzione del programma; nell'ambiente di sviluppo di Arduino, ogni variabile dichiarata al di fuori di una funzione (come ad es. setup(),loop(),..) è una variabile globale.

Variabile locale

Le variabili locali sono visibili solo dalle funzioni entro cui sono dichiarate o all'interno di funzioni o cicli dove vengono definite.

È possibile avere due o più variabili locali che hanno lo stesso nome ma in differenti parti del programma, che contengono valori differenti. Bisogna essere sicuri che solo una funzione abbia accesso alla variabile in modo da ridurre la probabilità di errore della scrittura del codice.

Un programma può diventare molto grande e complesso e le variabili locali sono di solito usate per assicurare che solo una funzione ha accesso alle proprie variabili. Questo previene errori di programmazione in quanto una funzione potrebbe inavvertitamente modificare una variabile utilizzata da un altro programma.

A volte è anche comodo dichiarare e inizializzare una variabile all'interno di un ciclo. Questo crea una variabile a cui si può accedere solo dall'interno delle parentesi che delimitano il ciclo.

Il seguente esempio evidenzia come dichiarare differenti tipi di variabili e evidenzia la diversa visibilità delle stesse.

```
int valore;         // variabile visibile ad ogni funzione

void loop( ){
 int k=0;
  for(int i=0;i<100;i++){
      k++;          // questa variabile è esterna al ciclo for ma è visibile solo nel
                    // ciclo loop. Per cui ha un diverso  livello di località rispetto
                    // alla variabile successiva f
     float f=analogRead(2);      // f è una variabile locale che non è disponibile
                                 // all'esterno del ciclo
}
```

Static

La parola chiave ***Static*** viene utilizzata per creare una variabile che è visibile solo da una funzione.

Diversamente dalla variabile locale che viene creata e distrutta ogni volta che una funzione la chiama, la variabile *Static* persiste all'interno della funzione che la chiama, mantenendo il proprio dato all'interno della funzione chiamante.

Le variabili dichiarate come statiche saranno definite ed inizializzate solo la prima volta che una funzione è chiamata.

Esempio

```
/*    Creazione di numeri casuali spostandosi in sopra e in sotto tra due limiti
      Il massimo scostamento  in un loop è regolato dal parametro step
      La variabile statica varia in più o in meno di una quantità random
      questa tecnica è chiamata "rumore rosa".
      -20 e 20 sono i limiti inferiore e superiore che delimitano il range di variabilità dei
      numeri casuali     */
#define limiteInferiore     -20
#define limiteSuperiore     20
int step;
int valoreRandom;
int totale;

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

void loop( ){                    // testa la funzione   generaRandom ogni 100ms
        step= 5;
        valoreRandom = generaRandom(step);
        Serial.println(valoreRandom);
        delay(100);
}

int generaRandom(int salto){
        static int casuale;       // variabile che memorizza un valore in
                                  // generaRandom( )   - dichiarata  static cosicché
                                  // memorizza il valore nella funzione chiamate
                                  // ma nessun'altra funzione può cambiarne il valore
        casuale = casuale + (random(-salto, salto + 1));

// l'if seguente verifica  che non superi il limite superiore e il limite inferiore
        if ( casuale < limiteInferiore){
                casuale = casuale + ( limiteInferiore - casuale);
               // riflette il numero in senso positivo
        }

        else if( casuale > limiteSuperiore){
                casuale = casuale - ( casuale - limiteSuperiore);
                 // riflette il numero in senso negativo
```

```
}
return casuale;
}
```

volatile

volatile è un termine conosciuto come qualificatore di variabile.

É di solito utilizzato prima della tipologia della variabile, per modificare il modo in cui il compilatore e una sequenza del programma tratta la variabile.

La dichiarazione di una variabile come volatile è una direttiva per il compilatore (software che trasla il codice C/C++ in codice macchina riconoscibile dal chip Atmega di Arduino).

Specificatamente, essa indica al compilatore di caricare la variabile dalla memoria RAM (memoria temporanea dove le variabili del programma sono memorizzate e manipolate), e non dalla memoria register.

In alcune condizioni il valore per una variabile memorizzata nel register potrebbe essere inesatto.

Una variabile può essere dichiarata volatile quando il suo valore può essere cambiato da qualcosa che accade al di fuori del controllo della sezione del codice nel quale appare, come ad esempio l'esecuzione di un thread.

In Arduino, l'unico posto dove questo è possibile che avvenga è nella sezione del codice associato con gli interrupt, chiamata routine di servizio degli interrupt.

Esempio

```
// accendi/spegni LED quando lo stato del pin si modifica
int pin = 13;                    // scelta del pin dove mettere il diodo
volatile int stato = LOW;     //stato è una variabile booleana

void setup( ) {
        pinMode(pin, OUTPUT);
        attachInterrupt(0, blink, CHANGE);  //vedi gestione Interrupt
}

void loop( ){
        digitalWrite(pin, stato);  //stato può essere o  LOW oppure HIGH
}

void blink( ){
        stato = !stato;   //cambia stato alla variabile stato
}
```

const

Il termine **const** sta per costante.

É un qualificatore di variabile che modifica il comportamento della variabile, rendendola una variabile di “solo lettura”.

Questo significa che la variabile può essere usata come ogni altro tipo di variabile ma che il suo valore non può essere cambiato. Verrà segnalato errore di compilazione nel momento in cui si cercherà di assegnare un valore a una variabile costante.

La costante viene definita con il termine const secondo le regole di scopo delle variabili che governano le altre variabili. Questo rende il termine const un metodo superiore per la definizione delle costanti ed è preferito rispetto al #define.

Esempio

```
const float pi = 3.14;
float x;
// ....
x = pi * 2;        // scopo di usare le costanti in matematica : la variabile x utilizza
                   // la costante precedentemente definita.
pi = 7;            // illegale - non è possibile modificare, sovrascrivere una costante
```

#define or const

Si può utilizzare **const** o **#define** per creare una costante numerica o di tipo String.

Per gli array occorre che venga utilizzato il termine const.

In genere const è preferito rispetto a #define per la definizione delle costanti.

Tipo di dati

void

Il termine void è usato solamente nella dichiarazione di funzioni.

Esso indica che la funzione non restituisce alcuna informazione dopo essere stata chiamata.

Esempio

I metodi setup( ) e void( ) non restituiscono alcuna informazione non c'è un'istruzione di return come in altre funzioni

```
void setup( ){
  // …
}

void loop( ) {
        // ...
}
```

boolean

Ogni variabile booleana occupa un byte di memoria (8bit) e può assumere solo due valori: livello logico **true** o livello logico **false**.

false è più semplice da utilizzare in quanto rappresenta lo zero (0) ;

true invece rappresenta una condizione di non-zero (non solo significato di “uno”) per cui anche -1, 2,-200 (valori diversi da 0) sono definite tutte condizioni di vero in senso booleano.

Importante: true e false sono scritte in minuscolo.

Esempio

```
int LEDpin = 5;                 // LED su pin 5
int switchPin = 13;             // switch su pin 13, altro estremo collegato a massa.
boolean running = false;

void setup( ) {
        pinMode(LEDpin, OUTPUT);
        pinMode(switchPin, INPUT);
        digitalWrite(switchPin, HIGH);        // attiva il resistore di  pull-up
}
```

```
void loop( ){
        if (digitalRead(switchPin) == LOW) {    // pressione del tasto – il resistore di
                                                // pull-up  mantiene il pin a valore alto
        delay(100);                             // delay per debounce switch
        running = !running;                 // alterna il valore della variabile running
        digitalWrite(LEDpin, running);          //segnala con LED
        }
}
```

char

Questo tipo di dato occupa fino a 1 byte di memoria e memorizza un carattere.

Per intenderci i caratteri letterali occupano una sola posizione (es. “a”, “d”,..).

I char sono memorizzati come numero corrispondenti al codice ASCII.

Il dato di tipo **char** è un tipo con segno per cui codifica numeri tra **-128** fino a **127** (da 2^7 a 2^7-1 in quanto il bit più significativo conserva l'informazione del segno).

L'insieme di caratteri che occupano posizioni multiple (es. “abc”) , viene chiamata **stringa**.

Esempio

```
char mioChar = 'A';
char mioChar = 65;          // entrambi i modi sono equivalenti
```

unsigned char

Per un tipo senza segno si utilizza “**unsigned char**” che codifica invece numeri da 0 a 255 (da 0 a 2^8). Questo per poter svolgere calcoli aritmetici con questi simboli in caso di necessità di risparmiare spazio di memoria.

Esempio

```
unsigned char mioChar = 240;
```

byte

questo tipo può sostituire il tipo char unsigned in quanto memorizza un numero senza segno a 8 bit, cioè da 0 a 255 (da 0 a 2^8) .

Esempio

```
byte b = B10010;      // "B" è il formattatore binario (B10010 = 18 decimale)
```

int

Integer è un modo per memorizzare numeri utilizzando 2 byte.

Il range di memorizzazione va quindi da -32768 a 32767 cioè da -2^{15} a (2^{15} -1).

La tecnica utilizzata nella memorizzazione dei numeri negativi è quella del complemento a due: il bit più significativo individua il segno del numero mentre il resto dei bit sono invertiti e sommati ad 1 (tecnica per risalire dal complemento a 2 di un numero al numero stesso).

Se il bit più significativo è 1 il numero sarà negativo e i rimanenti 15 numeri rappresentano il complemento a due del numero; se il bit più significativo è 0 il numero sarà positivo e i rimanenti 15 bit rappresenteranno il complemento a due del numero.

Sintassi

```
int var = val ;
```

var è il nome scelto per la variabile

val è il valore che si vuole assegnare

Esempio

```
int caso = - 3065;          // dove caso è il nome che si vuole utilizzare nel corso
                            // del programma, -3065 è il numero assegnato.
```

Quando le variabili eccedono la massima capacità avviene un meccanismo che prende nome di "roll over" legato all'overflow della memoria.

Esempio di effetto roll-over

```
int x;
x = - 32768;
x = x - 1;  // x contiene ora 32767 in quanto inizia da capo (roll over negative)

x = 32767;
x = x+1;  // x contiene -32768  ( roll over positivo )
```

unsigned int

Tipo che memorizza un intero senza segno memorizzato sempre in due byte.

I valori coprono un range che va da 0 a $2^{16} - 1 = 65535$.

Sintassi

```
unsigned int var = val ;
```

var è il nome scelto per la variabile

val è il valore numerico che si assegna alla variabile.

Esempio

```
unsigned int y = 2456 ;        // y è il nome della variabile, 2456 è il valore che gli
                                  // viene assegnato.
```

Anche in questo caso si verifica la condizione di roll over.

Esempio di roll-rover

```
int x ;
x = 0 ;
x = x - 1;              // x=65535 invece di -1
```

```
x=65535;
x=x+1;              // x=0 in quanto successivo al massimo
```

word

Un tipo word memorizza un numero senza segno a 16 bit, da 0 a 65535 (da 0 a 2^{16} -1) allo stesso modo di un unsigned int.

Sintassi

```
word var = val ;
```

var è il nome scelto per la variabile

val è il valore numerico che si assegna alla variabile.

Esempio

```
word  lunghezza=44663;
// lunghezza è il nome della variabile mentre 44663 è il valore assegnato
```

long

I tipo long permette di estendere la possibilità per memorizzare un numero.

Utilizza 32 bit (4 byte) per cui copre valori che vanno da -2147483648 a 2147483647 (-2^{31} a 2^{31} -1).

Sintassi

```
long var=val;
```

var è il nome scelto per la variabile

val è il valore numerico che si assegna alla variabile.

Esempio

```
long altezza=446657685;
// altezza è il nome della variabile, 446657685 è il numero
```

unsigned long

È un modo per estendere la possibilità di memorizzare numeri senza segno utilizzando 4 byte. Per questo il range copre numeri che vanno da 0 a 2^{32} - 1 cioè da 0 a 4,294,967,295.

Sintassi

```
unsigned long var=val;
```

var è il nome scelto per la variabile

val è il valore numerico che si assegna alla variabile.

Esempio

unsigned long profondo = 888777555;
//numero che viene memorizzato nella variabile profondo

floating point: numero in virgola mobile

I numeri reali vengono rappresentati secondo la modalità virgola mobile utilizzando un numero fisso di cifre e scalando utilizzando un esponenziale.

Un dato di questo tipo comprende numeri decimali.

I numeri a virgola mobile sono spesso utilizzati per approssimare valori analogici e continui in quanto essi hanno una maggiore risoluzione degli interi.

