

Struttura dei Programmi

Definizioni

- di variabili globali
- di costanti
- inclusioni

void setup() { statements ; }

void loop() { statements ; }

```
#define NomeDellaCostante valore
#include <NomeDellaLibreria.h>

Int NomeVariabile valore ;
```

è chiamata una volta sola, PRIMA che il programma viene fatto funzionare. E' usata per inizializzare il pinMode e la seriale e deve essere inclusa in un programma, anche se non ci sono istruzioni da eseguire

contiene il codice che deve essere eseguito ripetutamente, in essa vengono letti gli input, i segnali di output ecc...

I programmi Arduino si chiamano SKETCH (bozzetto, schizzo)

Il Linguaggio è un derivato del "C++" e si chiama WIRING (cablare, collegare con cavi)

Entrambi i termini richiamano l'uso di Arduino: far funzionare in fretta (come fare uno schizzo) dei componenti hardware (che si collegano tra loro con dei cavi)

I/O digitale

Esempi di Input Digitale in Automazione:

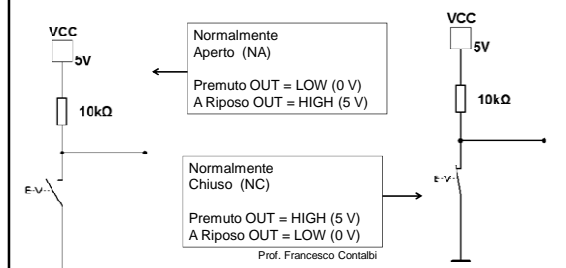
- Pulsanti
- Interruttori
- Fine Corsa
- Encoder e Sensori ON/OFF in genere
- Uscite da precedenti Circuiti Digitali

Esempi di Output Digitale in Automazione:

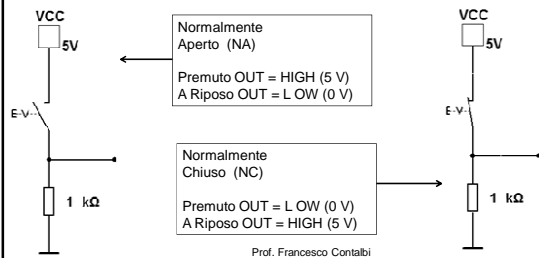
- Lampade / Segnalatori
- Led
- Relais
- Motori in modalità On/Off
- Ingressi di successivi Circuiti Digitali

Ingressi (INPUT) digitali:

- Pulsante (Push Button)
- Interruttore (Switch SPST)



Configurazioni alternative NON CONSIGLIATE



Istruzioni Gestione Switch

```
#define BUTTON 7 // pin di input dove è collegato il pulsante

int val = 0;

void setup() {

  pinMode(BUTTON, INPUT); // imposta il pin digitale come input
}

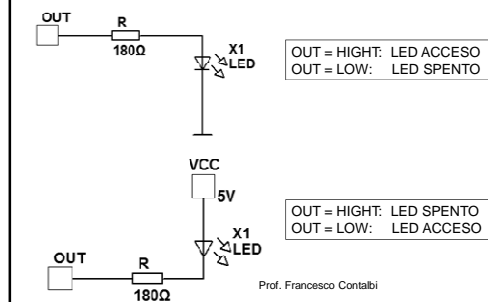
void loop() {

  val = digitalRead(BUTTON); // legge il valore del pulsante e lo conserva
}
```

Prof. Francesco Contalbi

Uscite (OUTPUT) digitali:

➤ Diodo LED (spia luminosa)



Istruzioni Gestione LED

```
#define LED 13 // LED collegato al pin digitale 13

void setup() {

  pinMode(LED, OUTPUT); // imposta il pin digitale come output
}

void loop() {

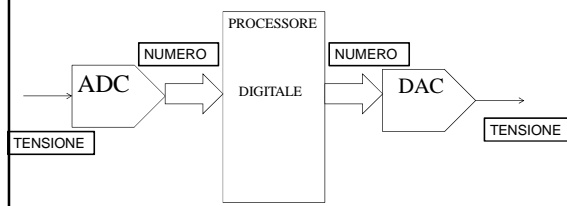
  digitalWrite(LED, HIGH); //accende il led
  digitalWrite(LED, LOW); //spegne il led
}
```

Prof. Francesco Contalbi

I/O Analogico

Segnali Analogici

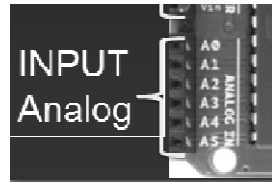
Una macchina digitale accetta in Input numeri e invia all'Output numeri. Occorre dunque un Convertitore da Analogico a Digitale (A/D o ADC) in Ingresso e un Convertitore da Digitale ad Analogico (D/A o DAC) in Uscita



Ingressi (INPUT) Analogici:

Qualsiasi grandezza opportunamente trasformata in una tensione può essere letta da Arduino tramite uno dei 6 convertitori analogico digitali A/D (ADC) interni

- Temperatura: trasduttore di temperatura
- Rotazione: potenziometro, encoder
- Intensità luminosa: foto resistore, foto transistor
- Distanza: sensore ad infrarossi, ultrasuoni
- Inclinazione: accelerometro
- Ecc...



