# Arduino: approfondimenti

**Autore: Samuele Crivellaro – Versione 02/02/2017**

Quest'opera è soggetta alla licenza Creative Commons Attribuzione - Non commerciale - Non opere derivate reperibile qui.

## Sommario

1. Il processo di build ..... 1
2. Il Bootloader ..... 4
3. Le direttive al preprocessore ..... 5
4. Cosa sono i file .cpp e i file .h? ..... 5
5. Creare una libreria ..... 6
   5.1 La libreria Blink ..... 6
   5.2 La libreria BlinkOb in linguaggio ad oggetti ..... 7
Esercizi ..... 10
Bibliografia essenziale ..... 10

## 1. Il processo di build

Nel presente paragrafo analizziamo la struttura degli sketch Arduino e il processo che porta dai file .ino al file .hex che viene caricato sul microcontrollore AVR.

Il processo può essere rappresentato in via semplificata come in Figura 1. Quando si verifica/compila un file .ino vengono eseguiti i passaggi indicati:

1. **Trasformazione:** il file .ino viene integrato con alcune righe; si ottiene così un file con lo stesso nome, avente però estensione .cpp
2. **Preprocessing:** il file .cpp ottenuto viene elaborato dal preprocessore secondo le direttive riportate nello sketch, come per esempio `#define` oppure `#include` (vedi paragrafo 3)
3. **Compilazione:** il file temporaneo ottenuto viene compilato ottenendo un file oggetto .o. Allo stesso modo vengono compilate anche i file delle librerie utilizzate; queste possono essere contenute nella stessa cartella del file .ino oppure nelle cartelle standard per le librerie Arduino.

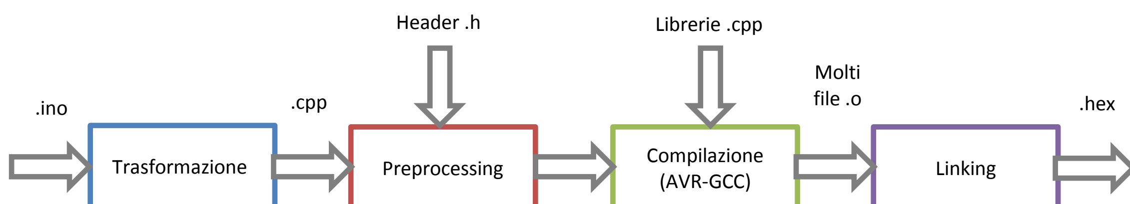


**Figura 1.** Rappresentazione semplificata del processo di build

Il compilatore usato da Arduino è AVR-GCC. GCC sta per GNU[1] Compiler Collection, ed è una collezione di compilatori (per C, C++, Java…) gratuiti. La versione AVR-GCC permette di compilare il codice C/C++ generando codice binario adatto ad essere caricato sui microcontrollori Atmel della famiglia AVR.

4. **Linking:** tutti i file oggetto generati dal compilatore sono uniti per ottenere alla fine un file .hex contenente il codice binario che deve essere caricato sul microcontrollore[2]

Analizziamo più nel dettaglio il processo appena descritto a partire da un semplice sketch che chiamiamo *blink.ino*:

**Listato 1.** blink.ino

```
int ledPin = 13;

void setup(){
  pinMode(ledPin, OUTPUT);
}

void loop(){
  blink(ledPin);
}

void blink(int ledPin){
  digitalWrite(ledPin, HIGH);
  delay(1000);
  digitalWrite(ledPin, LOW);
  delay(1000);
}
```

Per andare ad analizzare il file prodotti dal processo di verifica/compilazione è conveniente attivare le opzioni di compilazione dell'output verboso durante le fasi di compilazione e caricamento. Tali opzioni possono essere selezionate da *File → Preferenze*.

Avviato il processo di verifica è possibile, al suo termine, leggere l'output generato per individuare la cartella in cui sono salvati i file generati. Tra questi, in particolare, troveremo il file blink.cpp generato dal processo di *trasformazione*; in Figura 2 lo troviamo confrontato col file .ino di partenza.