Questi numeri vanno da – 3.4028235E^+38 fino a 3.4028235E^+38.

I numeri a virgola mobile hanno 6-7 cifre significative: ciò si riferisce al numero di cifre totali non al numero di cifre a destra della virgola. Se si necessita di maggiore precisione si può utilizzare la modalità double che in Arduino ha la stessa estensione di un float.

I numeri in notazione virgola mobile non sono molto precisi. Può avvenire ad esempio che 6.0/3.0 non sia uguale a 2.0, allo stesso modo se si vuole confrontare due numeri uguali occorre imporre che la differenza tra i due sia inferiore ad un certo valore piccolo.

Utilizzare i numeri in virgola mobile rallenta l'esecuzione dei calcoli rispetto all'utilizzo dei numeri definiti interi.

Sintassi

float var=val;

var è il nome scelto per la variabile

val è il valore numerico che si assegna alla variabile.

Esempio

```
int x,y ;
float z ;
x = 1;
y = x/2;        //y contiene 0 che è la parte intera di 0.5
z=float(x)/2.0;  //z contiene .5 (avendo utilizzato 2.0 e non 2)
```

double

Numero a virgola mobile e doppia precisione. Occupa lo spazio di 4 bytes.

L'implementazione del double in Arduino è la stessa del float, con nessun guadagno in precisione.

char array - String

Le stringhe di caratteri possono essere rappresentate in due modi:

1. usando il tipo String, che fa parte del codice di Arduino a partire dalla versione 0019,
2. costruendo una stringa con un array di caratteri e **null** per terminarla.

L'oggetto String permette più funzionalità a scapito però di maggiore occupazione di memoria.

Array di char

Un array è un insieme di elementi tutti dello stesso tipo. Nel caso specifico, in un oggetto di tipo String, gli elementi dello stesso tipo sono char.

Sintassi per dichiarare stringhe di caratteri

```
char Str1[15];          // definisce un array di char lungo 15 caratteri senza nessuna
                        // assegnazione
char Str2[8]={'a','r','d','u','i','n',o'};    // i caratteri sono 7, la lunghezza dell'array
                                              // definito è 8, durante la compilazione viene
                                              // accodato un carattere nullo per arrivare
                                              // alla lunghezza di 8
char Str3[8]={'a','r','d','u','i','n',o','/0'};    // esplicitamente viene aggiunto il
                                                   // carattere null (/0).
char Str4[ ]= “arduino”;          // in questo caso la sezione dell' array sarà definita
                                  // dopo che il compilatore avrà fatto il fit nell'array
                                  // cioè fino al valore null della stringa
char Str5[8]= “arduino”;          // inizializza l'array con esplicita sezione e stringa
                                  // costante
char Str6[15]= “arduino”;         // inizializza l'array con lunghezza 15 lasciando
                                  // uno spazio extra dopo la stringa di
                                  // inizializzazione.
```

Generalmente una stringa termina con un carattere “null” che in codice ASCII corrisponde a 0, permettendo alla funzione di capire quando la stringa è terminata, in caso contrario continuerebbe a leggere byte di memoria che non fanno parte della stringa. Questo significa che occorre inserire uno o più caratteri null rispetto a quello che basta per contenerlo.

Se la stringa è troppo lunga si può scrivere in questo modo:

```
char Str[ ] = “Questa è una stringa”
“  lunga che posso scrivere”
“ su linee diverse”;          // lasciare lo spazio di separazione tra le parole.
```

Array di stringhe

E' spesso conveniente , quando si lavora con una grande quantità di testo, lavorare con un array di stringhe.

Dato che le stringhe sono a loro volta degli array , l'array di stringhe è un esempio di array a due dimensioni.

Nel codice seguente , l' * (asterisco) dopo il tipo char (char*) indica che questo è un array di puntatori. I puntatori sono una trovata di C che non è necessario capire qui per capire come funzioni il codice.

char* mieStringhe[ ]= { “Stringa 1”, “Stringa 2”, “Stringa 3”, “Stringa 4”, “Stringa 5”, “Stringa 6” };

Esempio

```
void setup( ) {
  Serial.begin(9600);   // impostazione velocità di collegamento con il pc
}


void loop( ) {
  for ( int i=0; i<6; i++) {
    Serial.println( mieStringhe[i] );      // stampa su monitor pc le stringhe
                                           // presenti nell'array chiamato
                                           // mieStringhe[ ]
    delay(500);
  }
}
```

String – oggetto (classe)

La classe String, disponibile dalla versione 0019 di Arduino, permette l'uso e la manipolazione di stringhe di testo in maniera più complessa di quanto permette l'array di char.

Si possono concatenare stringhe, appendere altro testo ad esse, cercare e sostituire parti di stringhe, e molto altro.

Utilizza maggior memoria rispetto all'array di char, ma è una classe che viene utilizzata maggiormente quando si tratta di manipolare testo.

Metodi della classe

String( )

charAt( )

compareTo( )

concat?( )

endsWith( )

equals( )

equalsIgnoreCase( )

getBytes( )

indexOf( )

lastIndexOf( )

length( )
replace?( )
setCharAt( )
startsWith( )
substring( )
toCharArray( )
toLowerCase( )
toUpperCase( )
trim( )

operatori
[ ] (accesso agli elementi')
+ (concatenamento?)
== (confronto?)

Array

Un array è una collezione di variabili che sono accessibili tramite un indice numerico.

Creazione e/o dichiarazione di un array

Tutti i metodi seguenti sono validi modi per creare un array.

```
int mieiNumeri[6];       //  dichiarazione di un array senza inizializzazione
int mieiPin[ ]= {2,4,8,3,6};  // dichiarazione di un array senza  definirne la dimensione.
                              // Il compilatore conta gli elementi e crea un array avente
                              // dimensione adatta.
int valoriSensori[6]={2,4,-8,3,2,5};       // dichiarazione e inizializzazione di un array
char messaggio[5]= “ciao”;     // Dichiarazione ed inizializzaizone di un array di char.
                               // Occorre un carattere in più per definire il valore null che
                               // determina la fine dell'array di caratteri.
```

Accesso agli elementi di un array

Gli indici degli array partono dallo 0, cioè il primo elemento occupa la posizione 0.

In un array avente 10 elementi, l'indice 9 rappresenta l'ultimo della lista.

Sintassi per leggere un valore in un array
tipo var = mioArray[n] ;
var indica la variabile di lettura o di accesso;
mioArray[ ] è il nome dell'array di cui si vuole accedere agli elementi
n indica la posizione occupata.

Esempio

```
int  numeri[10]={3,4,6,8,5,12,15,7,9,13};      // definizione e inizializzazione
                                               // dell'array chiamato numeri[ ]
                                               // avente 10 elementi
int x ;                  //  inizializzazione della variabile x di tipo int
x = numeri[0];           // x=3
x = numeri[9];           // x=13
x = numeri[10];          // richiesta non valida. Contiene un numero casuale (altro
                         // indirizzo di memoria)
```

Attenzione:

- accedere con un indice oltre quello richiesto dalla sezione dell'array comporta andare a leggere memoria utilizzata per altri scopi.
- scrivere oltre può spesso portare al crash o al malfunzionamento del programma.
- questo tipo di errore potrebbe risultare di difficile individuazione in quanto diversamente dal Basic o da Java, il compilatore C non verifica se l'accesso all'array è nei limiti consentiti dichiarati.

Assegnare un valore ad un array

Per asseganre un valore ad un array basta fare:

Sintassi

```
tipo mioArray[ max ] ;
mioArray[ n ]= valore;
```

tipo identifica a quale tipo appartengono gli elementi che fanno parte dell'array;

mioArray[ ] è il nome scelto per l'array ;

max identifica la dimensione dell'array;

valore è l'elemento che si vuole assegnare all'array mioArray nella posizione n.

Esempio

```
int numeri[ 10 ];       //dichiarazione array di numeri interi
numeri[ 0 ]=3;          //assegnazione del valore 3 al primo elemento dell'array
numeri[ 1 ]=2;
numeri[ 7 ]= - 8;
```

Gli array vengono manipolati spesso con loop specifici.

Nell'esempio che segue vengono stampati sul monitor del pc tutti i numeri che fanno parte dell'array numeri[ ].

```
int i;
for(i=0; i<10; i = i+1){
        Serial.println(numeri[ i ]);
}
```

Conversioni

char( )

Converte un valore in un dato di tipo char

Sintassi
char(x)
Il parametro x è un valore di qualsiasi tipo.

La funzione restituisce un dato di tipo char.

Esempio
```
int x = 126 ;   // 126 è il valore che viene memorizzato in x
char y;
y=char(x);     // y è la conversione in char corrispondente a 126
```

byte( )

Converte un valore in un dato di tipo byte

Sintassi
byte(x)
Il parametro x rappresenta un valore di qualsiasi tipo.

La funzione restituisce un dato di tipo byte

Esempio
```
int x = 126 ;   // 126 è il valore che viene memorizzato in x
byte y;
y=byte(x);     // y è la conversione in byte corrispondente a 126
```

int( )

Converte un valore in un dato di tipo int

Sintassi
int(x)
Il parametro x rappresenta un valore di qualsiasi tipo.

La funzione restituisce un dato di tipo int

Esempio
```
byte x = 200 ;   // 200 è il valore che viene memorizzato in x
int  y;
y=int(x);     // y è la conversione in int corrispondente a 200
```

word( )

Converte un valore in un dato di tipo word o crea un dato di tipo word a partire da due byte

Sintassi

word(x)

word(h,l) *// crea un tipo word (16 bit) definendo i due byte che lo compongono*

Il parametro x rappresenta un valore di qualsiasi tipo;

Il parametro h rappresenta il byte più significativo (a sinistra, alto) , il parametro l invece rappresenta il byte meno significativo (il byte più a destra, basso).

La funzione restituisce un dato di tipo word.

Esempio

```
int x = 126 ;   // 126 è il valore che viene memorizzato in x
word y;
y=word(x);     // y è la conversione in word corrispondente a 126
int h = 126 ;   // 126 è il valore che viene memorizzato in x
int l=34;
word y;
y=word(h,l);     // y è la conversione in word corrispondente a   126*2^8+34
```

long( )

Converte un valore in un dato di tipo long

Sintassi

long(x)

Il parametro x rappresenta un valore di qualsiasi tipo.

La funzione restituisce un dato di tipo long.

Esempio

```
int x = 126 ;   // 126 è il valore che viene memorizzato in x
long y;
y=long(x);     // y è la conversione in long corrispondente a 126
```

float( )

Converte un valore in un dato di tipo float

Sintassi

float(x)

Il parametro x rappresenta un valore di qualsiasi tipo.

La funzione restituisce un dato di tipo float.

Esempio

```
int x = 126 ;          // 126 è il valore che viene memorizzato in x
float y=float(x);      // y è la conversione in float corrispondente a 126
```

Utilità

sizeof( )

L'operatore sizeof restituisce il numero di byte di cui è composta una variabile di qualsiasi tipo, oppure il numero di byte occupato da un array.

Sintassi

sizeof (variabile)

Il parametro variabile è qualsiasi tipo di variabile (int, float, byte...) o un array .

L'operatore sizeof è spesso usato per trattare con gli array (come le stringhe di caratteri) dove è conveniente essere abili a cambiare la sezione dell'array senza interrompere altri parti del programma .

Esempio: questo programma stampa una stringa di testo un carattere a volta (un carattere di tipo char è rappresentato da un byte, meglio tenerlo a mente), prova a cambiare la frase del testo.

```
char miaStringa[] = "Questo e' un test";          // attenzione agli accenti.... non è detto che abbiano
                                                  // corrispondenza in ASCII
void setup( ){
        Serial.begin(9600);
}

// sizeof in questo caso restituisce il numero di byte dell'array coincidente con il numero di  caratteri che costituiscono la stringa.
void loop( ) {
        for (int i = 0; i < sizeof(miaStringa) - 1; i++){
                Serial.print(i, DEC);
                Serial.print(" = ");
                Serial.println(miaStringa[i], BYTE);
        }
}
```

sizeof restituisce il numero totale di byte.

Così per la maggior parte delle variabili come int, è necessario fare un loop che permette di fare qualcosa di simile a questo. Ad esempio:

```
for (int i = 0; i < (sizeof(myInts)/sizeof(int)) - 1; i++) {
// istruzioni con myInts[i]
}
```

Istruzioni condizionali

Istruzione if

L'istruzione if permette di eseguire blocchi di codice diversi in base al risultato di una condizione.

La struttura è composta dalla parola if, seguita da una condizione booleana e poi da un blocco di istruzioni (eventualmente anche una sola istruzione) che viene eseguito se la condizione è verificata.

