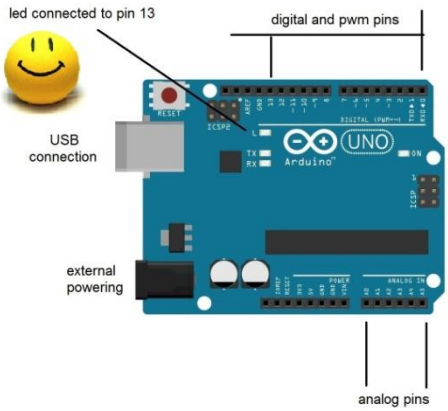
“Wiring”: the Arduino programming language

(translated by google translator)



Write a program is quite easy, but you must have a clear idea on the goal and on the way to achieve it.

To start using Arduino, just read the **introduction** of this handbook and then immediately move on to replicate the first projects of this collection.

If you read (and copy) programs, you will come to understand them. Then you can try to change them and finally to write new ones.

It's easier than seems. Just don't get discouraged and maybe read a couple of times some passage that seems difficult.

You need just a little volition and only at the beginning. And then it will be just a pleasant downhill road.

In this short guide are treated only some of the instructions provided by the “wiring” language.

They were considered only the most widespread instructions or rather, those more useful. With this subset you can deal all the use issues of sensors and actuators, and write also complex programs.

For questions or suggestions, mail to giocarduino@libero.it

The chapters of this guide

Introduction

Wiring is a programming language derived from C ++, used to write programs to run on the ATmega328 microcontroller that equips Arduino.

The microcontroller reads, activates or deactivates Arduino pins following the instructions contained in a ***sketch***, a program written in ***wiring*** and ***compiled*** (translated into machine code) through the **IDE** (Integrated Development Environment), a free tool operating on pc**.**
(The above sentence is probably the most complex sentence in this manual!)

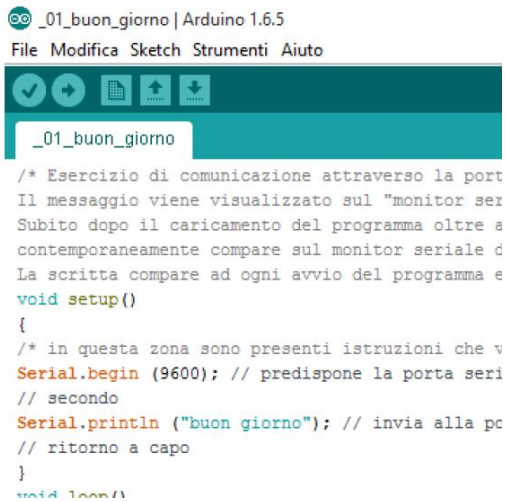
The IDE is therefore essential to operate on Arduino and can be downloaded from here:

https://www.arduino.cc/en/main/software

IDE allows you to use your PC to write program, to compile it and transfer it on Arduino, via the USB connection.

IDE has a window for writing program, some icons for verification, compiling, loading and saving programs and a series of a fairly intuitive pull-down menu. In "help menu"' is also present a full explanation of each programming element (variables, functions, instructions and their syntax).

Under a practical aspect, once IDE downloaded, installed and opened, you have to specialize it (only once, at first use), by selecting "Tools" and then "Arduino ....". Select finally, from pop up menu, the Arduino board that you are using (Arduino uno, Arduino mega, Arduino due, etc.).



La testata dell'IDE

Now, to compile, load and launch a program you must:

- Connect Arduino to PC via a USB cable (Arduino lights will illuminate).
- Copy (or type) a program in IDE editing area, already occupied by some predefined instructions (which shall be replaced by the new program).
- Press the Compile button (the right-pointing arrow, placed on the IDE command line, on top left)
- Wait the compilation end, noticeable from some white words, that appear on IDE bottom and from Arduino leds, that pulses for few moments.

After compilation, the program is automatically loaded on Arduino, that start immediately to work and do actions for which is programmed.

As first program to be run, we suggest to resort to the "good morning" found here. The program is simple and easy to interpret, does not require any particular component (only Arduino, a USB cable and a PC) and allows you to realize what is the compiler, how it works and also become familiar in serial monitor use, resident on IDE.

If during compilation, some red/orange message appears in the IDE lower area, means that something is wrong. In this case you have to interpret messages and adopt the necessary corrective actions.