In particolare possiamo fare alcune osservazioni:

- La riga `#include "Arduino.h"` viene aggiunta sempre; tale direttiva avvisa il preprocessore di aggiungere il file *Arduino.h* al file *blink.cpp* prima della compilazione. *Arduino.h* contiene tutti i prototipi delle funzioni fondamentali usate in Arduino (p.es. `digitalWrite()`, `digitalRead()`…).
- Subito dopo sono aggiunti tutti i prototipi delle funzioni che compaiono in *blink.ino*,
- ed infine il codice in esso presente.
- Le righe `#line 1 "blink.ino"` e `#line 1` servono solamente per definire il numero della riga successiva che compare nel file ed eventualmente il nome del file da cui provengono le righe successive. Sono solo delle indicazioni che il compilatore userà nel momento in cui dovesse segnalare degli errori.

---

[1] Il progetto GNU è un progetto collaborativo lanciato il 27 settembre 1983 da Richard Stallman il cui scopo ultimo è la creazione di un sistema operativo composto esclusivamente da software libero chiamato GNU. Il nome GNU è l'acronimo ricorsivo di "GNU's Not Unix". I software prodotti sono sviluppati esclusivamente grazie a una comunità di programmatori che mettono regolarmente in condivisione fra di loro tutte le modifiche al codice sorgente effettuate. Fulcro di tutta l'attività del Progetto GNU è la licenza chiamata GNU General Public License (GNU GPL), che sancisce e protegge le libertà fondamentali che, secondo Stallman, permettono l'uso e lo sviluppo collettivo e naturale del software. (da https://it.wikipedia.org/wiki/Progetto_GNU)

[2] In realtà vengono generati più file in sequenza; il file .hex è l'ultimo della sequenza