Esempio di if seguito da un blocco di istruzioni

```
if(x>120){
   digitalWrite(LED1,HIGH);   // accendi LED1
   digitalWrite(LED2,LOW);    // spegni LED2
   delay(2000);               // mantieni queste condizioni per 2 secondi
   digitalWrite(LED1,LOW);    // spegni LED1
   digitalWrite(LED2,HIGH);   // accendi LED2
   delay(2000);               // mantieni queste condizioni per 2 secondi
}
```

if seguito da una sola istruzione: in questo caso si possono omettere le { }

```
if (x>120)   digitalWrite(LED,HIGH);

if (x>120)
digitalWrite(LED,HIGH);

if (x>120) { digitalWrite(LED,HIGH); }

if (x>120){
    digitalWrite(LED,HIGH);
}
```

Questi modi di scrivere sono tutti corretti.

Le condizioni che vengono valutate all'interno delle parentesi tonde () richiedono l'uso di uno o più operatori seguenti (operatori di confronto):

```
x == y      // x uguale a y
x != y      // x diverso da y
x<y         // x minore di y
```

```
x>y       // x maggiore di y
x<=y      // x minore o uguale a y
x>=y      // x maggiore o uguale a y
```

Attenzione:

scrivere x=10 significa assegnare alla x il valore 10, scrivere x==10 significa verificare se x è uguale a 10.

Se per errore si scrive if(x=10) la condizione risultante sarà sempre vera in quanto ad x viene assegnato il valore 10 che essendo un valore non nullo restituisce sempre vero.

Condizioni if/else

La parola **else** (facoltativa) introduce il blocco di istruzioni (o la singola istruzione) che deve essere eseguito nel caso in cui la condizione introdotta da if risulti falsa.

Esempio

```
if(pin5<500){
   digitalWrite(LED1,HIGH);   //accendi LED1
   digitalWrite(LED2,HIGH);   //accendi LED2
}
else
{
   digitalWrite(LED1,LOW);   //spegni LED1
   digitalWrite(LED2,LOW);   //spegni LED2
}
```

praticamente else testa un'altra condizione if.

Un altro esempio chiarisce il modo in cui è possibile effettuare test su condizioni differenti utilizzando if (condizione), else if(condizione) ed else

```
if(pin5<250){
   digitalWrite(LED1,HIGH);          // accendi LED1
   digitalWrite(LED2,HIGH);          // accendi LED2
}
else if(pin5>=700){
   digitalWrite(LED1,LOW);           // spegni LED1
   digitalWrite(LED2,LOW);           // spegni LED2
}
else                          // (if(x>=250 && x<700)   valori di x compresi tra 250 e 700
{
   digitalWrite(LED1,HIGH);          // accendi LED1
   digitalWrite(LED2,HIGH);          // accendi LED2
}
```

Istruzioni switch ... case

Spesso in un programma c'è necessità di confrontare il valore di una variabile con una serie di valori fino a trovare quello corrispondente. Disponendo solo della struttura if il codice potrebbe diventare contorto.

Nel linguaggio di programmazione per Arduino esiste un meccanismo che permette di raggruppare confronti ed azioni in una sola istruzione migliorandone la leggibilità.

Questo meccanismo è chiamato **switch/case.**

switch/case controlla il flusso del programma permettendo di specificare codice differente che dovrebbe essere eseguito a seconda delle condizioni.

In particolare, **switch** confronta il valore della variabile al valore specificato dalla clausola **case**.

Sintassi:

```
switch(var){
        case label1:
                //istruzioni
                break;
        case label2:
                //istruzioni
                break;
        default:
                //istruzioni
}
```

dove var è una variabile che va a confrontarsi con i diversi casi

e label sono i valori di confronto con var

Esempio: pin5 è la variabile di confronto

```
switch(pin5){
case 250:   //se pin5 è uguale come valore a 250 accendi tutti e due i led
       digitalWrite(LED1,HIGH);        //accendi LED1
       digitalWrite(LED2,HIGH);        //accendi LED2
       break;                          //permette di uscire dalla ricerca del case

 case 500:
   digitalWrite(LED1,LOW);             //spegni LED1
   digitalWrite(LED2,LOW);             //spegni LED2
   break;

default : break;
}
```

Quando pin5 è uguale al valore definito dalla clausola case si esegue il blocco di istruzioni che seguono il case, se nessuno dei valori coincide con il valore pin5 si esegue l'istruzione che segue i case cioè default (che è opzionale). La condizione di default è facoltativa: se manca e nessun valore coincide con la variabile l'esecuzione termina senza alcun effetto.

A differenza dell' if non è necessario utilizzare un blocco di istruzioni per far eseguire un gruppo di istruzioni in sequenza.

La parola **break** permette la fuoriuscita dal ciclo e viene tipicamente utilizzata alla fine di ogni clausola case.

Se mancasse l'istruzione break, anche dopo aver trovato la condizione giusta, verrebbero eseguiti comunque i confronti successivi fino alla prima istruzione di break oppure fino alla fine del ciclo. In alcuni casi questa procedura è corretta, in genere si includono le istruzioni break per selezionare una sola possibilità.

Praticamente il break interrompe l'esecuzione di un ciclo e l'esecuzione del programma riprende dal codice che segue la successiva parentesi chiusa graffa }.

L'omissione dell'istruzione break è utile quando la stessa istruzione deve essere eseguita per più valori distinti. In questo caso le clausole per tali valori sono poste di seguito, senza istruzioni, e switch esegue la prima istruzione che trova

Sintassi

```
switch(var){
        case label1:
        case label2:
        case label3:
                //istruzioni
        break;
        default:
                //istruzioni
}
```

dove ***var*** è una variabile che va a confrontarsi con i diversi casi e ***label*** sono i valori di confronto con var

Nell'esempio seguente verrà stampato x è un numero pari se x è 2, 4, 6, 8 altrimenti stamperà x è dispari.

```
int x
switch(x){
 case 2:
case 4:
case 6:
case 8:
   Serial.println("x è un numero pari");
   break;
   default:   Serial.println("x è dispari");
}//switch
```

*Istruzione **for / ciclo for***

Il ciclo **for** ripete un'istruzione o un blocco di istruzioni per un numero prefissato di volte.

Il ciclo for viene utilizzato per una iterare (ripetere) un blocco di istruzioni per un numero prefissato di volte e utilizza un contatore per incrementare e terminare un ciclo. Viene spesso utilizzato per indicizzare gli elementi di un array.

Sintassi

```
for (inizializzazione ; condizione ; incremento){
  istruzioni;
}
```

Esempio

```
for(int i=0; i<10;i++){
        int pin = i ;
        digitalWrite(pin,HIGH);      // mette i pin che vanno da 0 a 9 a livello alto
                                     // dopo una pausa di 500ms
        delay(500);
}
```

inizializzazione: è un'espressione che inizializza un ciclo.

In un ciclo guidato da un indice, esso viene dichiarato ed inizializzato qui.

Le variabili dichiarate in questa parte sono locali e cessano di esistere al termine dell'esecuzione del ciclo.

condizione: è la condizione valutata a ogni ripetizione. Deve essere un'espressione booleana o una funzione che restituisce un valore booleano (i<10 ad es.). Se la condizione è vera si esegue il ciclo, se falsa l'esecuzione del ciclo termina.

incremento: è un'espressione o una chiamata di funzione. Solitamente è utilizzata per modificare il valore dell'indice o per portare lo stato del ciclo fino al punto in cui la condizione risulta falsa e terminare.

Esempio: l'esempio seguente può essere considerato come parte di un programma utilizzato per variare l'intensità di luminosità di un diodo led.

```
void loop( ){
        for(int i=0; i<255;i++){
                analogWrite(PWMPin,i);
                delay(150);
        }//for
}
```

*Ciclo **while***

Il ciclo while esegue ripetutamente un'istruzione o un blocco per tutto il tempo in cui una data condizione è vera.

Sintassi

```
while(condizione) {
   corpo di istruzioni;
}
```

La condizione è un'espressione booleana: se il suo valore è true (vera), le istruzione specifiche nel corpo vengono eseguite e la condizione è nuovamente valutata; si continua così finché la condizione diventa falsa.

Il corpo del ciclo può essere un blocco di istruzioni ma può anche essere una sola istruzione.

Esempio

```
var=0;
while(var<200){
     avanti( );    //chiama un metodo che fa andare avanti due motori
     var++;
}
```

*Ciclo **do/while***

Nel ciclo while la condizione viene verificata prima dell'esecuzione del corpo di istruzioni (questo significa che se la condizione è falsa il corpo non viene eseguito), nel ciclo **do... while** invece la condizione viene verificata sempre dopo avere eseguito il corpo di istruzioni, che viene eseguito sempre almeno una volta.

Sintassi

```
do {
   corpo di istruzioni;
 } while(condizione);
```

Il corpo consiste nelle istruzioni da eseguire durante la ripetizione mentre la condizione è un'espressione booleana che se vale true il corpo viene eseguito se vale false si esce dal ciclo.

Nota bene: il corpo di istruzioni viene eseguito almeno una volta.

Esempio di lettura da un sensore, utilizzando un ciclo do/while fino a quando questo non supera un certo valore

```
do{
        delay(50);
        x=readSensor( );          //metodo che legge l'uscita da un sensore
} while(x<100);
```

il sensore viene comunque letto una volta anche se la sua uscita è superiore a 100.

break

break è un'istruzione che viene utilizzata con lo scopo di uscire da un ciclo, do, while oppure for bypassando le condizioni normali che regolano l'esecuzione del ciclo.

È usato anche per uscire da un'istruzione switch.

Esempio

```
for ( x=0; x < 255 ;  x++ ){
   digitalWrite(PWMpin, x );
   sens = analogRead ( sensorPin ) ;
   if (sens < soglia){
      x = 0 ;
      break ;
   }
}
```

continue

L'istruzione **continue** salta il resto dell'iterazione corrente di un ciclo (do,for,while). Continua testando le condizioni delle espressioni del ciclo e procedendo con le altre iterazioni seguenti.

Esempio

```
for(x=0; x<255; x++){
        if(x>40 && x<120){          // crea un salto nell'iterazione
                continue;           // esce dal ciclo e riprende dalla istruzione successiva
        }
        digitalWrite(Led,HIGH);
        delay(50);
}
```

return

Consente di uscire dalla funzione corrente ed eventualmente restituisce se richiesto dalla funzione chiamante, un valore di quella funzione.

L'istruzione `return` è usata per *ritornare* da una funzione terminata, oppure per *restituire un valore* di ritorno generato dalla funzione stessa se richiesto.

Sintassi

```
return;
return valore;
```

dove valore rappresenta un valore di variabile tipo.

Esempio in cui return restituisce un intero, in questo caso 0.

```
int testSensore( ){
   if(analogRead(0)>400){
      return 1;
  }else{
     return 0;
  }
}
```

goto

L'istruzione **goto** consente di effettuare salti incondizionati da un punto all'altro del codice.

Permette quindi di trasferire il flusso dati in un punto del programma definito da una label (con label si indica una semplice parola).

Un'attenta scrittura del codice può evitare di ricorrere a istruzioni goto, anche se è a volte necessario introdurre variabili aggiuntive. L'assenza di goto rende comunque il codice più facile da analizzare.

In molti linguaggi di programmazione l'utilizzo di goto viene scoraggiato e se proprio si deve si consiglia di utilizzarlo in modo giudizioso.

Sintassi

```
label:
        void svolgiFunzione( );
parte del programma.
goto label;      //invia il flusso dati nella posizione della label
```

#define

È una direttiva che permette al programmatore di assegnare un valore costante prima della fase della compilazione.

L'istruzione #**define** è un'istruzione al preprocessore. In poche parole, quando il compilatore ha a che fare con una direttiva #define, legge il valore assegnato alla costante (anche se non è propriamente una costante, in quanto non viene allocata in memoria), cerca quella costante all'interno del programma e gli sostituisce il valore specificato in real-time.

Con la *const*, invece, si crea una vera e propria variabile a sola lettura in modo pulito e veloce. Dichiarare una costante è di gran lunga preferito rispetto alla #define.

Operatori aritmetici

operatore di assegnamento =

Memorizza il valore a destra del segno di uguale nella variabile che si trova alla sinistra del segno uguale.

Il segno di = è chiamato operatore di assegnamento ed ha un significato differente da come si utilizza nel caso di equazioni aritmetiche. Questo operatore chiama il microcontrollore a valutare sempre il valore o l'espressione che si trova a destra del segno uguale e lo memorizza nella variabile a sinistra del segno uguale.