Con un **Ingresso Analogico** è possibile leggere tensioni **comprese** tra 0 e +5V **MASSIMI**

10 è la **RISOLUZIONE** (in bit) dei ADC interni di Arduino

Il numero intero restituito è compreso tra **0** e $2^{10} - 1 = 1023$

In totale si hanno $2^{10}=1024$ numeri (valori) possibili

Divedendo la massima tensione (+5V) applicabile all'ingresso per 1024 otteniamo una unità di **4,9 mV** (La risoluzione espressa in Volt) = MINIMA tensione misurabile

Prof. Francesco Contalbi

Gestione SW

```
#define IN_ANALOG 0 // 0, 1, 2, ...5 sono i 6 input analogici
int analogVal=0; // contiene il numero letto dal ADC sul canale
.....
```

```
void setup() {
.....
}
```

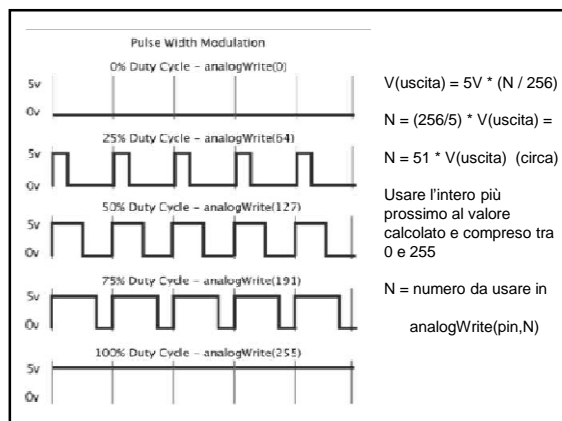
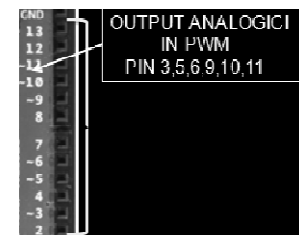
```
void loop() {
.....
  analogVal = analogRead(IN_ANALOG); // legge il valore attuale
.....
}
```

La funzione **analogRead()** riceve come parametro il numero del PIN Analogico da leggere e restituisce un numero intero assegnandolo ad una variabile.

Uscite (OUTPUT) Analogici:

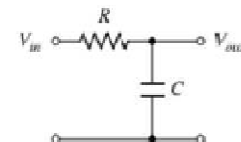
Come gli altri uC viene usata la tecnica PWM (Pulse Width Modulation) in cui la tensione voluta è il valor medio di una forma impulsiva

- Temperatura: Resistore di potenza, valvola proporzionale
- Spostamento: Motore in continua
- Intensità luminosa: lampade, LED
- Ecc...



Motori, Riscaldatori, Lampade, LED (l'occhio in realtà) e molti altri carichi sono sensibili al valor Medio e dunque non ci sono problemi

Se però il Carico NON si comporta come un FILTRO PASSA BASSO, occorre un circuito da interporre tra l'uscita di Arduino e il carico medesimo



Gestione SW

```
#define analogPin = 9; // uscita analogica connessa al pin
//digitale 9

Int val;          // conterrà il valore da 0 a 255 per il PWM
.....
void setup() {
.....
}
void loop() {
.....
analogWrite(analogPin, val); // aziono l'uscita analogica
.....
}
```

Comunicazioni Seriali con Arduino

Libreria “ Serial “

E' una libreria integrata direttamente nell'ambiente e che quindi **NON** deve essere dichiarata con `#include ...` ma di cui si possono direttamente “chiamare” le sue funzioni tramite `Serial.nomeDellaFunzione()` ;

Queste sono:

- `begin()`
- `end()`
- `available()`
- `flush()`
- `read()`
- `print()`
- `println()`
- `write()`

✓ Tutte le schede Arduino hanno almeno una porta seriale (conosciuta come UART o USART) e gestite tramite Serial.

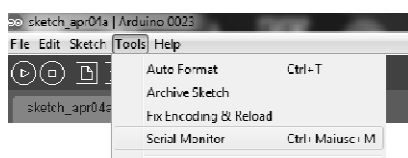
✓ Essa comunica con i pin digitali 0 (RX) e 1(TX) alla stessa maniera con cui comunica con il computer via USB (e infatti la porta Usb è collegata a questi pin).

✓ 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 il “monitor seriale” dell'IDE di Arduino per comunicare con la scheda, selezionando la stessa velocità in baud (bit al secondo) utilizzato nella chiamata di `begin()`.

✓ Non connettere questi pin direttamente con la porta seriale RS232 del computer; quest'ultima opera a +/- 12V contro i 0/5 V sui pin 0 e 1 e può danneggiare la scheda Arduino.

Il Serial Monitor



Apri un collegamento seriale sia in lettura che in scrittura con Arduino.

Una volta che questo è programmato consente di interagire direttamente con lui

`Serial.begin(velocità) ;`

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 se **NON** si collega il PC ma altri dispositivi
il parametro velocità rappresenta la velocità in bit per secondi (baud) ed è dichiarata come tipo long.

La funzione non restituisce alcun valore.

`Serial.end() ;`

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

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

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

`Serial.available() ;`

La funzione non richiede alcun parametro e restituisce un "int" il cui valore è pari al numero di byte (caratteri) disponibili nel buffer di lettura, che nella versione 2 di Arduino sono 128 al massimo.