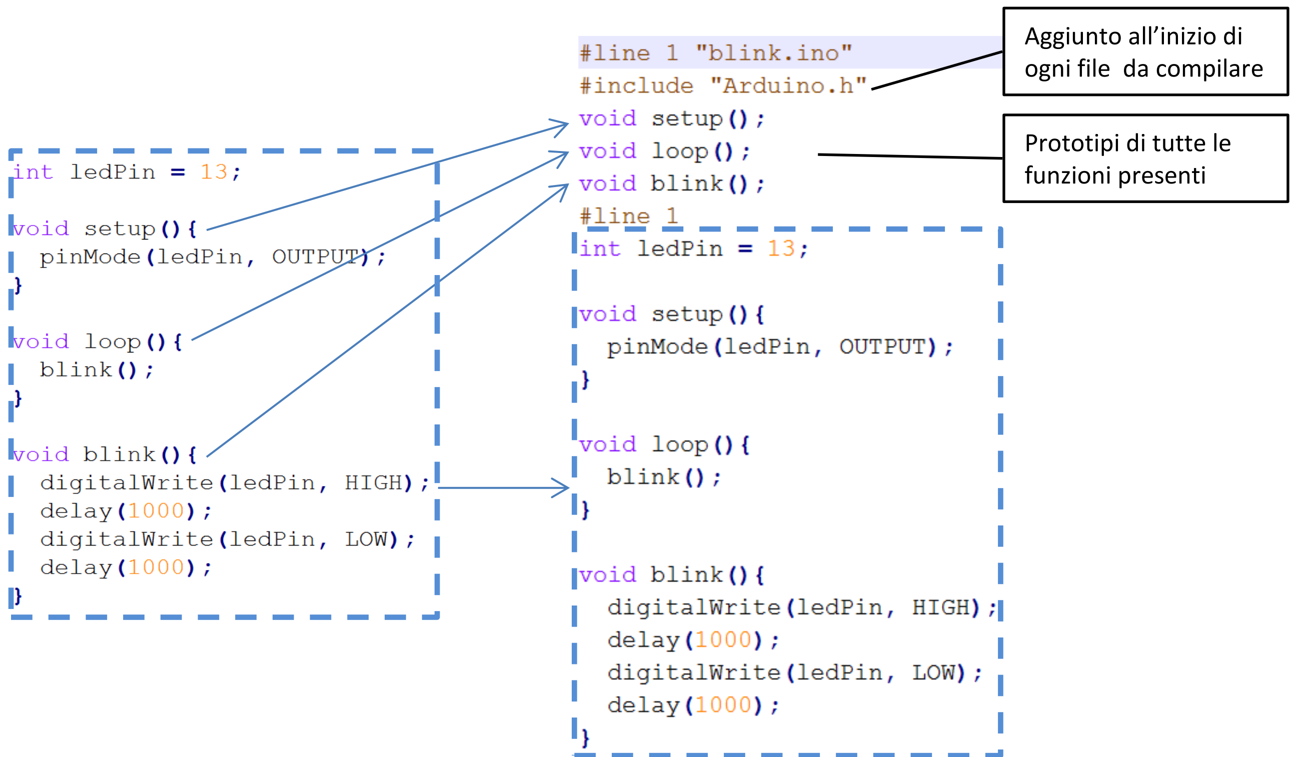


**Figura 2.** Illustrazione del passaggio dal file .ino al file .cpp

Nella stessa cartella in cui è presente *blink.cpp* si possono trovare anche il file oggetto *blink.cpp.o* (generato dalla compilazione di *blink.cpp*). Possiamo notare che sono presenti anche altri file oggetto, generati dalla compilazione delle altre librerie usate nello sketch. Infine si può trovare un unico file .hex: *blink.cpp.hex* , che contiene il codice binario da caricare sul microcontrollore e che è ottenuto dal processo di *link* di tutti i file oggetto.

Quando si ha a che fare con progetti più complessi, può essere utile dividere lo sketch tra più file .ino. Se questi sono inseriti in un'unica cartella, essi saranno comunque intesi come facenti parte di un unico progetto. In questo caso il flusso di build precedentemente descritto subisce una leggera modifica: nella fase di *trasformazione* viene generato un unico file nato dalla fusione di tutti i file .ino. Tale file avrà per nome il nome del file principale del progetto, ovvero del file .ino avente lo stesso nome della cartella in cui è contenuto il progetto.

A titolo di esempio è interessante scrivere lo sketch precedente dividendolo in due file, principale.ino:

**Listato 2.** principale.ino

```
int ledPin = 13;

void setup(){
  pinMode(ledPin, OUTPUT);
}

void loop(){
  blink(ledPin);
}
```

e blink.ino:

**Listato 3.** blink.ino

```
void blink(int ledPin){
```

```
  digitalWrite(ledPin, HIGH);
  delay(1000);
  digitalWrite(ledPin, LOW);
  delay(1000);
}
```

inserendoli nella cartella “principale”.

Procedendo col processo di verifica/compilazione si otterrà il seguente file principale.cpp:

**Listato 4.** principale.cpp

```
#line 1 "lampeggia.ino"
#include "Arduino.h"
void setup();
void loop();
void blink(int ledPin);
#line 1
int ledPin = 13;

void setup(){
  pinMode(ledPin, OUTPUT);
}

void loop(){
  blink(ledPin);
}
#line 1 "blink.ino"
void blink(int ledPin){
  digitalWrite(ledPin, HIGH);
  delay(1000);
  digitalWrite(ledPin, LOW);
  delay(1000);
}
```

Il file .hex generato al termine dell’operazione di build può essere finalmente caricato sul microcontrollore. Per fare questo viene usato un software dedicato chiamato **AVRdude**; anche qualora si usi direttamente l’IDE di Arduino, questo si appoggia comunque su AVRdude. Esistono sostanzialmente due modi per portare a termine l’operazione:

1. Caricare il file .hex su un microcontrollore su cui sia stato in precedenza caricato un bootloader (vedi paragrafo 2). In questo caso la programmazione avviene usando semplicemente un cavo USB; sarà il bootloader ad occuparsi di caricare il programma nella memoria flash. Questa operazione può essere fatta in modo automatico dall’IDE di Arduino, oppure “manualmente” usando AVRdude a riga di comando.
2. In alternativa è possibile caricare il file .hex su un microcontrollore “vergine”, ovvero senza bootloader. In questo caso sarà necessario usare un programmatore ISP (In-System Programming); eventualmente è possibile configurare anche una scheda Arduino affinché funzioni da programmatore.

## 2. Il Bootloader

I microcontrollori AVR della Atmel hanno grandi qualità, ma possono essere un po’ difficoltosi da programmare, soprattutto per il principiante. Chi ha creato la piattaforma Arduino ha in buona parte eliminato questi problemi caricando sul microcontrollore Atmega328 un *bootloader*. Grazie ad esso è possibile programmare la scheda direttamente attraverso una porta seriale, e quindi per programmare Arduino sarà sufficiente un semplicissimo cavo USB.