La variabile che si trova sul lato sinistro del segno di assegnamento deve essere in grado di memorizzare il valore. Se non è abbastanza grande da poter memorizzare questo valore , il valore memorizzato nella variabile sarà non corretto.

Attenzione: non confondere il segno = di assegnamento con il simbolo di confronto == che valuta se due espressioni sono uguali.

Addizione(+), sottrazione (-), moltiplicazione (*) e divisione (/)

Queste operazioni restituiscono la somma, la differenza, il prodotto e il quoziente tra due operandi.

L'operazione viene svolta utilizzando il tipo di data degli operandi, così per es., 9/4 dà 2 poiché 9 e 4 sono interi.

Questo significa anche che le operazioni possono portare in overflow il risultato in quanto il risultato da memorizzare potrebbe essere maggiore di quanto la variabile possa memorizzare (es. aggiungere 1 a 32767 fa -32768 a causa del roll-over e quindi del traboccamento dei bit in eccesso).

Se gli operandi appartengono a tipi di data differenti, il risultato è del tipo più capiente.

Se uno dei due operandi sono di tipo float o double, per il calcolo verrà utilizzato la rappresentazione di tipo a virgola mobile.

Sintassi

```
result = valore1+valore2;
result = valore1-valore2;
result = valore1*valore2;
result = valore1/valore2;
```

dove valore1 e valore2 sono variabili o costanti.

Esempi

```
y=y+3;          // a y sostituisci il valore di y+3
x=x - 7;        // memorizza in x il valore che aveva inizialmente a cui si sottrae 7
i=j*6;
r=g/3;
```

I numeri interi coprono valori che vanno da -32768 a 32767 (-2^{15} a $2^{15}-1$) per cui far

60*1000 porterebbe a valori oltre i 32767 e quindi l'overflow dei bit con conseguente risultato negativo del prodotto.

Nota bene: Scegliere sempre variabili che siano in grado di memorizzare i risultati dei calcoli.

Per le funzioni math che richiedono l'utilizzo di frazioni si consiglia di utilizzare variabili di tipo float, portando in conto lo svantaggio che una variabile di gran sezione rallenta la velocità di computazione.

Per convertire una variabile in un altro tipo si utilizza la tecnica del casting, vista già in precedenza

Es. per trasformare un numero di tipo float in un numero di tipo si fa così : (int)NumFloat

Modulo %

Il modulo calcola il resto di una divisione quando un intero è diviso per un altro intero. Spesso utilizzato per gestire variabili entro un certo range (es. sezione di un array).

Sintassi

result=dividendo%divisore

dove il dividendo è il numero che deve essere diviso e il divisore è il numero che divide.

Restituisce il resto della divisione.

Esempi

```
x=7%5;    //  x contiene 2
x=9%4;    //  x contiene 1 in quanto 9/4=2 con resto 1
```

Esempio di codice con %

```
/*
aggiorna un valore in un array ogni volta che attraversa un ciclo */
int valori[10];
int i = 0;

void setup( ) {
}

void loop( )
{
        valori[i] = analogRead(0);
        i = (i + 1) % 10;              // operatore modulo - variabile rolls over
                                       // ogni 10 volte aggiorna il valore di valori
}
```

l'operatore % non lavora su tipi float.

Operatori di confronto

Gli operatori di confronto vengono utilizzati normalmente insieme all'istruzione if che testa se una certa condizione è stata ricevuta, così come un certo input abbia raggiunto un certo valore. Il formato **per un test di if è**

```
if  (Variabile > 50)
{
        // svolgi istruzioni
}
```

che testa se una variabile è superiore a 50.

Le istruzioni che saranno valutate tra parentesi utilizzano quindi i seguenti operatori di confronto

x = = y *// x uguale a y*
x ! = y *// x diverso da y*
x < y *// x minore di y*
x > y *// x maggiore di y)*
x < = y *// x minore o uguale a y*
x > = y *// x maggiore o uguale a y*

Attenzione a non confondere l'operatore di confronto = = con quello di assegnamento = come precedentemente sottolineato.

Operatore booleano

gli operatori booleani possono essere utilizzati all'interno di una condizione di if

&& operatore AND

La condizione sarà verificata se entrambi gli operandi sono veri

Sintassi

```
if (condizione1 && condizione2){
        //esegui istruzioni
}
```

Esempio

```
if (digitalRead(2) == HIGH && digitalRead(3) == HIGH) {
        // legge i valori ad esempio di due interruttori
        // esegue delle istruzioni
}
```

questa condizione è vera se entrambe le condizioni sono vere.

|| *operatore OR*

La condizione sarà verificata se almeno una condizione è verificata.

Sintassi

```
if (condizione1  ||  condizione2){
        //  esegui istruzioni
}
```

Esempio

```
if (x > 0  ||  y > 0) {
        // se x>0 oppure y>0 esegui delle istruzioni
}
```

vera se x o y sono maggiori di 0.

! *operatore not*

Vero se l'operando è falso

```
if  ( !x ) {
        // esegui istruzioni
}
```

vera se x è falsa

Attenzione

Operatore logico	Operatore tra bit
&& indica AND logico	**&** indica AND tra bit
|| indica OR logico	**|** indica OR tra bit
! indica il NOR logico	~ indica il NOR tra bit

Non confondere gli operatori logici con gli operatori che lavorano con singoli bit.

Le due tipologie di operatori sono nettamente differenti.

Operatori tra bit (su interi a 16 bit)

AND tra bit &

In questo caso l'operazione di AND avviene a livello di bit.

Lavora su ogni posizione di bit indipendentemente, applicando la regola dell'AND: se entrambi i bit sono uguali a 1 il risultato, su singolo bit, è 1, altrimenti è 0.

Un modo per esprimere l'AND tra bit è riassunto come segue:

0 0 1 1 operando1

0 1 0 1 operando2

---

0 0 0 1 (operando1 & operando2) – risultato su singolo bit

In Arduino le espressioni di interi utilizzano 16 bit per cui l'applicazione dell'operatore & tra bit provoca il confronto simultaneo tra due numeri a 16 bit.

Esempio di frammento di codice

```
int a = 92;          // in binario :  0000000001011100
int b = 101;         // in binario :  0000000001100101
int c = a & b;       // risultato  :  0000000001000100, o 68 in decimale.
```

Ognuno dei 16 bit sono processati utilizzando l'AND tra bit e tutti e 16 i risultati vengono memorizzati nella variabile c corrispondente a 68 decimale.

Uno dei maggior utilizzi dell'operatore tra bit AND è quello di selezionare un particolare bit (o particolari bit) utilizzando la tecnica del masking . *Vedi in seguito per un esempio*

Operatore OR tra bit |

L'operatore OR tra bit (barra verticale) lavora su ogni bit indipendentemente dalla loro combinazione.

Il bit di uscita è 1 se almeno uno dei due operandi è 1.

0 0 1 1 operando1

0 1 0 1 operando2

---

0 1 1 1 (operando1 | operando2) - risultato

Esempio di OR tra bit :

```
int a = 92;          // in binario: 0000000001011100
int b = 101;         // in binario: 0000000001100101
int c = a | b;       // risultato:  0000000001111101, o 125 in decimale
```

Un comune impiego degli operatori AND e OR tra bit è quello di leggere, modificare e scrivere su una porta. Nei microcontrollori una porta è rappresentata da un numero a 8 bit che rappresenta le condizioni dei pin. Scrivendo sulla porta si controllano lo stato di tutti i pin.

In ATMEGA di Arduino ad esempio la porta PORTD, si riferisce allo stato dei pin digitali 0,1,2,3,4,5,6,7. Se è presente 1 in una certa posizione vuol significare che quel pin si trova allo stato HIGH. (I pin sono necessariamente richiedono di essere settati come output con il comando pinMode()). Così la scrittura PORTD=B00110001 significa aver reso i pin 0 ,4 e 5 a livello alto.

Una leggera difficoltà è che, in questo modo, sono stati cambiati anche lo stato dei pin 0 e 1 che Arduino utilizza per la comunicazione seriale interferendo così con tale comunicazione.

Per esempio l' algoritmo per un programma che accende dei diodi LED collegati ai pin digitali 2–7, con tutte le diverse combinazioni si può ottenere nel modo che segue:

Data la PORTD azzeriamo solo i bit corrispondenti ai pin che vogliamo tenere sotto controllo con un AND tra bit.

Combiniamo i valori modificati di PORTD con i nuovi valori per i pin sotto controllo (con OR tra bit). Il numero binario DDRD rappresenta il registro di direzione dei bit della porta D: 1 indica output, 0 input.

```
int j;
void setup( ){
      DDRD = DDRD | B11111100;     // imposta a 1 i bit dei pin dal 2 al 7,
                                   // lascia i bit dei pin 0 e 1 inalterati (xx||0 0=x x)
                                   // alla stessa maniera di pinMode(pin,OUTPUT)
                                   // dei pin dal 2 al 7
       Serial.begin(9600);
}
void loop( ){
       for (int i=0; i<64; i++){
             PORTD = PORTD & B00000011;      // cancella i bit 2 - 7, lascia
                                                 // inalterati i pin 0 e 1 (xx & 11 == xx)
             j = (i << 2);                   // shifta la variabile i  di due bit per comandare
                                             // solo i bit che vanno dal 2 al 7
                                             // per evitare i pin 0 e 1
             PORTD = PORTD | j;    // combina  le informazioni della porta con le
                                   // nuove informazioni dei pin dei led
             Serial.println(PORTD, BIN);     // controllo della maschera
             delay(100);
       }
}
```

Operatore tra bit XOR (^)

L'uscita è uguale a 1 se entrambi gli ingressi sono uguali, è 0 se i due bit sono diversi.

```
0 0 1 1     operando1
0 1 0 1     operando2
------------
0 1 1 0     (operando1 ^ operando2) = risultato
```

Esempio di codice :

```
int x = 12 ;              // binario: 1100
int y = 10 ;              // binario: 1010
int z = x ^ y ;           // binario: 0110, o decimale 6
```

Esempio di lampeggiamento di un diodo led su pin 5 utilizzando OR Esclusivo tra bit

```
void setup( ){
      DDRD = DDRD | B00100000;        // imposta il pin digitale 5 come OUTPUT
      Serial.begin(9600);
}
void loop( ){
      PORTD = PORTD ^ B00100000;    // inverti il bit 5 , lascia gli altri bit inalterati
      delay(100);
}
```

Operatore NOT tra bit ~

Come gli altri operatori si applica ai bit del singolo operando

Ogni bit viene negato come segue

```
0 1     operando1
------------
1 0   ~ operando1
```

```
int a = 103;              // binario:   0000000001100111
int b = ~a;               // binario:   1111111110011000 = -104 decimale
```

Il risultato negativo -104 è dovuto al fatto che il bit più significativo 1 rappresenta il segno on questo caso quindi coincidente con il -.

Scorrimento verso sinistra (<<) e verso destra (>>)

Lo scorrimento in pratica fa scorrere verso destra o verso sinistra i bit di un certo numero di posizioni specificato dall'operando che si trova a destra del comando.

Sintassi

variable << numero di bit

variable >> numero di bit

La variabile può essere di tipo byte, int, long mentre il numero di bit da shiftare < = di 32

Esempio:

int a = 5; *// binario: 0000000000000101*

int b = a << 3; *// binario: 0000000000101000, o 40 in decimale*

int c = b >> 3; *// binario: 0000000000000101, tornati alla condizione di partenza 5*

Quando si fa lo shift verso sinistra di un valore x di y bits (x << y), occorre far attenzione che non vadano persi i bit più significativi.

Per esempio

int a = 5; *// binario: 0000000000000101*

int b = a << 14; *// binario: 0100000000000000 – in questo caso il primo bit a // sinistra 1 in 101 è stato perso.*

Se si è certi che nessuno dei bit 1 durante l'operazione di shift viene perso l'operazione di shift verso sinistra significa moltiplicare per 2 elevato al numero di spostamenti effettuati.

Esempio

1<< 0 == 1 ($1=2^0$)

1<< 1 == 2 ($1=2^1$)

1<< 2 == 4 ($1=2^2$)

1<< 3 == 8 ($1=2^3$)

…

1<<8 ==256 ($1=2^8$)

Quando lo shift invece si fa verso destra di un certo numero y di cifre (x>>y) e il bit più significativo è 1 il comportamento dipende dalla tipologia del dato.

Se x è di tipo intero il bit più significativo rappresenta il segno e su questo occorre discutere.