Esempio: (da verificare !!!!!)

```
int n_buffer = 0; // N°byte nel buffer di lettura
int letto = 0; // byte letto
void setup() {
  Serial.begin(9600); // apertura porta seriale, impostazione
                      // velocità di trasmissione dati a 9600 bps
}
void loop() {
  if ((n_buffer=Serial.available()) > 0) { // se c'è qualcosa nel
    // buffer di lettura
    letto= Serial.read(); // leggi i byte che arrivano
    Serial.println("Ricevuto N°byte = "); Serial.print(n_buffer);
    Serial.println(byteInArrivo, DEC); // scrive i dati ricevuti
  }
}
```

`Serial.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()`.

La funzione non richiede alcun parametro restituisce alcun valore.

`Serial.read() ;`

Legge dalla seriale (in particolare dal serial monitor) e il dato viene portato "dentro Arduino". O meglio assegnato ad una variabile

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

```
int inByte = 0; // incoming serial byte
int outputPin = 13;
void setup(){
```

```
  // start serial port at 9600 bps:
  Serial.begin(9600);
  pinMode(outputPin, OUTPUT);
}
void loop() {
  if (Serial.available() > 0) {
    // get incoming byte:
    inByte = Serial.read();
    if (inByte == 'H') {
      digitalWrite(outputPin, HIGH);
    }else if (inByte == 'L') {
      digitalWrite(outputPin, LOW);
    }
  }else{
    Serial.print('E');
    delay(2000);
    Serial.print('F');
    delay(2000);
  }
}
```

`Serial.print(valore);`

Questa funzione stampa sulla porta seriale un testo ASCII leggibile dall'uomo. Porta cioè "fuori da Arduino", ad es. al Serial Monitor

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. 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."
```

`Serial.print(valore,formato);`

valore indica il valore da scrivere di qualsiasi tipo.
La funzione non restituisce alcun valore.

i valori permessi di formato 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"
```

`Serial.println(valore);`
`Serial.println(valore, formato);`

E' esattamente come le precedenti "print" solo che aggiunge alla fine un carattere di ritorno a capo (ASCII 13 oppure '\r') seguito da un carattere di nuova linea (ASCII 10, oppure '\n').

L'effetto è quello di "far andare a capo" se il dispositivo ricevente è un terminale che gira sul PC (come il Serial Monitor o Terminal etc.)

Attenzione se invece chi riceve i dati è un generico dispositivo !!

`Serial.write(valore);`
`Serial.write(stringa);`
`Serial.write(buf,len);`

Scrivi 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().

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 dell'Array, ossia il N° di byte che lo compongono.

FUNZIONI SPECIFICHE IN WIRING

ARDUINO

Wiring ha molte funzioni utili in libreria, ossia usabili senza averle precedentemente "create".

Ne vediamo due, molto importanti quando si interfacciano ad Arduino dei trasduttori:

`map` Trasforma un numero intero in un altro intero

`pulseIn` Legge la durata di un impulso

altroValore = map (valore, valoreMIN, valoreMAX, altroValoreMIN, altroValoreMAX)

- valore: è il numero da trasformare
- altroValore è il numero trasformato
- valoreMIN: limite inferiore del range corrente
- valoreMAX: limite superiore del corrente range
- altroValoreMIN: limite inferiore del range obiettivo
- altroValoreMAX: limite superiore del range di arrivo

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.

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

La funzione può trattare anche numeri negativi, così come in questo esempio:
y = map(x, 1, 50, 50, -100);
è valido e lavora bene.

```
/* Esempio: un trasduttore di temperatura fornisce ad Arduino da 0 a 5 V per T
da 20 a 70 °C. Indicare sul Serial Monitor la temperatura misurata */

int N=0; // valore convertito dal ADC interno: N va da 0 (per V=0 e T= 20 °C)
// a 1023 (per V= 5 V e T= 80 °C)

int T=0; // valore di T corrispondente a N

void setup() {
  Serial.begin(9600); // comunicazione seriale a 9600 bps
}

void loop() {
  N = analogRead(A0); // leggo dall'ADC A0
  T = map(N, 0, 1023, 20, 80); // N=0 T= 20 °C; N=1 023 T = 80 °C
  Serial.print(" Valore di temperatura [°C] = ");
  Serial.println(T);
  delay(3000);
}
```

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). 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

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

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.

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
}
```

ARDUINO

Programmazione

(ELEMENTI DI PROGRAMMAZIONE

IN "C")

#define nome valore

È una direttiva che permette al programmatore di assegnare un valore costante prima della fase della compilazione (è cioè una istruzione per il PREPROCESSORE e quindi NON INSERIRE il ;).

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 compile-time (cioè al momento della compilazione). Esattamente come con Word si usa il trova / sostituisci

Con la *const*, invece, si crea una vera e propria variabile a sola lettura in modo pulito e veloce. Dichiarare una costante è preferito rispetto alla #define ma occupa spazio in memoria !!

Si possono anche creare delle MACRO ma NON le vedremo.

#include nome-di-una-libreria

È una direttiva che permette al programmatore di usare una libreria che NON sia già automaticamente considerata dal compilatore. E' cioè una istruzione per il PREPROCESSORE e quindi NON INSERIRE il ;

Se il nome si trova tra < > significa che la libreria è nel percorso standard, altrimenti occorre dare "nomepercorso/nome-di-libreria"

Esempio:

```
#include <Servo.h> // userò le funzioni per gestire i Servo
```

In pratica il Compilatore legge il file come se il contenuto fosse stato scritto direttamente dove si trova #include

/*...*/ blocco commenti

I blocchi commenti, o commenti multi-linea, sono aree di testo che sono ignorate dal compilatore (non occupano dunque memoria) e sono usate per una descrizione lunga del codice oppure commenti che aiutano la comprensione di parti del programma. Il blocco commento incomincia con /* e termina con */ e può occupare più righe.

/* questo è un blocco commento chiuso; non dimenticare di chiudere il commento.

Anche i segni di blocco commento sono bilanciati */

// commenti in singola linea

I commenti in singola linea incominciano con // e terminano con la successiva linea di codice. Come per il blocco commento, anche i commenti in singola linea sono ignorati dal programma e non occupano spazio di memoria.

I commenti su un'unica linea sono spesso usati dopo un'istruzione per fornire maggiori informazioni sull'azione che compie e per futura memoria.

// questo è un commento su una sola linea

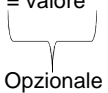
Tipi di dati

- Costanti: il loro valore NON può mai cambiare
- Variabili: il loro valore cambia durante il programma
- Semplici: composte da un solo elemento
- Array: un solo nome identifica un insieme dello stesso tipo

Dichiarazione di Variabili

Una variabile è un modo per nominare e memorizzare un valore numerico per un successivo utilizzo da parte del programma. Deve SEMPRE essere dichiarata PRIMA di usarla.

La dichiarazione implica il tipo che determina quanto spazio occupa in memoria, un nome che la identifica e un valore (temporaneo)

tipo nome = valore ;

 Opzionale

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 (e non solo " 1 " per cui anche -1, 2, -200 (valori diversi da 0) sono tutte condizioni di vero in senso booleano.

Importante: true e false sono scritte in minuscolo.

Esempio

```
boolean running = false;
.....
if (digitalRead(switchPin) == LOW) {
running = ! running; // fa il NOT su running
```

Numeri Naturali (integer)

Sono numeri senza virgola: se glieli impongo eliminano qualunque parte decimale

Nota: le variabili intere se superano il loro range vanno in Overflow, come per il contachilometri di un'auto.

Ad esempio se $x = 32.767$ e ad x aggiungiamo 1 con $x = x + 1$ o con $x++$ il nuovo valore di x sarà -32.768.

Quindi il range da 32.767 a -32.768 è da considerare non come una retta di numeri ma come una circonferenza il cui massimo e minimo sono consecutivi.

byte **NB: NON PIU' USATA NELL'IDE 1.xx usare:**

unsigned char

Memorizza numeri a 8 bit (1 byte) interi (senza decimali) e senza segno ed hanno un range da 0 a 255 (0 a $2^8 - 1$).

byte pippo = 180; // dichiara pippo come una variabile di tipo byte

char

Occupa 1 Byte con segno, quindi va da -128 a +127 (-2^7 a $2^7 - 1$). Serve soprattutto a memorizzare un carattere ASCII puro (che va da 0 a 127)

char pippo = 'A'; // contiene 65 = codice ASCII di A. => char pippo = 65

int

Gli interi sono dei tipi di dato usati per memorizzare numeri senza decimali e memorizzano valori a 16 bit (2 byte) con segno nel range da -32.768 a 32.767 (-2^{15} a $2^{15} - 1$).

int pippo = -1500; // dichiara pippo come una variabile di tipo intero

long

estende la dimensione degli interi (senza virgola) con segno, memorizzati con 32 bit (4 byte) e quindi il range dei valori possibili va da -2.147.483.648 a 2.147.483.647 (-2^{31} a $2^{31} - 1$).

long pippo = 90 000; // dichiara che pippo è di tipo long

Interi Senza Segno

Come per quelli con segno MA facendo precedere il qualificatore **unsigned al tipo**

unsigned char da 0 a 255 (0 a $2^8 - 1$).
 byte (sinonimo per unsigned char)

unsigned int da 0 a 65 535 (0 a $2^{16} - 1$).
 word (sinonimo per unsigned int)

unsigned long da 0 a 4 294 967 295 (0 a $2^{31} - 1$).

Numeri Reali o *floating point* (notazione numeri a virgola mobile)

Sono SEMPRE con segno

float

E' un tipo di dato usato per i numeri in virgola mobile ed è usato per la rappresentazione di numeri piccolissimi o grandissimi con o senza segno e con o senza decimali. I float sono memorizzati utilizzando 32 bit (4 byte) nel range tra $3,4028235E+38$ a $-3,4028235E+38$, ed hanno 6-7 cifre TOTALI
float pippo = 3.14; // dichiara che pippo è di tipo in virgola mobile

double

In Arduino coincide con float e dunque NON si ha alcun vantaggio. Su un PC di solito double occupa 64 bit aumentando sia la precisione (N° di cifre) che l'esponente (doppia precisione)
double pippo = 3E-14;

Nota: i numeri in virgola mobile non sono esatti e possono condurre a risultati errati; $(1/3)*3$ NON dà 1 !!. L'esecuzione di calcoli con tipi float è più lunga dei calcoli realizzati con tipi interi. Se la situazione lo permette evitate l'uso di variabili di tipo float, usare ad es long o unsigned long

Costanti

Si dichiarano come per le variabili MA facendo precedere il qualificatore const

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

Costanti intere

Le costanti intere sono numeri usati direttamente in uno sketch come ad es. 123. Per default questi numeri sono 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	0b1001101	prefisso B	lavora solo con 8 caratteri 0 e 1
8 (ottale)	0173	prefisso 0 (zero)	caratteri da 0 a 7
16 (esadecimale)	0x7B	prefisso 0x	caratteri 0-9, A-F, a-f

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

B11001100*256 + B10101010 dove il primo è la parte alta del numero

La moltiplicazione per 256 significa traslare tutto di 8 posti verso sinistra.

Attenzione a NON lasciare 0 prima di un numero !! Verrebbe interpretato come un Ottale !!!

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)

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