Il bootloader è un file .hex che viene avviato quando viene accesa la scheda. In qualche maniera è simile al BIOS di un computer ed assolve fondamentalmente a due compiti:

1. Controlla se un PC sta cercando di programmarlo; in tale caso carica il programma in arrivo nella memoria flash. Questo è il motivo per cui, quando si cerca di caricare uno sketch su Arduino, questo viene resettato: in questo modo, al riavvio, il bootloader andrà a "raccogliere" il programma in arrivo dal PC.
2. Qualora, invece, il computer non stia cercando di caricare uno sketch, il bootloader dice alla CPU di avviare il programma già contenuto nella memoria flash.

## 3. Le direttive al preprocessore

Le direttive al preprocessore sono delle indicazioni che vengono inserite all'interno dei file da compilare per ottenere determinati comportamenti in fase di *preprocessing*. Ciascuna direttiva è facile da individuare, poiché è sempre preceduta dal simbolo del cancelletto #.

Le direttive che si incontrano più spesso sono due:

- Direttiva `#define identificatore stringa_da_sostituire`
  Per esempio, se si scrive #define PI 3.14, prima del processo di compilazione verrà sostituita la stringa "3.14" in ogni punto del programma nel quale compare la stringa "PI"
- Direttiva `#include <file_header.h>`
  Viene incluso un file *header* (estensione .h). Si possono includere anche file .c o .cpp, ma solitamente è preferibile includere solamente file .h così da rendere più semplice la manutenzione del software.

## 4. Cosa sono i file .cpp e i file .h?

Andando a curiosare nella cartella contenente una libreria standard, si trovano quattro almeno cose; nella libreria Wire, per esempio, si trova

- Una cartella *examples* contenente alcuni sketch di esempio di uso della libreria
- Un file *keywords.txt* contenente le parole-chiave che si desidera siano evidenziate nell'IDE di Arduino
- Un file *Wire.cpp* contenente la libreria vera e propria
- Un file *Wire.h*, detto file *header*, che contiene quella che potremmo chiamare una "descrizione" della libreria. In particolare in esso troveremo:
  - Direttive al preprocessore
  - Definizione della struttura della libreria, con, in particolare, la dichiarazione delle classi e delle funzioni contentute nella libreria

A questo punto ci si può chiedere il motivo per cui, quando si vuole usare una libreria, non si includa direttamente il file .cpp. Nonostante, in effetti, questo sia possibile, questa non è generalmente una buona pratica, poiché rende più complessa la fase di manutenzione. Infatti, qualora la libreria usata nel nostro sketch dovesse essere modificata, se essa non è stata inclusa nello sketch (essendo stato incluso solamente il suo file header), sarà sufficiente ricompilare solo la libreria e non anche lo sketch. Il processo di link, invece, andrà ovviamente eseguito di nuovo.

Un'altra domanda che può sorgere a questo punto è la seguente: perché si rende necessario includere il file *header*? Non sarebbe sufficiente compilare lo sketch e le librerie che interessano e successivamente farne il link? No, perché, in questo caso il compilatore, nel momento in cui va a compilare lo sketch, troverebbe in esso presenti chiamate a funzioni che esso non conosce (visto che i processi di compilazione dello sketch e delle librerie sono indipendenti) e segnalerebbe perciò un errore. Inserire il file *header*, invece, significa inserire i *prototipi* delle funzioni che saranno utilizzate, e in questo modo il compilatore non potrà più "dire" che non "conosce" quelle funzioni.

# 5. Creare una libreria

Una libreria può essere pensata in prima battuta come un insieme di funzioni che facilitano il programmatore nella creazione di codice. In particolare le librerie portano con sé alcuni notevoli vantaggi:

- Permettono all'utente di usare funzionalità particolari di Arduino senza conoscerne i dettagli; per esempio si può usare la libreria standard EEPROM per accedere alla memoria ROM del microcontrollore, senza conoscere i dettagli che permettono effettivamente questo accesso.
- Permettono di condividere codice tra utenti diversi.
- Permettono di riutilizzare codice già creato per progetti sviluppati in precedenza.
- Rendono il codice maggiormente leggibile.