In questo caso il segno viene riproposto sempre nel bit più significato e spesso questo comportamento non è quello voluto (problema dell'estensione del segno)

int x = - 16; *// binario: 1111111111110000*

int y = x >> 3; *// binario: **111**1111111111110 estensione del segno*

questo inconveniente si risolve nel modo che segue:

int x = -16; *// binary: 1111111111110000*

int y = (unsigned int) x >> 3; *// binary: 0001111111111110*

Se si è sicuri di evitare il problema dell'estensione del segno si può utilizzare la tecnica dello shift verso destra per dividere per 2 elevato alla potenza dello spostamento

effettuato.

Esempio

```
int x = 1000;
int y = x >> 3;        // divisione intera di 1000 per  8, dà luogo a y = 125.
```

Operatori composti

++ (incremento) -- (decremento)

Incrementa o decrementa una variabile

Sintassi

```
x++ ;   // il valore di x viene reso prima disponibile e poi viene incrementato
++x ;   // il valore della variabile x prima viene incrementato di 1 e poi reso disponibile
x-- ;   // il valore della variabile x viene prima reso disponibile e poi decrementato
--x ;   // il valore della variabile x viene prima decrementato e poi reso disponibile
```

x è una variabile di tipo int oppure long (possibilmente senza segno)

Il valore restituito è il valore di partenza o il nuovo valore incrementato o decrementato della variabile.

Esempio

```
x = 2;
y = ++x ;      // x ora  contiene 3, y contiene 3
y = x -- ;     // x contiene 2 , y contiene 3
```

*+= -= *= /=*

Esegue un'operazione matematica su una variabile insieme ad un'altra variabile o costante.

Sintassi

```
x += y;          // equivalente all'espressione x = x + y;
x -= y;          // equivalente all'espressione x = x - y;
x *= y;          // equivalente all'espressione x = x * y;
x /= y;          // equivalente all'espressione x = x / y;
```

Parametri

x: qualsiasi variabile

y: qualsiasi variabile o costante

Esempi

```
x = 2;
x += 4;          // x ora contiene 6
x -= 3;          // x ora contiene 3
```

x *= 10; *// ora contiene 30*
x /= 2; *// x ora contiene 15*

AND composto tra bit (&=) e OR composto tra bit (| =)

Le composizioni tra operatori a livello di bit sono spesso utilizzati per cancellare e settare uno specifico bit di una variabile.

AND composto (& =)

L'operatore AND composto è spesso utilizzato con una variabile e una costante per forzare un particolare bit di una variabile allo stato basso.

Questa operazione è spesso indicata con il nome di "resettaggio" di bit.

Sintassi

x &= y ; *// equivalente a x = x & y;*

Il parametro x può essere di tipo char, o una variabile di tipo int o long , mentre il parametro y una costante intera o char, int, o long

Esempio

Prima di tutto ricordiamoci come funziona l'operatore AND (&) tra bit

0 0 1 1 operando1

0 1 0 1 operando2

---

0 0 0 1 (operando1 & operando2)

I bit che sono & con 0 sono posti a 0, mentre i bit che sono & con 1 non cambiano

mioByte & B00000000 = 0;

mioByte & B11111111 = mioByte;

Nota: poiché si sta trattando di operatori tra bit è preferibile usare i formattatore binario con le costanti.

Ad esempio per resettare a 0 i bit nella posizione 0 e 1di una variabile, lasciando inalterato il resto, usa l'operatore &= con la costante B11111100

1 0 1 0 1 0 1 0 variabile

1 1 1 1 1 1 0 0 maschera

---

1 0 1 0 1 0 0 0 risultato memorizzato nella variabile di partenza anche se i bit sono stati azzerati

Di seguito un esempio con i bit sostituiti dal simbolo x

x x x x x x x x variabile

1 1 1 1 1 1 0 0 mask

---

x x x x x x 0 0 variabile

Così se

mioByte = 10101010;
mioByte &= B1111100 == B10101000;

OR composto (|=)

L'operatore composto di OR tra bit (|=) è spesso usato tra una variabile e una costante che setta (mette a 1) particolari bit in una variabile.

Sintassi

x |= y ; // equivalente a x = x | y;

Il parametro x è una variabile di tipo char, int long

y invece una costante di tipo integer o char, int, o long

Esempio

Prima di tutto ricordiamo il comportamento dell'operatore tre bit OR (|)

0 0 1 1 operando1
0 1 0 1 operando2
---
0 1 1 1 risultato

I bit che sono OR con 0 rimangono inalterati, quelli OR con 1 sono settati a 1. Per cui:

mioByte | B00000000 = mioByte;
mioByte | B11111111 = B11111111;

Per esempio per settare i bit in posizione 0 e 1 di una variabile, lasciando inalterati gli altri valori , si può utilizzare l'operatore composto (|=) tra bit con la costante B00000011

1 0 1 0 1 0 1 0 variabile x
0 0 0 0 0 0 1 1 maschera
---
1 0 1 0 1 0 1 1 variabile x

Di seguito stessa rappresentazione, solo che al posto dei bit vengono posti i simboli x

x x x x x x x x variabile x
0 0 0 0 0 0 1 1 maschera
---
x x x x x x 1 1 variabile x

Così :

mioByte = B10101010;
mioByte |= B00000011 == B10101011;

Funzioni

I/O digitali

pinMode(...); digitalWrite(...); analogWrite(...);

pinMode(pin,mode);

pin numero del pin di cui si vuole impostare la modalità ingresso o uscita

mode sempre INPUT o OUTPUT

Esempio

```
int ledPin=10;      //led connesso al pin digitale 10

void setup( ){
        pinMode(ledPin,OUTPUT);    //imposta  il pin digitale come uscita (output)
}
```

I pin analogici possono essere usati come pin digitali purché ci si riferisca con i nomi A0, A1...A5.

digitalWrite

Scrive un valore HIGH o LOW su un pin impostato come digitale.

Se il pin è stato impostato come OUTPUT con il pinMode(), la sua tensione sarà impostata al corrispondente valore di 5V per HIGH e 0V per LOW.

Se il pin è impostato come INPUT tramite il pinMode(), scrivendo un valore HIGH con digitalWrite() si abiliterà un resistore interno di pullUp da 20K. Scrivendo basso sarà disabilitato il pullup. Il resistore di pullup è sufficiente per far illuminare un led debolmente, per cui sembra che il diodo lavori anche se debolmente (questa è una probabile causa).

Il rimedio è di impostare il pin su un'uscita con il metodo pinMode

Attenzione: il **pin 13** è più difficile da utilizzare come input digitale rispetto agli altri pin poiché ha già collegati un diodo led e un resistore sulla board. Se si abilita il resistore di pullup esso viene portato a 1,7V invece dei 5V attesi poiché si deve tener conto della serie di questi due componenti e questo significa che restituisce sempre LOW (Arduino legge sempre 0). Per ovviare a questo inconveniente occorrerebbe inserire esternamente un resistore di pull down.

digitalWrite(pin,valore)

pin è il pin di cui si vuole impostare il valore

valore HIGH o LOW

Esempio

```
int ledPin=13;  // led connesso al pin digitale  13

void setup( ){
        pinMode(ledPin, OUTPUT);      // imposta il pin digitale come output
}
void loop( )
{
digitalWrite(ledPin, HIGH);           // accende il LED
delay(1000);                          // aspetta per 1 secondo
digitalWrite(ledPin, LOW);            // spegni il  LED
delay(1000);                          // attendi 1 secondo
}
```

I pin analogici possono essere usati come pin digitali purché ci si riferisca con i nomi A0, A1...A5.

digitalRead

Legge il valore da un pin digitale specifico.

Sintassi

```
digitalRead(pin);
```

pin è il numero intero del pin di cui si vuole leggere il valore alto o basso.

Esempio

```
int ledPin = 13;            //  LED connesso al pin digitale13
int inPin = 7;              //  pulsante connesso al pin digitale7
int val = 0;                //  variabile che memorizza  il valore letto

void setup( ) {
        pinMode(ledPin, OUTPUT);          // imposta il  pin digitale 13 come uscita
        pinMode(inPin, INPUT);            // imposta il pin digitale 7 come ingresso
}

void loop( ){
        val = digitalRead(inPin);         // legge il pin di ingresso e lo memorizza in val
        digitalWrite(ledPin, val);        // imposta il  LED a seconda dell'azione svolta
                                          // sul pulsante
 }
```

Se il pin non è connesso per un qualsiasi motivo digitalRead() può restituire sia HIGH che LOW in modo del tutto casuale.

I pin analogici possono essere usati come pin digitali purché ci si riferisca con i nomi A0, A1...A5.

I/O analogici

analogReference(type)

Configura la tensione di riferimento utilizzata per gli ingressi analogici (cioè il valore massimo del range dei valori di ingresso). Le opzioni sono:

DEFAULT: il valore analogico di riferimento è 5V sulle schede di Arduino a 5V oppure 3.3V su quelle a 3.3V.

INTERNAL: 1,1V sull'ATmega168 o Atmega328 e 2.56 V sull'ATmega8.

EXTERNAL: la tensione applicata al pin AREF è usata come riferimento.

Type quindi può essere DEFAULT, INTERNAL ed EXTERNAL.

Attenzione: se si vuole utilizzare un riferimento di tensione esterno, applicato al pin AREF, occorre impostare il riferimento analogico EXTERNAL tramite analogRead(). Altrimenti si metterà in corto con la tensione attiva interna causando un danneggiamento della scheda Arduino.

Alternativamente si può collegare al pin di riferimento esterno della tensione AREF un resistore di 5k e poter così scegliere tra tensione interna e tensione esterna. Si nota che il resistore modificherà la tensione usata come riferimento esterno poiché sul pin AREF c'è internamente un resistore da 32k. I due resistori esplicano il compito di partitore di tensione ad esempio se la tensione applicata è di 2.5V sul pin AREF ci arriva 2.5*32/(32+5)=2,2V circa.

analogRead()

Legge il valore da un pin di tipo analogico. La scheda Arduino contieni 6 canali (8 nel mini e Nano, 16 sul Mega) convertitori analogici digitali a 10 bit.

Questo significa che 5 V vanno suddivisi da 0 a 1023 per cui la risoluzione sarà di 5V/1023 unità =4.9mV per unità. Il range di partenza e la risoluzione possono essere modificate agendo su analogReference().

Occorrono circa 100 microsecondi per leggere da un input analogico per cui la frequenza di lettura è circa 10000 volte in un secondo.

Sintassi

analogRead(pin);

pin: numero del pin analogico da leggere che va da 0 a 5 sulla maggior parte delle schede, da 0 a 7 sulla Mini e nano, da 0 a 15 sul Mega).

Restituisce un intero da 0 a 1023.

Se l'input analogico non è connesso per qualche motivo il valore fornito da analogRead fluttuerà in funzione di diversi fattori.

Esempio

```
int analogPin = 3;          // Cursore del potenziometro ( terminale centrale) connesso
```

```
                        // al pin 3 analogico ; gli altri due morsetti rispettivamente a
                        // massa e a +5V
int val = 0;            // variabile che memorizza il valore letto

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

void loop( )
{
   val = analogRead(analogPin);      // legge il valore del pin di ingresso
   Serial.println(val);              // visualizza il  valore
}
```

analogWrite( )

Scrive un valore analogico (PWM) su un pin.

Può essere utilizzato per far illuminare un led variandone l'intensità luminosa oppure per comandare un motore e variarne la velocità.

Dopo la chiamata del metodo analogWrite(), sul pin verrà generata un'onda quadra regolare con un duty cycle specifico fino alla prossima chiamata di analogWrite() o di digitalWrite(), o di digitalRead() sullo stesso pin.

La frequenza del segnale PWM è all'incirca 490Hz.

Sulla maggior parte delle schede di Arduino questa funzione lavora sui pin 3,5 6,9,10 e 11.

Sulle altre schede potrebbero esserci delle varianti.

I pin analogici non necessitano di essere impostati prima della chiamata di analogWrite() con pinMode nel setup.

Sintassi

analogWrite(pin,valore);

Il parametro pin rappresenta il pin su cui scrivere;

il parametro valore rappresenta il valore del duty cycle tra 0 (sempre off) e 255(sempre on).

Nota:

La pwm in uscita sui pin 5 e 6 ha duty cycle maggiori di quanto aspettato.

Questo a causa di interazioni con i metodi millis() e delay() che condividono lo stesso timer interno usato per generare il segnale PWM.

Questo fa sentire maggiormente il suo peso per bassi valori di duty-cycle (0-10% circa) e potrebbe causare il non perfetto azzeramento in modalità di turn-off.