```
10.0          // 10
2.34E5        // 2.34*105 = 234000
67e-12        // 67.0*10-12 = 0.000000000067
```

```
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. Il Compilatore dà errore
```

Per creare una costante numerica o di tipo String si può utilizzare `const` o `#define`.

Per gli array occorre che venga utilizzato il termine `const`.

Costanti

char

boolean

string

Char : carattere ASCII inserito tra singoli apici
Per rappresentare l'apice usare `'\'`

`'A'` `'6'` `'.'` `'/'`

Boolean: usare `true` o `false` scritte in minuscolo

Stringhe: caratteri tra virgolette doppie.
Per rappresentare le virgolette usare `\"`

`"ciao"` `"sono una stringa"` `\"virgolette\"`

Casting

Per convertire un tipo in un altro il C e il C++ usano la:

- Conversione automatica o implicita
- Conversione forzata o esplicita o Casting
- Il C++, Java e Wiring (e molti altri) mettono a disposizione anche opportuni metodi che però non vedremo.

ATTENZIONE: si possono avere gravi errori se il tipo convertito è più "piccolo" di quello di partenza

Conversione automatica (implicita)

- Si ha quando in una operazione “mischio” numeri di tipo diverso: quello più “piccolo” viene convertito (promosso) nel tipo di quello più grosso.
- Oppure quando assegno con l' = Il tipo a destra viene convertito in quello di sinistra.

Esempio: $9 / 4.0$ oppure $9.0 / 4$ fa 2.25 perché tutto viene “promosso” a float. Se però poi faccio `int pippo` e `pippo = 2.25` `pippo` vale 2 perché è `int` e perdo i decimali (ma non ci sono errori perché 2 sta in un `int`)

Conversione esplicita: Casting

(tipoDiDatoVoluto) numeroDaConvertire

notare le ()

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

`i = (int)3.6;` sarà uguale a 3

Si perdono gli eventuali decimali e se `NumFloat` è “grosso” (ad es 80 000) si ha pure un risultato errato

Array

- E' un insieme di valori a cui si accede con un indice.
- Un valore nell'array può essere richiamato con il nome dell'array e l'indice numerico che corrisponde al numero d'ordine del valore nell'array.
- Gli array vengono indicizzati partendo dal numero zero e quindi il primo valore dell' array avrà indice 0.
- Un array deve essere dichiarato ed opzionalmente si possono assegnare i valori prima di utilizzarlo.

```
int pippo [ ] = { 10, -13, 0, 55, -21}; // array di 5 interi
```

Allo stesso modo è possibile dichiarare un array dando il tipo e la dimensione e in seguito assegnare i valori:

```
int pippo[5]; // dichiara un array 5 di interi, valori non specificati
.....
pippo[0] = 10; // assegna in 1°posizione il valore 10
pippo[4] = -21; // assegna in 5°e ultima posizione il valore -22
```

Per recuperare un valore all'interno di un array, bisogna dichiarare una variabile del tipo dell'array a cui poi viene assegnato il valore assunto allo specifico indice:

```
x = pippo[3]; // ora x ha il valore 55
```

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.

Gli array sono molto spesso utilizzati all'interno dei cicli `for`, dove il contatore di incremento è anche usato come indice posizionale per ogni valore dell'array.

L'esempio che segue viene utilizzato per l'accensione e spegnimento rapido (fliker) di un LED. Usando un ciclo `for`, il contatore incomincia da 0, scrive il valore nell'array `fliker[]` in posizione 0, in questo caso 180, al pin 10, pausa per 200ms, poi si sposta nella posizione successiva in quanto l'indice è incrementato di 1.

```

/* il ciclo for verrà eseguito fino a quando la condizione i<8 risulta vera, cioè
per valori di i che vanno da 0 a 7 compreso */

int ledPin = 13; // LED sul pin 13 => uso il led "interno"
unsigned char fliker[] = {180, 30, 255, 200, 10, 90, 150, 60}; // 8 elementi

void setup()
{
  pinMode(ledPin, OUTPUT);
}
void loop()
{
  for (int i=0; i<8; i++)          // esegue un ciclo un numero di volte
                                  // pari al numero di elementi dell'array
  {
    analogWrite(ledPin, fliker[i]); // ad ogni ciclo viene attivato ledPin
                                  // con un valore di duty cycle
                                  // corrispondente al valore indicato
                                  // dall'indice dell'array
    delay(200);                  // pausa di 200 ms.
  }
}

```

Array di char = Stringa

- Dichiarazione ed uso tipo linguaggio "C" costruendo una stringa con un array di caratteri e il carattere **null (ASCII 0 o /0) per terminarla.**
- Dichiarazione ed uso tipo linguaggio "C++" con la classe **String** che fa parte del codice di Arduino a partire dalla versione 0019. Più versatile ma con maggior occupazione di memoria: non verrà trattata

```

char Str[ ]= "arduino"; // E' il metodo più semplice: il compilatore aggiunge
                        // alla fine il valore null per terminare la stringa

char Str[8]='a','r','d','u','i','t','n','o','/0'; // esplicitamente viene aggiunto il
                                                    // carattere null (/0).

char Str[8]='a','r','d','u','i','t','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 Str[15]; // definisce un array di char lungo 15 caratteri senza nessuna
// assegnazione

char Str[8]= "arduino"; // inizializza l'array con esplicita sezione e stringa
// costante

char Str[15]= "arduino"; // inizializza l'array con lunghezza 15 lasciando
// uno spazio extra dopo la stringa di inizializzazione

```

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.