Sappiamo già dal capitolo 4 che ciascuna libreria deve contenere, come minimo, un file .cpp ed un file .h. Nel prossimo paragrafo creeremo una libreria di esempio e in quello successivo una libreria con le medesime funzionalità usando il paradigma di programmazione OOP (Object Oriented Programming).

## 5.1 La libreria *Blink*

Creiamo ora una libreria che permetta di far lampeggiare un LED connesso ad uno dei pin digitali di Arduino. Dal punto di vista della funzionalità, non aggiungeremo nulla di nuovo rispetto agli sketch nel capitolo 1.

Possiamo vedere il file .cpp della libreria come un'evoluzione dello sketch contenuto nel Listato 3; andremo semplicemente ad aggiungere due direttive al preprocessore all'inizio del file e le due funzioni *on()* ed *off()*:

**Listato 5.** Blink.cpp

```
#include <Arduino.h>
#include "Blink.h"

void blink(int ledPin){
  on(ledPin);
  delay(200);
  off(ledPin);
  delay(200);
}

void on(int ledPin){
  digitalWrite(ledPin, HIGH);
}

void off(int ledPin){
  digitalWrite(ledPin, LOW);
}
```

È bene commentare brevemente le prime due righe del file:

- `#include <Arduino.h>` serve ad includere il file header contenente tutte le funzioni e le costanti che si usano normalmente in Arduino,
- `#include "Blink.h"` include l'header della libreria (vedi Listato 6) e fa sì che il compilatore non restituisca errori quando, per esempio, all'interno del programma viene usata la funzione *on()* pur non essendo questa ancora stata definita. L'header della libreria va sempre incluso nel file .cpp.

Osservando con attenzione il programma sorge naturale una domanda: qual è la differenza tra la prima direttiva `#include`, dove il nome del file header è racchiuso tra parentesi triangolari, e la seconda, dove è invece racchiuso tra doppie virgolette? Semplificando, possiamo questo: se il file .h è racchiuso tra doppie virgolette, il compilatore inizierà a cercarlo dalla cartella in cui è contenuto il file che si sta compilando; se

invece è contenuto tra parentesi triangolari, la ricerca inizierà dalla cartella in cui, per default, sono contenute le librerie di sistema.

Come già sappiamo, accanto al file .cpp, dobbiamo creare anche un file header contenente i prototipi delle funzioni. Nel nostro caso il file sarà Blink.h:

**Listato 6.** Blink.h

```
#ifndef BLINK_H
#define BLINK_H

void blink(int ledPin);
void on(int ledPin);
void off(int ledPin);

#endif
```

Qui, oltre ai prototipi delle tre funzioni contenute in *Blink.cpp*, possiamo notare delle direttive al preprocessore che, fino ad ora, non abbiamo ancora incontrato. Esse formano quello che, in gergo, viene definito **include guard**; grazie ad esso si evita che lo stesso file header venga incluso più volte. La direttiva `#ifndef`, in sostanza, fa sì che tutto quello che si trova tra essa e la direttiva `#endif` venga "inviato" dal preprocessore al compilatore solo se non è già stato definito BLINK_H. In questo modo le righe comprese tra `#ifndef` e `#endif` vengono sicuramente incluse non più di una volta, visto che la loro inclusione comporta proprio la definizione di BLINK_H.

I due file *Blink.cpp* e *Blink.h* possono essere inseriti nella stessa cartella del file principale, oppure, per rendere la libreria "autonoma", in una cartella a parte, da inserire nella directory di sistema per le librerie. In tale cartella va inserita anche una cartella *Examples* e un file *keywords.txt*, che nel nostro caso può essere fatto così:

```
blink   KEYWORD2
on      KEYWORD2
off     KEYWORD2
```

In esso andremo ad indicare le parole chiave che vogliamo vengano evidenziate nell'editor dell'IDE di Arduino; dopo ciascuna parola chiave (nell'esempio "blink", "on" ed "off") è sufficiente indicare un codice a cui corrisponderà, nell'editor, un certo stile del carattere con cui essa verrà visualizzata. La parola chiave ed il codice corrispondente vanno separati da una tabulazione. I codici più frequentemente usati in Arduino sono: `KEYWORD2`, usato per metodi e funzioni, e `LITERAL1`, usato per le costanti definite nella libreria.