Esempio

Setta l'uscita sul LED proporzionalmente al valore letto da un potenziometro.

```
int ledPin = 9;                  // LED connesso al pin digitale 9
int analogPin = 3;               // potenziometro connesso al pin analogico 3
int val = 0;                     // variabile per memorizzare il valore letto
void setup( )
{
  pinMode(ledPin, OUTPUT);     // imposta il pin come uscita
}
void loop( )
{
   val = analogRead(analogPin);     // legge il pin di input con valori  da 0 a 1023
   analogWrite(ledPin, val / 4);      // i valori di analogWrite  vanno da 0 a 255
}
```

I/O avanzati

tone( )

Genera su un pin un'onda quadra con la specificata frequenza (il duty cycle rimane sempre al 50%). La durata può essere specificata, altrimenti l'onda continuerà fino alla chiamata del metodo noTone().

Il pin può essere connesso con un piezo buzzer un altro speaker che lavora sui toni.

Può essere generato solo un tono a volta.

Se un tone() sta già lavorando su un pin, la chiamata di tone() su un altro pin non avrà alcun effetto. Se tone() sta lavorando sullo stesso pin, la sua chiamata imposterà il valore della sua frequenza.

L'uso di tone() interferirà con l'uscita PWM dei pin 3 e 11.

Nota: per ottenere diverse intonazioni o pin multipli, occorre chiamare noTone() su un pin prima di chiamare tone sul prossimo pin.

Sintassi

tone(pin,frequenza);

tone(pin,frequenza, durata);

pin: è il pin sul quale s vuole generare il tono

frequenza: è il valore della frequenza del tono in hertz

durata: durata del tono in ms (opzionale).

noTone( )

Ferma la generazione dell'onda quadra generata dalla chiamata di tone(). Non ha alcun effetto se non è stato generato in precedenza un tono.

Sintassi

noTone(pin);

dove con pin ci si riferisce al pin sul quale si vuole fermare la generazione dell'onda quadra.

shiftOut( )

shift out porta fuori un byte di dati un bit a volta.

Inizia dal bit più significativo (quello più a sinistra) o da quello meno significativo (quello più a destra).

Ogni bit è scritto in sequenza su un dato pin, dopo che un pin di clock ha ricevuto l'impulso per indicare che quel bit è disponibile.

Sintassi

shiftOut(PinDati,PinClock,bitOrder,valore)

dove il parametro PinDati è il pin sul quale vengono posti in uscita ogni bit (int);

PinClock è il pin di toggle una volta che il pindati è stato impostato al corretto valore (int);

bitOrder definisce secondo quale ordine vengono inviati i bit; o MSBFIRST o LSBFIRST (Most Significant Bit First o Least Significant Bit First);

valore rappresenta il byte dati da mandar fuori.

La funzione non restituisce alcun valore.

Si ricorda che il PinDati e il PinClock devono essere impostati come output nel setup() tramite la chiamata della funzione pinMode().

shiftOut attualmente è chiamato a inviare in uscita 1 byte (8 bit) così richiede due passi per inviare uscite di valore maggiore di 255.

```
// Per il seriale MSBFIRST a partire dal più significativo
int dato = 500;   //dato è un numero di oltre 8 bit
// shift out del byte più alto
shiftOut(PinDati, PinClock, MSBFIRST, (dato >> 8 ));
// shift out del più basso
shiftOut(PindDati, PinClock, MSBFIRST, dato);
// Per shiftout a partire dal byte meno significativo LSBFIRST
dato = 500;
// shift out byte più basso
shiftOut(PinDati, PinClock, LSBFIRST, dato);
// shift out del byte più alto
shiftOut(dataPin, clock, LSBFIRST, (dato >> 8));  //si specifica che dato>>8 è la parte da shiftout
```

Esempio
Controllo di un multiplexer (shift register) 74HC595

```
//Pin connesso al latch del 74HC595
   int latchPin = 8;
// Pin connesso al clock del  74HC595
   int clockPin = 12;
```

```
// Pin connesso al pin dati del  74HC595
  int dataPin = 11;

void setup( ) {      //impostazione dei pin di output in quanto devono essere indirizzati nel loop
        pinMode(latchPin, OUTPUT);
        pinMode(clockPin, OUTPUT);
        pinMode(dataPin, OUTPUT);
}
void loop( ) {
                                        //routine di conteggio in avanti
for (int j = 0; j < 256; j++) {
        digitalWrite(latchPin, LOW);  // durante la trasmissione il  latchPin deve essere collegato a
                                        //massa e mantenuto a livello basso
        shiftOut(dataPin, clockPin, LSBFIRST, j);
        digitalWrite(latchPin, HIGH);    //  riporta il pin del latch al segnale alto in quanto non è
                                          // più necessario attendere informazioni
        delay(1000);
        }
}
```

pulseIn( )

Legge un impulso (alto o basso) su un pin.

Per esempio se il valore è HIGH, pulseIn() attende che il pin diventi alto, inizia il conteggio, quindi aspetta che il pin diventi basso e ferma il tempo.

Restituisce la lunghezza dell'impulso in microsecondi. Restituisce 0 se nessun impulso inizia entro uno specifico tempo.

Gli intervalli di lavoro di questa funzione sono stati determinati empiricamente e quindi probabilmente saranno soggetti a errori per impulsi lunghi.

pulseIn lavora su impulsi che vanno da 10 microsecondi fino a 3 minuti di durata.

Sintassi

pulseIn(pin, valore);

pulseIn(pin,valore, timeout);

pin indica il pin (numero intero) di Arduino sul quale si va a leggere la durata dell'impulso;

valore si riferisce al tipo da leggere cioè se alto (HIGH) o basso(LOW);

timeout è opzionale ed indica il numero in microsecondi di attesa per l'impulso di partenza; il valore di default è 1secondo(senza segno long).

La funzione restituisce la durata dell'impulso, tipo long senza segno, in microsecondi oppure restituisce 0 se l'impulso non parte prima del tempo di timeout.

Esempio

```
int pin = 7 ;  // pin sul quale arriva un segnale di tipo impulsivo (potrebbe essere
               // anche il segnale inviato da un telecomando  IR )
unsigned long durata;   // durata è il nome della variabile in cui memorizzare
                        // l'ampiezza dell'impulso alto o basso che esso sia.
```

```
void setup( )
{
        pinMode(pin, INPUT);   // il pin deve ricevere un'informazione impulsiva per cui
                               // impostato di tipo INPUT
}
void loop( )
{
        durata = pulseIn(pin, HIGH);    // misura dell'ampiezza in microsecondi
                                        // dell'impulso in questo caso a livello alto
}
```

Time

mills( )

Restituisce il numero di millesecondi di tipo unsigned long da quando Arduino ha iniziato a far funzionare un certo programma.

Questo numero andrà in overflow, tornando a 0, dopo circa 50 giorni.

Esempio

```
unsigned long tempo ;
void setup( ){
        Serial.begin(9600);
}
void loop( ){
        Serial.print ("Tempo: ");
        tempo = millis( );        //stampa il tempo da quando il programma funziona
         Serial.println(tempo);
        delay(1000);              // attesa di 1 secondo in modo da non inviare una gran
                                  // quantità di dati da visualizzare su PC
}
```

La variabile definita per millis() deve essere un “unsigned long” altrimenti se si cercasse di forzare matematicamente ad altro tipo potrebbe verificarsi un errore.

micros( )

Restituisce il numero di microsecondi da quando Arduino ha iniziato a far funzionare il programma corrente.

Questo numero andrà in overflow dopo circa 70 minuti.

Per schede di Arduino a 16MHz (uno e Nano) questa funzione ha una risoluzione di 4 microsecondi (il valore restituito è sempre multiplo di 4).

Esempio

```
unsigned long tempo_micro;
void setup( ){
        Serial.begin(9600);
}
void loop( ){
        Serial.print("Tempo: ");
        tempo_micro = micros( );
        Serial.println(time);        // stampa tempo_micro da quando il programma è
                                     // iniziato
        delay(1000);                 // attesa di 1 secondo in modo da non inviare una gran
                                     // quantità di dati
}
```

delay( )

Mette in pausa il programma per un certo intervallo di tempo in millesecondi specificato da un parametro.

Sintassi

delay(ms)

ms è il numero di millisecondi di pausa di tipo unsigned long (long senza segno).

Esempio

```
int ledPin = 13;      // LED connesso al pin digitale 13 già presente su Arduino
void setup( )
{
        pinMode(ledPin, OUTPUT);     // imposta il pin digitale come output
}
void loop( )
{
        digitalWrite(ledPin, HIGH);     // imposta allo stato alto il pin digitale led acceso
        delay(1000);                    //  aspetta 1 secondo
        digitalWrite(ledPin, LOW);      //  spegni LED
        delay(1000);                    // attendi 1 secondo
}
```

Attenzione

È semplice creare un led che lampeggia con la funzione di delay() e molti programmi usano delay corti come switch.

In effetti l'uso del delay() in uno sketch comporta degli inconvenienti.

Nessuna lettura di sensori, calcoli matematici, o manipolazioni di pin può essere attivo durante un'operazione di delay(), questo porta la maggior parte delle attività in sosta.

Molti evitano l'utilizzo del delay per gli eventi legati alla temporizzazione di eventi di durata superiore al decimo di secondo a meno che lo sketch Arduino sia veramente semplice.

Alcune funzionalità di Arduino procedono nel loro corso anche se la funzione delay() sta

bloccando il chip Atmega, in quanto tale funzione delay() non disabilita gli interrupt.

La comunicazione seriale che appare al pin RX viene registrata, i valori della PWM(analogWrite) e lo stato dei pin sono mantenuti, e gli interrupt funzionano come devono.

delayMicroseconds()

Ferma il programma per una certa quantità di microsecondi specificato da un parametro.

Attualmente il valore maggiore che produce un accurato delay è 16383. Questo valore può modificarsi con le future release di Arduino.

Per ritardi maggiori intorno ad un centinaio di microsecondi, conviene utilizzare delay().

Sintassi

delayMicroseconds(us)

us: numero di microsecondi di pausa (int di tipo unsigned- senza segno)

Questa funzione non restituisce alcun valore.

Esempio

```
int outPin = 8 ;                    // pin digitale 8 di uscita
void setup( )
{
        pinMode(outPin, OUTPUT);    // imposta il pin di uscita
}
void loop( )
{
        digitalWrite(outPin, HIGH); // imposta il pin allo stato alto
        delayMicroseconds(50);      // metti in pausa per  50 microsecondi
        digitalWrite(outPin, LOW);  // imposta il pin allo stato basso
        delayMicroseconds(50);      // metti in pausa per 50 microsecondi
}
```

Il programma appena scritto configura il pin 8 come pin di uscita e trasmette un treno di impulsi con un periodo di 100 microsecondi.

Problemi noti

Questa funzione lavora in modo preciso in un range che parte da 3 microsecondi in su. Non è possibile affermare che delayMicroseconds() sarà preciso per tempi di valore inferiori.

delayMicroseconds() non disabilita gli interrupts.

Funzioni matematiche Math

min(x,y)

Calcola il minimo tra due numeri.

Parametri: x e y sono due numeri di qualsiasi tipo

Restituisce il numero più piccolo tra i due

Esempio

```
sensVal = min(sensVal, 100);     // assegna a sensVal il più piccolo tra il valore
                                 // precedente sensVal e 100 assicurando che non venga
                                 // mai superato 100.
```

Attenzione:

Probabilmente in un contatore, max() è usato per costringere l'estremo inferiore di un range di valori mentre min() è utilizzato per costringere l'estremo superiore del range.

A causa del modo in cui la funzione min() è implementata, evitare di usare altre funzioni all'interno delle parentesi,altrimenti si potrebbe ottenere un risultato errato.

```
min(a++, 100);      // questo modo di scrivere produce un risultato non corretto

a++;
min(a, 100);        // questo modo di scrivere mantiene altre funzioni matematiche
                    //al di fuori di questa funzione
```

max(x,y)

Determina il massimo tra due numeri.

Parametri: x e y sono rispettivamente il primo e il secondo numero, di qualsiasi tipo di dati.

Restituisce il valore maggiore tra i due.

Esempio

```
sensVal = max(senVal, 20);      // assegna alla variabile sensVal il valore maggiore tra
sensVal o 20
                                // assicurandosi che  sia al massimo 20
```

Attenzione:

Probabilmente in un contatore, max() è usato per costringere l'estremo inferiore di un range di valori mentre min() è utilizzato per costringere l'estremo superiore del range.