```

Attenzione !!

Senza il terminatore null il compilatore **NON** riesce a determinare la fine della stringa. Prosegue quindi a leggere le locazioni di memoria fino a che non trova, per caso, un carattere null. Si crea dunque un errore. Se la stringa è in scrittura ciò provoca la sovrascrittura di parti di memoria, proprio come quando è errato l'indice di un Array

Array di Array di char = Array di Stringhe

Poiché una stringa è un Array di char, un Array di stringhe è un Array di Array di char

Un Array di Array è detto Array a 2 dimensioni o anche **MATRICE** (ma si possono avere 3, 4 ecc dimensioni)

```
char* mieStringhe[ ]= { "Stringa 1", "Stringa 2", "Stringa 3", ...};
```

Il * che segue char (**char***) indica che questo è un array di puntatori.

I puntatori sono la forza e il tormento del C: non è necessario usarli per programmare in Wiring e **NON** lo faremo

Esempio:

```
char* mieStringhe[] = {"pippo", "pluto", "paperino"};

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

void loop() {
  for ( int i=0; i<3; i++) {
    Serial.println( mieStringhe[i] ); // stampa su monitor del pc le
    //stringhe presenti nell'array chiamato mieStringhe[]
    delay(500);
  }
}
```

Visibilità dei nomi..... o Scope

- Se una avariabile è dichiarata FUORI da qualunque funzione o { } (blocco) è una variabile pubblica (o globale). Viene cioè vista in qualunque punto del file sorgente e quindi usabile ovunque.
- Se è dichiarata DENTRO una funzione o un { } allora è privata (o locale). Viene vista solo dentro la funzione o il { }
- Se una variabile privata ha lo stesso nome di una pubblica, la privata nasconde quella pubblica che quindi non viene "toccata" nella funzione o { }

Static

Una variabile privata cessa di esistere e dunque perde il valore quando esco dalla funzione o blocco in cui è definita

Se viene dichiarata static (ad es static int pippo;) essa mantiene invece il valore che aveva ... dunque quando la funzione viene richiamata la variabile ha il precedente valore

Operatori Aritmetici

Assegnamento: =

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

La variabile che si trova sul lato sinistro del segno di assegnamento deve essere abbastanza grande in modo da poter memorizzare il valore altrimenti il risultato memorizzato nella variabile non sarà 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 (/)

L'operazione viene svolta utilizzando il tipo degli operandi, così per es., 9/4 dà 2 poiché 9 e 4 sono interi. Invece 9.0/4.0 dà 2.25 perché di tipo float, come anche 9.0/4 o 9/4.0

Le operazioni possono portare in overflow il risultato se questo è 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 differenti, il risultato è del tipo più capiente ossia float se almeno uno è float o long se almeno uno è long ecc..

Modulo %

Resto = dividendo % divisore

Calcola il resto di una divisione quando un intero è diviso per un altro intero. Spesso è utilizzato per mantenere una variabile entro un certo range (es. indice di un array o minuti in un ora ecc..). Gli operandi devono essere interi (non float)

```
x=9%4; // x contiene 1 in quanto 9/4=2 con resto 1
x = i % 10; // x conterrà sempre un numero 0 .. 9
x=j%60 // x = secondi in un minuto: 0 .. 59
```

Incremento ++ e Decremento - -

Incrementa o decrementa di 1 una variabile. PRE O POST incr/decr

```
x++; // x viene prima usato e poi aumentato di 1
++x; // x viene prima aumentato di 1 e poi usato
x--; // x viene prima usato e poi diminuito di 1
--x; // x viene prima diminuito di 1 e poi usato
```

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

```
x = 2;
y = ++x; // x ora contiene 3, y contiene 3
y = x--; // x contiene 1, y contiene 2
```

ATTENZIONE: x subisce sempre l'incremento o il decremento, ma il valore che prende in un calcolo E' DIVERSO se si usa il pre o post incr/decr

Assegnamento Composto

Le assegnazioni composte combinano le operazioni aritmetiche con assegnazioni di variabili e vengono largamente usate nei cicli for. Le assegnazioni composte più comuni (ma ve ne sono molte altre) includono:

```
x+=y // uguale a x = x + y,
x-=y // uguale a x = x - y,
x*=y // uguale a x = x * y, moltiplica x per y
x/=y // uguale a x = x / y, divide x per y
```

x*=3 moltiplica per tre il valore precedente di x e riassegna il risultato a x

Operatori Logici (Booleani)

```
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
```

Danno un valore che può essere solo vero (true) o falso (false). Sono principalmente usati nell' if while for switc ecc..

Ricordare di usare == e NON = e che false è una qualunque espressione pari a 0 e true diversa da 0

Connettivi Logici

NOT OR AND

! Operatore NOT

Vero se l'operando è falso

```
if ( !x ) { // vera se x è falsa
// esegui istruzioni
}
```

|| Operatore OR

La condizione sarà verificata se almeno una condizione è verificata.