## 5.2 La libreria *BlinkOb* in linguaggio ad oggetti

In questo paragrafo andremo a creare una nuova libreria: *BlinkOb*. Essa avrà le medesime funzionalità della libreria *Blink* cerata nel paragrafo 5.1, ma sfrutterà la programmazione ad oggetti. Prima di adddentrarci nella creazione della libreria, introduciamo brevemente alcuni concetti del paradigma di programmazione orientata agli oggetti. L'obiettivo della presente trattazione, comunque, è semplicemente quello di mettere in grado l'utente di leggere ed interpretare librerie ad oggetti create da altri programmatori.

Quando si lavora usando il paradigma OOP bisogna pensare all'*oggetto* come al fulcro attorno al quale ruota tutto. Possiamo dire che:

- la programmazione orientata agli oggetti crea degli oggetti e poi "ci lavora sopra";
- un oggetto è definito a partire da un suo modello generico, chiamato classe;
- in una classe sono contenuti dei dati (nei campi) e dei metodi, ovvero delle funzioni che permettono all'utente di eseguire operarazioni sugli oggetti di quella classe.

Per capire di cosa stiamo parlando, la cosa più semplice è analizzare una classe esistente. Abbiamo scelto la classe Stepper, contenuta nella libreria standard Stepper di Arduino. Proviamo ad analizzare il file Stepper.h:

**Listato 7.** Stepper.h

```
// ensure this library description is only included once
#ifndef Stepper_h
#define Stepper_h

// library interface description
class Stepper {
  public:
    // constructors:
    Stepper(int number_of_steps, int motor_pin_1, int motor_pin_2);
    Stepper(int number_of_steps, int motor_pin_1, int motor_pin_2, int motor_pin_3, int
             motor_pin_4);

    // speed setter method:
    void setSpeed(long whatSpeed);

    // mover method:
    void step(int number_of_steps);

    int version(void);

  private:
    void stepMotor(int this_step);

    int direction;        // Direction of rotation
    int speed;          // Speed in RPMs
    unsigned long step_delay;    // delay between steps, in ms, based on speed
    int number_of_steps;      // total number of steps this motor can take
    int pin_count;        // whether you're driving the motor with 2 or 4 pins
    int step_number;        // which step the motor is on

    // motor pin numbers:
    int motor_pin_1;
    int motor_pin_2;
    int motor_pin_3;
    int motor_pin_4;

    long last_step_time;      // time stamp in ms of when the last step was taken
};

#endif
```

Analizziamo brevemente tale file; dopo la include guard, troviamo:

- La definizione della classe Stepper.
- La parola chiave public cui seguono:
  - la definizione di due metodi “**costruttori**” usati per creare gli oggetti di tipo Stepper (che hanno lo stesso nome della classe);
  - due metodi, setSpeed e step, utili, rispettivamente, a settare la velocità del motore e a farlo ruotare di un numero di passi prefissato, ed un metodo denominato version;
- La parola chiave private contenente il metodo stepMotor e una serie di variabili.

Ciò che è contenuto dopo “public” è visibile all’esterno, ovvero chi fa uso della classe può usare i metodi e le variabili contenute dopo tale parola chiave. Ciò che è contenuto dopo “private” non è visibile all’esterno, ovvero vi si può accedere solo dall’interno della classe.

In questo caso, quindi, gli unici metodi che possono essere usati dall’utente della classe sono i tre costruttori, setSpeed, step e version.

È normalmente una buona pratica evitare che variabili interne alla classe siano accessibili dall’esterno.

Rendiamo ora più chiari questi ragionamenti creando la libreria *BlinkOb*, ed iniziamo scrivendo per prima cosa il suo file header *BlinkOb.h*:

**Listato 8.** BlinkOb.h

```
#ifndef BLINKOB_H
#define BLINKOB_H

class BlinkOb{
  public:
    BlinkOb(int ledPin);
    void blink();
    void on();
    void off();
  private:
    int pin;
};

#endif
```

Analizziamo il codice. Le direttive al preprocessore fungono, come già visto, da include guard. Subito dopo incontriamo la definizione della classe BlinkOb, che contiene quattro metodi pubblici: BlinkOb (il costruttore), blink (per far fare un lampeggio al LED, on (per accendere il LED) ed off (per spegnerlo).