A causa del modo in cui la funzione max() è implementata, evitare di usare altre funzioni all'interno delle parentesi, altrimenti si potrebbe ottenere un risultato errato.

```
max(a--, 0);    // evitare questa situazione – risultato  non corretto
```

```
a--;            // usa questo invece
max(a, 0);      // lascia fuori altre funzioni matematiche
```

abs(x)

Calcola il valore assoluto di un numero

Il parametro x è un numero di qualsiasi tipo

La funzione restituisce x se x è maggiore o uguale a 0; restituisce -x se x è minore di 0

Attenzione:

Utilizzare una funzione all'interno delle parentesi come parametro, potrebbe portare a risultati non corretti.

```
abs(a++);       // da evitare – potrebbe portare a risultati scorretti
a++;            // questo è consentito
abs(a);         // lascia fuori le altre funzioni matematiche
```

constrain(x,a,b)

Sintassi

type valore=constrain(x,a,b);

Questa funzione costringe un numero a stare dentro un certo range.

Il parametro x è il numero da costringere entro il range e può essere di qualsiasi tipo;

a è l'estremo inferiore del range e può essere di qualsiasi tipo di dati

b è l'estremo superiore del range e può essere di qualsiasi tipo di dati

La funzione restituisce x se x è tra a e b;

restituisce a se x è minore di a;

restituisce b se x è maggiore di b.

Esempio

```
sensVal = constrain(sensVal, 10, 150);      // limita il range del valore del sensore
                                            // tra 10 e 150
```

map(valore, daValoreBasso, daValoreAlto, aValoreBasso, aValoreAlto)

Ri-mappa un numero da un range ad un altro range. Cioè, un valore di daValoreBasso sarà mappato in uno di aValoreAlto, un valore di daValoreAlto in aValoreAlto, i valori tra daValoreAlto e daValoreBasso in valori tra aValoreAlto e aValoreBasso.

Sintassi

map(valore,daValoreBasso,daValoreAlto,aValoreBasso,aValoreAlto);

valore: è il numero da mappare

daValoreBasso: limite inferiore del range corrente

daValoreAlto: limite superiore del corrente range

aValoreBasso: limite inferiore del range obiettivo

aValoreAlto: limite superiore del range obiettivo

La funzione restituisce il valore mappato.

Non costringe i valori entro un certo range, poiché i valori fuori range sono a volte premeditati e utili. La funzione constrain() può essere utilizzata o prima o dopo questa funzione se sono desiderati dei limiti di range.

Nota: il limite inferiore di un range può essere più grande o più piccolo del limite superiore così la funzione map() può essere usata per invertire il range di un numero.

Esempio

```
y = map(x, 1, 50, 50, 1);
```

La funzione può trattare anche numeri negativi, così come in questo esempio:

```
y = map(x, 1, 50, 50, -100);
```

è valido e lavora bene.

La funzione map() usa numero interi così non potrà generare frazioni, quando si costringe a restituire una frazione questa sarà troncata, non approssimata.

Esempio

```
void setup( ) {
}
void loop( )
{
        int val = analogRead(0);
        val = map(val, 0, 1023, 0, 255);
        analogWrite(9, val);
}
```

pow(base,esponente)

Questa funzione calcola il valore di un numero elevato ad un certa potenza.

pow() può essere usata per elevare un numero ad una potenza frazionaria.

Questa funzione è usata spesso per far la mappatura esponenziale di valori o curve (interpolazione esponenziale).

Sintassi

```
double valore=pow(base,esponente);
```

valore è il risultato dell'elevamento a potenza

base è il numero di tipo float che rappresenta la base della potenza.

esponente è la potenza di cui la base è elevata di tipo float

La funzione restituisce il risultato dell'elevamento a potenza in formato double.

sqrt(x)

Calcola la radice quadrata di un numero.

Sintassi

double valore=sqrt(x);

valore è il risultato della chiamata della radice quadrata;

il numero x è un qualsiasi tipo di dato.

La funzione restituisce la radice quadrata di x in formato double.

Trigonometria

sin(rad)

Calcola il seno di un angolo espresso in radianti. Il risultato sarà compreso tra -1 e 1.

Sintassi

double valore=sin(rad);

valore è il risultato della funzione sin;

rad rappresenta l'angolo in radianti di tipo float;

La funzione restituisce il seno dell'angolo in formato double.

cos(rad)

Calcola il coseno di un angolo espresso in radianti. Il risultato sarà compreso tra -1 e 1.

Sintassi

double valore=cos(rad);

valore è il risultato della funzione cos;

rad: angolo in radianti di tipo float

La funzione restituisce il coseno dell'angolo in formato double.

tan(rad)

Calcola la tangente di un angolo espresso n radianti. Il risultato sarà compreso tra meno infinito e più infinito.

Sintassi

double valore=tan(rad);

valore è il risultato della funzione tan;

rad è l'angolo espresso in radianti di tipo float

La funzione restituisce la tangente dell'angolo espresso in double.

Numeri casuali (Random numbers)

randomSeed(seme) seed=seme

randomSeed() inizializza il generatore di numeri pseudo casuali, causando la partenza in un arbitrario punto di una propria sequenza random. Questa sequenza, sebbene molto lunga e random, è sempre la stessa.

Se è importante avere una sequenza di valori generati da random() , in una sottosequenza di uno sketch, si usa randomSeed() per inizializzare la generazione di numeri casuali con un input propriamente random, come un analogRead() o un pin non connesso.

Può accadere che si possa usare una sequenza pseudo-random che si ripeta esattamente. Questa può realizzarsi chiamando la funzione randomSeed() con un numero fisso, prima di far partire la sequenza random.

Il parametro seme può essere di formato sia long che int, ed è il numero passato alla funzione che genera il seme (il punto di partenza)

La funzione non restituisce alcun valore.

Esempio

```
long randNumber;
void setup( ){
        Serial.begin(9600);
        randomSeed(analogRead(0));   // seme puramente casuale
}
void loop( ){
        randNumber = random(300);
        Serial.println(randNumber);
        delay(50);
}
```

random( )

Questa funzione genera numeri pseudo-random

Sintassi

random(max)

random(min,max)

min è limite inferiore di valori random, incluso (opzionale)

max è limite superiore dei valori random, escluso

La funzione restituisce un numero random tra i valori min e max-1 (long)

Esempio

```
long randNumber;
void setup( ){
        Serial.begin(9600);
        // se il pin0 analogico non risulta connesso, il rumore analogico casuale genererà differenti
        // semi nella chiamata di randomSeed( ) ogni volta che lo sketch viene chiamato.
        // randomSeed( ) quindi darà luogo a diverse funzioni random.
        randomSeed(analogRead(0));
}
void loop( ) {
        randNumber = random(300);               // stampa un numero random da 0 a 299
        Serial.println(randNumber);

        randNumber = random(10, 20);            // stampa un numero random da 10 a 19
        Serial.println(randNumber);
        delay(50);
}
```

Bits and Bytes

lowByte( )

Estrae il byte più a destra di una variabile (ad esmpio di una variabile di tipo word).

Sintassi

byte valore = lowByte(x);

Il parametro x è un valore di qualsiasi tipo

La funzione restituisce un valore di tipo byte.

highByte( )

Estrae il byte più a sinistra di una parola (word) o il secondo più basso byte di un dato di tipo di data più grande del word.

Sintassi

byte valore=highByte(x)

dove il parametro x è un valore di qualsiasi tipo

La funzione restituisce un valore di tipo byte

bitRead( )

Questa funzione legge un bit di un numero per cui restituisce il valore del bit 0 oppure 1

Sintassi

int valore = bitRead(x,n)

dove valore può assumere 0 oppure 1, x rappresenta il numero da cui leggere e il parametro n è il bit da leggere, partendo da 0 per il bit meno significativo (quello più a destra)

bitWrite( )

Scrive un bit di una variabile numerica

Sintassi

bitWrite(x,n,b)

dove il parametro x rappresenta la variabile numerica sulla quale scrivere, n rappresenta la posizione dove scrivere il bit, iniziando dalla posizione meno significativa (il più a destra); b è il bit da scrivere 0 oppure 1.

La funzione non restituisce alcun valore di ritorno.

bitSet( )

Pone in set (scrive 1) un bit di una variabile numerica.

Sintassi

```
bitSet(x,n);
```

dove il parametro x indica la variabile sulla quale settare il bit; il parametro n indica il bit da settare, a partire dal bit meno significativo (lo 0 coincide con il bit più a destra).

bitClear( )

Resetta (scrive 0) un bit di una variabile numerica.

Sintassi

```
bitClear(x,n)
```

dove x è la variabile numerica di cui porre a zero il bit ed n indica la posizione del bit da resettare a partire dallo 0 coincidente con il bit meno significativo (quello più a destra).

La funzione non restituisce alcun valore.

bit( )

Calcola il valore di ogni specifico bit(bit 0 è 1, bit 1 è 2, bit 2 è 4, …)

Sintassi

```
int valore=bit(n);
```

dove il parametro n è il bit di cui si vuole calcolare il valore.

La funzione restituisce il valore del bit.

External Interrupts Interruzioni esterne.

attachInterrupt(interrupt,funzione,modo)

Specifica la funzione da chiamare quando avviene un'interruzione esterna(interrupt).

Sostituisce ogni altra precedente funzione che è stata collegata all'interrupt.

La maggior parte delle schede Arduino hanno due interrupt esterni: numero 0 (sul pin digitale 2) e numero 1 (sul pin digitale 3).

L'Arduino mega ne ha in aggiunta 4: numero 2 (pin 21), 3(pin 20), 4(pin19), 5(pin18).

Parametri

interrupt: numero dell'interrupt (int)

funzione: è la funzione che viene chiamata quando si verifica l'interrupt; questa funzione non deve avere parametri e non deve restituire alcun valori; a volte è rinviata, indirizzata ad una routine di gestione degli interrupt.

modo definisce quando l'interrupt deve essere attivato.

Quattro costanti sono predefinite come valori validi per modo:

LOW per attivare l'interrupt ogni volta che il pin è basso

CHANGE per attivare l'interrupt ogni volta che il pin cambia valore

RIOSING per attivare l'interrupt quando il pin passa da basso ad alto

FALLING per attivare l'interrupt quando il pin passa da valore alto a basso.

La funzione non restituisce alcun valore.

Nota

delay() non può far parte delle funzioni attaccate alla gestione delle interrupt, non può lavorare e il valore restituito dalla funzione millis() non verrà incrementato.

I dati seriali ricevuti durante la funzione possono essere sbagliati.

Qualsiasi variabile modificabile all'interno della funzione attached si può dichiarare come volatile.

Utilizzo degli interrupt

Gli interrupt sono utili per far fare cose che succedono automaticamente nei programmi per microcontrollori, e possono risolvere un certo tipo di problemi.

Un esempio di utilizzo di un interrupt può essere la lettura di un encoder di rotazione.

Per essere sicuri che un programma catturi l'impulso di un encoder rotativo, senza perdere un impulso, il programma potrebbe risultare troppo complicato in quanto dovrebbe costantemente controllare il sensore di linea dell'encoder, per catturare gli impulsi quando avvengono.

Anche altri sensori richiedono un controllo costante e continuo siffatto, per esempio leggere un sensore di suono che deve catturare (accorgersi) un click, oppure un sensore IR che deve catturare l'evento di caduta di una moneta.

In tutte queste situazioni usare un interrupt può alleggerire il microcontrollore che può dedicarsi ad altri compiti senza dimenticarsi di rimanere all'erta.

Esempio

```
int pin = 13;
volatile int state = LOW;
void setup( )
{
        pinMode(pin, OUTPUT);
        attachInterrupt(0, blink, CHANGE);
}
void loop( )
{
        digitalWrite(pin, state);
}
        void blink( )  // ogni volta che si chiama questa funzione cambia lo stato della
                       // variabile state
{
        state = !state ;
}
```

detachInterrupt(interrupt)

Descrizione: Spegne un dato interrupt.

Il parametro interrupt definisce il numero dell'interrupt da disabilitare (0 oppure 1)

Interrupts

interrupts( )

Riabilita gli interrupt (dopo che essi sono stati disabilitati da noInterrupts()).

Gli interrupts permettono ad alcuni importanti eventi di avvenire in background e sono abilitati per defalut.

Alcune funzioni non lavorano mentre gli interrupts sono disabilitati e le comunicazioni in arrivo potrebbero essere ignorate.

Gli interrupt possono leggermente variare il tempo del codice e possono essere disabilitati per particolari sezioni critiche del codice.