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

```
if (x > 0 || y > 0) {
```

```
// se x>0 oppure y>0 oppure se x e y >0 esegui delle
istruzioni
}
vera se x o y sono maggiori di 0
```

&& Operatore AND

La condizione sarà verificata se entrambi gli operandi sono veri

```
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.
```

ATTENZIONE !!!

Se ci si dimentica di raddoppiare il carattere | nell'Or o il & nell'And il compilatore NON dà errore perché | e & sono altri tipi di operatori.

Dunque si ottengono risultati imprevedibili ma sicuramente errati. E' difficile scoprire tale errore

Strutture di controllo

- Decisione (o salto condizionato): if()/else switch()/case
- Iterazione o Ciclo o Loop: while(); for() do/while()
- Salto incondizionato: goto
- Sequenza ... istruzioni una .. dietro l'altra

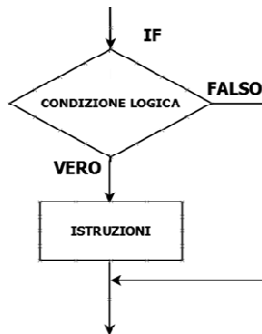
Blocco di istruzioni

Tutte le istruzioni racchiuse tra { } sono considerate come fossero un'unica istruzione

Le singole istruzioni, per facilità di lettura sono spostate a destra di alcuni spazi (Tab). Si dice identate. In C non è obbligatorio ma fortemente consigliato.

Variabili dichiarate nel blocco sono visibili solo nel blocco sono cioè private al blocco (scope)

If (condizione logica)



If (espressione logica) istruzione ;

If (espressione logica) { blocco di istruzioni ;}

L'istruzione o il blocco di istruzioni vengono eseguiti solo se la condizione logica è vera. Vero è un numero diverso da 0

Le espressioni logiche o **Booleane** sono del tipo

```

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.

Esempio di if seguito da un blocco di istruzioni

```
if (x>120) {
```

```

    digitalWrite(LED1,HIGH); // accendi LED1
    digitalWrite(LED2,LOW); // spegni LED2
    delay(2000);
    digitalWrite(LED1,LOW); // spegni LED1
    digitalWrite(LED2,HIGH); // accendi LED2
    delay(2000);
  }
```

If seguito da una sola istruzione

In questo caso si possono omettere le { }

Esempi:

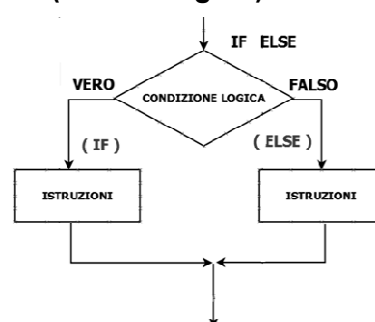
```
if (x>120) digitalWrite(LED,HIGH);
```

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

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

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

If (condiz logica) else

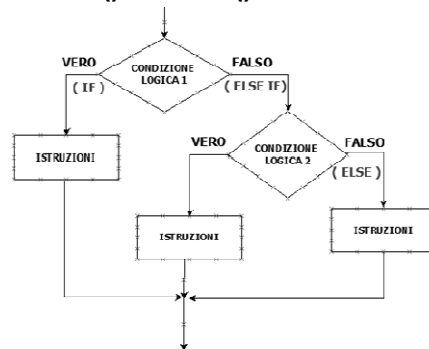


La parola else (facoltativa) introduce il blocco di istruzioni (o la singola istruzione) che deve essere eseguito nel caso in cui la condizione introdotta da if risulti falsa.

Esempio:

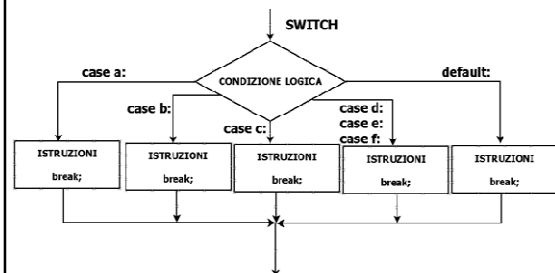
```
if(pin5<500){           // VERO
  digitalWrite(LED1,HIGH); //accendi LED1
  digitalWrite(LED2,HIGH); //accendi LED2
}
else                    // FALSO
{
  digitalWrite(LED1,LOW); //spegni LED1
  digitalWrite(LED2,LOW); //spegni LED2
}
```

else if () Gli If () NIDIFICATI:



```
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(pin8<=700) { .....
}
else
{
  digitalWrite(LED1,HIGH); // accendi LED1
  digitalWrite(LED2,HIGH); // accendi LED2
}
```

switch () ... case



Usato quando bisogna confrontare il valore di una variabile con una serie di valori costanti fino a trovare quello corrispondente. Una volta che si trova la corrispondenza vengono eseguite le istruzioni necessarie.

Con l' if il codice potrebbe diventare contorto.

In C, C++ ecc.. esiste lo **switch/case**.

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

```
switch(var){
  case label1:
    //istruzioni
    break;
  case label2:
    //istruzioni
    break;
  default:      // opzionale
    //istruzioni
}
var è la variabile che si deve confrontare con i
diversi casi
label sono i valori costanti di confronto con var
Non è necessario inserire le istruzioni tra { }
```

Esempio:

```
switch(pin5){ // pin5 è la variabile da confrontare

  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;
}
```

default è opzionale.

- Se è presente le istruzioni sotto di lui vengono eseguite se **NON** vi è stata nessuna corrispondenza coi vari case.
- Se non è presente e **NON** vi è stata nessuna corrispondenza coi vari case non viene eseguita nessuna istruzione.

break fa uscire immediatamente dallo switch (ma anche dai cicli) e viene tipicamente utilizzata alla fine di ogni clausola case.

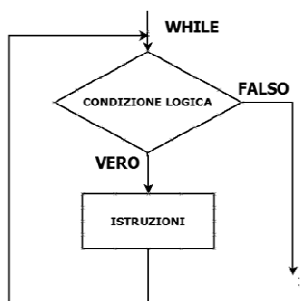
Se manca l'istruzione break, anche dopo aver trovato la condizione giusta, verrebbero eseguiti comunque i confronti successivi fino alla prima istruzione di break oppure fino alla fine dello switch. In alcuni casi questa procedura è quella voluta ma in genere si includono le istruzioni break per selezionare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

Esempio di NON uso di break

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");
}
```

Ciclo while ()



while (condizione logica) {

istruzioni ;

}

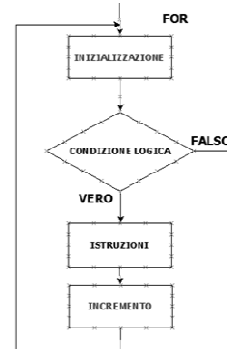
Il ciclo while esegue ripetutamente un'istruzione o un blocco per tutto il tempo in cui la condizione logica è vera. La condizione è un'espressione booleana: se il suo valore è true (vera), la o le istruzione/i presenti nel corpo (cioè tra le { }) vengono eseguite e la condizione è nuovamente valutata. 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.

ATTENZIONE !! Se per errore la condizione rimane sempre vera il programma è bloccato (usare il reset)

Esempio:

```
var=0;
while(var<200){
    avanti(); //chiama un metodo che fa andare avanti due motori
    var++; // nel primo ciclo var è 0 ma prima del secondo
           // viene incrementata e vale 1 e così via finché arriva a
           // 200 e a tal punto si esce dal while
}
```

Ciclo for



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

ATTENZIONE !! Le tre parti sono separate dal ";" e non da ",".
inizializzazione e incremento sono OPZIONALI

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.

Funziona come un while ma è più versatile perché vi sono inizializzazione e incremento direttamente inserite. Ad es for(; cl ;) è identico a while(cl) {

inizializzazione: è un'espressione che viene eseguita una sola volta, prima che il ciclo abbia inizio.

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 eseguita idealmente prima della "}". 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

```
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);
}
```

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);
    }
    .....
}
```

Ciclo do ... while



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

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.

Esempio di lettura da un sensore, utilizzando un ciclo do/while fino a quando questo non supera un certo valore: il sensore viene comunque letto una volta anche se la sua uscita è superiore a 100.

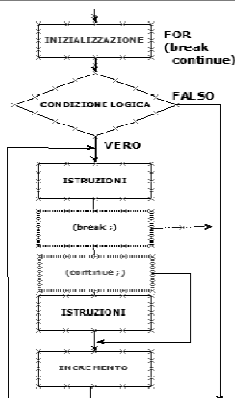
```
do{
    delay(50);
    x=readSensor(); //metodo che legge l'uscita da un sensore
} while(x<100);
```

Il do while è poco usato, tipicamente nell' 1% dei casi. Comunque ogni tanto è utile

break

continue

goto



break

E' un'istruzione che viene utilizzata nei cicli (while, for, do) e nello switch che quando incontra fa uscire immediatamente e in condizionatamente

Esempio:

```
while(x < 255 ){
    digitalWrite(PWMPin, x );
    sens = analogRead ( sensorPin ) ;
    if (sens < soglia){
        x = 0 ;
        break ; // esco immediatamente dal while
    }
    x++;
}
```

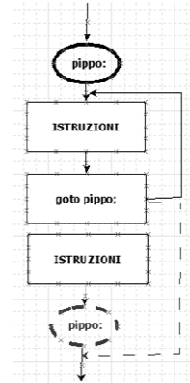
continue

E' un'istruzione che viene utilizzata nei cicli (while, for, do) che quando incontrata salta il resto dell'iterazione corrente fa ritornare immediatamente alla condizione di test nel while, esegue l'incremento e passa al test nel for

```
while(x < 255){
    digitalWrite(PWMPin, x);
    sens = analogRead ( sensorPin );
    if(sens>40 && sens< 120){ // crea un salto nell'iterazione

        continue; // esce dall'if e torna al test x< 255
    }
    x++;
}
```

goto



Consente di effettuare salti incondizionati da un punto all'altro del codice, e quindi di trasferire il flusso dati in un punto del programma definito da una label (etichetta) che poi è una semplice parola seguita da " : "

Il goto si può sempre evitare (teorema di Jacopini Bomh), anche se a prezzo di contorcimenti e all'uso di variabili aggiuntive. Dunque in alcuni (pochi) casi è utile ma normalmente l'assenza di goto rende il codice più facile da analizzare e da mantenere.

In molti linguaggi di programmazione l'utilizzo di goto viene scoraggiato e a volte... non c'è, come in Java

label: // notare i " : "

```
void svolgiFunzione( );
    istruzioni ;
    goto label; //invia il flusso dati nella posizione
                // della label che può essere
                // dappertutto nel file sorgente
```