Il codice vero e proprio della classe sarà contenuto nel file BlinkOb.cpp:

**Listato 9.** BlinkOb.cpp

```
#include <Arduino.h>
#include "BlinkOb.h"

BlinkOb::BlinkOb(int ledPin){
  pinMode(ledPin, OUTPUT);
  pin = ledPin;
}

void BlinkOb::blink(){
  on();
  delay(1000);
  off();
  delay(1000);
}

void BlinkOb::on(){
  digitalWrite(pin, HIGH);
}

void BlinkOb::off(){
  digitalWrite(pin, LOW);
}
```

Infine lo sketch che fa lampeggiare il LED connesso al pin 13 usando la libreria BlinkOb sarà il seguente:

**Listato 10**

```
#include "BlinkOb.h"

BlinkOb led(13);

void setup(){
}

void loop(){
  led.blink();
}
```

Qui notiamo due cose importanti:

- alla terza riga abbiamo creato un oggetto di tipo BlinkOb che abbiamo chiamato led;
- alla penultima riga abbiamo utilizzato il metodo blink per far lampeggiare il led; notiamo la notazione usata, tipica del linguaggio ad oggetti: led.blink(), ovvero il nome dell'oggetto seguito dal punto e dal metodo che vogliamo usare.

Un'ultima osservazione: nel Listato 8 i metodi on() ed off() sono stati dichiarati pubblici; in effetti, visto l'obiettivo della libreria (far lampeggiare un LED), può essere superfluo permettere all'utente le semplici operazioni di accensione o spegnimento del LED. Per questo motivo è probabilmente più opportuno dichiarare come privati i due metodi on() e off().

A questo punto è possibile ampliare la libreria creata in molti modi, per esempio:

- aggiungendo un metodo che permetta di reimpostare l'intervallo di lampeggio.
- Aggiungendo un nuovo costruttore che permetta di impostare, oltre al numero del pin cui è connesso il LED, anche l'intervallo di lampeggio. È importante notare che è possibile creare più metodi con lo stesso nome ma parametri differenti; nel nostro esempio potremmo creare, oltre al costruttore `BlinkOb(int ledPin)`, anche un secondo costruttore: `BlinkOb(int ledPin, int intervallo)`. In questo caso, quando si crea un oggetto BlinkOb si può decidere se usare l'intervallo di tempo di default (inserendo un solo parametro) oppure di scegliere liberamente tale intervallo (inserendo due parametri). La possibilità di avere metodi multipli con lo stesso nome e numero differente di parametri, è tipica della programmazione orientata agli oggetti e viene denominata **polimorfismo**.

## Esercizi

1. Cosa accade nel file principale.cpp del paragrafo 1 se si aggiungono due righe vuote all'inizio del file *blink.ino*?
2. Nell'esempio del paragrafo 1 si sono creati due file .ino (*principale.ino* e *blink.ino*) inserendoli nella cartella "principale". Cosa accade se il nome della cartella viene cambiato in "blink"?
3. Creare una libreria (non ad oggetti) per la gestione di un LED permetta, in particolare, di:
    - Accendere e spegnere il LED
    - Farlo lampeggiare per un numero prefissato di volte alla velocità desiderata
    - Aumentare progressivamente la luminosità del LED da zero al suo valore massimo con la rapidità desiderata
    - Diminuire progressivamente la luminosità del LED dal suo valore massimo a zero con la rapidità desiderata
4. Modificare la libreria dell'esercizio 3, rendendola una libreria OOP.

## Bibliografia essenziale


**Sul processo di build**

http://www.cplusplus.com/forum/articles/10627/
https://www.arduino.cc/en/Hacking/BuildProcess
https://grahamwideman.wikispaces.com/Arduino+--+Project+structure+and+build+process
http://openhardwareplatform.blogspot.it/2011/03/inside-arduino-build-process.html

**Sulla creazione di una libreria**

http://playground.arduino.cc/Code/Library
https://www.arduino.cc/en/Hacking/LibraryTutorial
https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification
Margolis M., *Arduino – Progetti e soluzioni*, 2013[2], O'Reilly

https://www.arduino.cc/en/Reference/APIStyleGuide

**Sul Bootloader**

https://learn.sparkfun.com/tutorials/installing-an-arduino-bootloader