Esempio

```
void setup( ) { }
void loop( ){
        noInterrupts( );
        // … sezione critica del codice
        interrupts( );
        // altro codice }
```

noInterrupts

Disabilita interrupts (che sono stati abilitati con interrupts( )).

```
void setup( ) { }
void loop( ) {
        noInterrupts( );
        // sezione critica del codice
        interrupts( );
        // altro codice da eseguire
}
```

Comunicazione

Serial

Usata per la comunicazione tra Arduino e il computer o altri dispositivi.

Tutte le schede Arduino hanno almeno una porta seriale (conosciuta come UART o USART): Serial.

Essa comunica con i pin0 (RX) e 1(TX) alla stessa maniera con cui comunica con il computer via USB (la porta usb è collegata a questi pin vedi schema).

Se si stanno utilizzando le funzioni legate a Serial, non si possono usare i pin 0 e 1 per gli I/O digitali.

Si può utilizzare l'ambiente monitor seriale definito nell'ambiente di sviluppo di Arduino per comunicare con la scheda Arduino, cliccando sul bottone serial monitor (icona in alto a destra , dopo upload) della toolbar dello sketch e selezionando la stessa velocità in baud (bit al secondo) utilizzato nella chiamata di begin().

L'Arduino Mega ha tre porte seriali aggiuntive: Serial1 sul pin 19 (RX) e 18 (TX), Serial2 sul pin 17(RX) e 16(TX), Serial3 sul pin 15 (RX) e 14(TX). Per usare questi pin per comunicare con il personal computer, sarà necessario aggiungere un adattatore USB-Seriale, poiché questi non sono connessi con l'adattatore USB-Seriale del Mega.

Per utilizzare questi per comunicare con dispositivi esterni seriali in logica TTL, connettere il pin TX con il pin RX del dispositivo, e la massa del Mega alla terra del dispositivo. Non connettere questi pin direttamente con la porta seriale RS232 del computer; essa opera a +/- 12V e può danneggiare la scheda Arduino.

begin()

Imposta la velocità di trasmissione dati in bit al secondo(baud) per la trasmissione seriale.

Per comunicare con il computer ci sono diverse velocità ammissibili: 300, 1200, 2400, 4800, 9600, 14400, 57600, 115200. Si può comunque specificare altre velocità – per esempio per comunicare altri pin 0 e 1 a componenti che richiedono una particolare velocità.

Sintassi

```
Serial.begin( velocità)
```

Solo per Arduino Mega

```
Serial1.begin(velocità)
Serial2.begin(velocità)
Serial3.begin(velocità)
```

il parametro velocità rappresenta la velocità in bit per secondi (baud) ed è dichiarata come tipo long.

La funzione non restituisce alcun valore.

Esempio

```
void setup( ) {
        Serial.begin(9600);      // apri la porta seriale e imposta la velocità a 9600 bps
}
void loop( ) { }
```

Esempio per Arduino Mega :

```
// Per usa tutte e quattro le porte seriali
// (Serial, Serial1, Serial2, Serial3),  con differenti velocità di trasmissione

void setup( ){
        Serial.begin(9600);
        Serial1.begin(38400);
        Serial2.begin(19200);
        Serial3.begin(4800);
        Serial.println("Hello Computer");
        Serial1.println("Hello Serial 1");
        Serial2.println("Hello Serial 2");
        Serial3.println("Hello Serial 3");
}
void loop( ) { }
```

end( )

Disabilita la comunicazione seriale, permettendo ai pin RX e TX di essere utilizzati come I/O generali.

Per riabilitare la comunicazione seriale basta richiamare la funzione Serial.begin().

Sintassi

```
        Serial.end( ) ;
```

Solo per Arduino Mega:

```
        Serial1.end( ) ;
        Serial2.end( ) ;
        Serial3.end( ) ;
```

Questa funzione non restituisce alcun valore e non richiede alcun parametro.

available( )

Restiuisce il numero di byte disponibili per leggere dalla porta seriale.

Sintassi

```
        Serial.available( )
```

Per Arduino Mega :

Serial1.available( )
Serial2.available( )
Serial3.available( )

La funzione non richiede alcun parametro e restituisce un valore di tipo byte necessari per la lettura.

Esempio

```
int byteInArrivo = 0;                    // per iniziare la trasmissione seriale
void setup( ) {
        Serial.begin(9600);              //  apertura porta seriale, impostazione velocità
                                         // di trasmissione dati a 9600bps
}
void loop( ) {
        if (Serial.available( ) > 0) {
                byteIniziale = Serial.read( );          // leggi i byte che arrivano
                Serial.print("Ricevuto: ");
                Serial.println(byteInArrivo, DEC);      // scrive i dati ricevuti
}
```

Esempio per Arduino Mega:

```
void setup( ) {
        Serial.begin(9600);
        Serial1.begin(9600);
}
void loop( ) {
        // legge dalla porta 0, invia sulla porta 1:
        if (Serial.available( ) ) {
                 int inByte = Serial.read( );
                Serial1.print(inByte, BYTE);
  }
  // legge dalla porta 19, invia sulla porta 18:
        if (Serial1.available( ) ) {
                int inByte = Serial1.read( );
                Serial.print(inByte, BYTE);
  }
}
```

read( )

Legge i dati in entrata dalla comunicazione seriale.

Sintassi

Serial.read( )

Solo per Arduino Mega :

Serial1.read( )
Serial2.read( )
Serial3.read( )

La funzione non necessita di alcun parametro e restituisce il primo byte disponibile sulla comunicazione seriale (tipo int) oppure -1 se non c'è alcun dato inviato.

Esempio

```
int byteInArrivo = 0;              // per iniziare la trasmissione seriale
void setup( ) {
        Serial.begin(9600);        //  apertura porta seriale, impostazione velocità di
                                   // trasmissione dati a 9600bps
}
void loop( ) {
        if (Serial.available( ) > 0) {

byteInArrivo = Serial.read( );              // leggi i byte che arrivano
Serial.print("Ricevuto: ");
Serial.println(byteIniziale, DEC);       // scrive i dati ricevuti
}
```

flush( )

Svuota il buffer dei dati seriali in entrata. Cioè, ogni chiamata di Serial.read() o Serial.available() restituirà solo i dati ricevuti dopo tutte le più recenti chiamate di Serial.flush().

Sintassi

Serial.flush( )

Per Arduino Mega :

Serial1.flush( )
Serial2.flush( )
Serial3.flush( )

La funzione non richiede alcun parametro restituisce alcun valore.

print( )

Questa funzione stampa sulla porta seriale un testo ASCII leggibile dall'uomo.

Questo comando può assumere diverse formati: i numeri sono stampati usando un carattere ASCII per ogni cifra, i float sono scritti allo stesso modo e per default a due posti decimali, i byte vengono inviati come caratteri singoli mentre i caratteri e le stringhe sono inviati come sono.

Sintassi

Serial.print(valore)

Serial.print(valore,formato)

valore indica il valore da scrivere di qualsiasi tipo mentre formato indica la base del sistema numerico (per i dati di tipo integer) o il numero di posti decimali per i tipi floating point.

La funzione non restituisce alcun valore.

Esempi :

```
Serial.print(78)          // dà come risultato  "78"
Serial.print(1.23456)     // dà come risultato "1.23"
Serial.print(byte(78))    // dà come risultato "N" (il valore  ASCII di N è 78)
Serial.print('N')         // dà come risultato "N"
Serial.print("Ciao mondo.")  // dà come risultato "Ciao mondo."
```

Un secondo parametro opzionale specifica la base, il formato in uso; i valori permessi sono BYTE, BIN (binario e base 2), OCT(ottale o base 8), DEC (decimale o base 10), HEX (esadecimale o base 16).

Per i numeri a virgola mobile, questo parametro specifica il numero di posti decimali da utilizzare.

Esempi :

```
Serial.print(78, BYTE)       // stampa  "N"
Serial.print(78, BIN)        // stampa "1001110"
Serial.print(78, OCT)        // stampa "116"
Serial.print(78, DEC)        // stampa "78"
Serial.print(78, HEX)        // stampa "4E"
Serial.println(1.23456, 0)   // stampa "1"
Serial.println(1.23456, 2)   // stampa "1.23"
Serial.println(1.23456, 4)   // stampa "1.2346"
```

Esempio:

```
/*
Uso di un ciclo FOR per ottenere e stampare dati in diversi formati.
*/
int x = 0;                   // variabile da stampare
void setup( ) {
```

```
        Serial.begin(9600);          // impostazione porta seriale a 9600 bps
}
void loop( ) {
        Serial.print("NO FORMAT");     // stampa di label per la visualizzazione su PC
        Serial.print("\t");                  // stampa un tab (spazio vuoto)
        Serial.print("DEC");
        Serial.print("\t");
        Serial.print("HEX");
        Serial.print("\t");
        Serial.print("OCT");
        Serial.print("\t");
        Serial.print("BIN");
        Serial.print("\t");
        Serial.println("BYTE");
        for(x=0; x< 64; x++){
                Serial.print(x);           // stampa come un ASCII-  base decimale -  "DEC"
                Serial.print("\t");
                Serial.print(x, DEC);    // stampa come un ASCII-  base decimale -  "DEC"
                Serial.print("\t");
                Serial.print(x, HEX);      // stampa come un ASCII-  base esadecimale
                Serial.print("\t");
                Serial.print(x, OCT);      // stampa come un ASCII-  base ottale
                Serial.print("\t");
                Serial.print(x, BIN);      // stampa come un ASCII-  base binaria
                Serial.print("\t");
                Serial.println(x, BYTE);  // stampa come riga di valori byte con ritorno di
                                          // carrello ( println )
                delay(200);               // delay 200 millisecondi
        }
        Serial.println("");               // stampa un altro ritorno a capo
}
```

Avvertimento

L'ultimo carattere che deve essere scritto è trasmesso sulla porta seriale dopo che Serial.print() è terminato. Questo potrebbe essere un problema in casi particolari.

println( )

Scrive i dati sulla porta seriale come testo leggibile dall'uomo in codice ASCII seguito da un carattere di ritorno a capo (ASCII 13 oppure '\r') e un carattere di nuova linea (ASCII 10, oppure '\n').

Questo comando assume la stessa forma di Serial.print().

Sintassi

Serial.println(valore)

Serial.println(valore, formato)

Il parametro valore indica il valore da scrivere di qualsiasi tipo mentre formato indica la base del sistema numerico (per i dati di tipo integer) o il numero di posti decimali per i tipi floating point.

La funzione non restituisce alcun valore.

Esempio

```
// Legge un valore analogico dal pin 0 e lo stampa in diversi formati
int analogVal = 0;              // variabile che memorizza i valori analogici
void setup( ) {
        Serial.begin(9600);     // imposta la porta seriale a 9600 bps
}

void loop( ) {
        analogVal = analogRead(0);          // legge il valore analogico dal pin0
                                            // stampa in diverso formato
        Serial.println(analogVal);                  // stampa come ASCII- base 10
        Serial.println(analogValue, DEC);           // stampa come ASCII- base 10
        Serial.println(analogValue, HEX);           // stampa come ASCII- base 16
        Serial.println(analogValue, OCT);           // stampa come ASCII- base 8
        Serial.println(analogValue, BIN);           // stampa come ASCII- base 2
        Serial.println(analogValue, BYTE);          // stampa come riga di BYTE
        delay(10);                                  // attendi 10 ms prima della
                                                    // prossima lettura
}
```

write( )

Scrive i dati binari sulla porta seriale.

Questi dati sono inviati come byte o serie di byte; per inviare i caratteri rappresentati le cifre di un numero usare invece la funzione print().

Sintassi

Serial.write(valore)

Serial.write(stringa)

Serial.write(buf,len)

Arduino Mega supporta anche : Serial1, Serial2, Serial3 al posto di Serial.

Il parametro valore è il valore da trasmettere come singolo byte; il parametro stringa è una stringa da spedire come serie di byte; il parametro buf è un array da spedire come serie di byte; il parametro len rappresenta la lunghezza del byte.

Indice generale: Elementi base del linguaggio di programmazione di Arduino

Gli autori autorizzano l'uso del documento e la sua divulgazione a fini educativi e non a scopo di lucro.

Gli autori non si assumono alcuna responsabilità per danni diretti o indiretti provocati dall'utilizzo integrale o parziale dei contenuti presenti nel manuale.

Gli autori ringraziano quanti, con segnalazioni di errori e suggerimenti, permetteranno di migliorare le loro pubblicazioni.

Per comunicazioni dirette con gli autori:

tizianamarsella@libero.it

romano.lombardi@alice.it