Sometimes, especially at Arduino first use, an error message is proposed because the usb socket has not been properly addressed by the PC.

To remedy is normally sufficient selecting the correct port (com1, com2, com3 ...) from menu' Tools-> port, disconnect and reconnect the USB cable and then recompile program.

Since reading a programming guide is absolutely not pleasant, is appropriate, for newbies, only read this chapter (in practice IDE install and use) and then switch to practical experimentation, performing the first projects of this collection by copying and pasting, in IDE window, the programs proposed in each project sheet.

Having experienced (and tried to interpret) the first two or three projects, is advisable read the next four chapters, and access the next sections only to deepen the understanding of gradually used instructions and functions.

After experiencing the first ten projects and having obviously understood each coding line, you will not be probably more necessary to resort to this tome that, although synthetic and written in a flat shape, is quite arid.

What it is, how design and how write a program

A program is a list of elementary instructions that computer performs, one after another, uncritically. In writing a program you must adhere to certain rules of spelling, grammar and syntax, and you must arrange instructions, so that the computer reach, step by step, exactly to desired result.

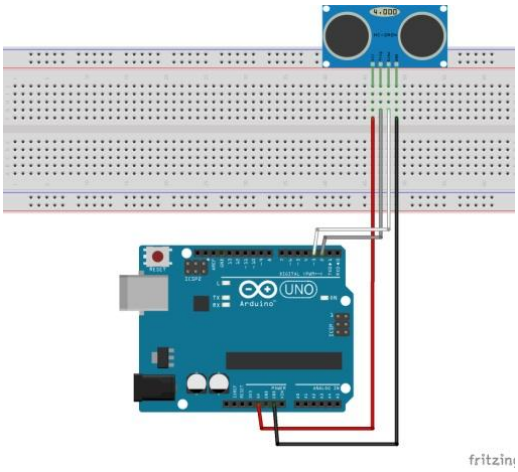
It seems difficult, but it is actually quite easy. Just think a program as a novel consisting of three chapters. In the first chapter characters are described with their characteristics, in the second chapter background is described and in the third and last chapter facts and actions involving the various characters are described. Similarly, three sections can be identified in each program, the first in which libraries, constants and variables are defined, i.e. the characters and allies who will then take part in the game, in the second part environment and communications are described and specialized, while in third part are described actions, that various characters (variables) aided by functions and routines (pieces of program that perform a function and which for convenience have been moved to a different area), execute to arrive at the desired result.

That means, before you write a program, is essential not only have a clear idea about the aim, but also what path you must follow to achieve it. Therefore it must be designed, maybe dividing it into parts such as:

- Data acquisition from sensors: what analyze, in what sequence and in what mode
- Decisions to be made, based on data from sensors
- Actuators enabling or disabling: what actuators, what operation (on / off) and in what sequence

Is also useful preventively draw connections between Arduino, sensors and actuators, define pins used by components and their use direction (input or output). An excellent free program for schematic drawing is available here:
http://fritzing.org/download/

On side is proposed the schematic (drawn by Fritzing), and in the next lines is reported the code of a program usable to measure a distance by using an ultrasound device.



In the sample program we have been used colors for various parts easy identification, and was commented each line, in order to clarify each instruction meaning and purpose.

Those who dont know programming language, may observe parts, read comments and groped to decipher some instruction.

Detailed information about program structure and on meaning of every single instruction, will be provided forward.

Key to colors used in this sample program:

Gray: **notes and comments,** that do not affect the program operation but makes it more easily understandable and manageable

Purple: **variables,** work areas in which are stored data used by program. Each variable is characterized by a name and a code which defines the type

Red: routines, also called "methods". Are instructions that are not part of the main program and that are performed only when they are called (or "launched") by another routine or by the main program. The routines use, makes the main program more easily interpretable

Green: initial instructions or "***setup*** instructions", execute only once, when the program starts

Blue: main program (or, rather, "***loop*** instructions"), executed and repeated continuously, until the power is taken off or the reset button is pressed

```
/*---------------------------------------------- example program ----------------------------------------------

This program uses a HC-SR04 device to measure distance from an obstacle. The program
launches an ultrasound beam and waits to receive a return signal. The time between launch and
signal return determines the obstacle distance. The time is converted to centimeters and then
highlighted on serial monitor (resident on PC). The schematic sees the "trigger" connected to
Arduino pin 2, '"echo" to pin 3, the negative to ground and the positive to the 5 volts supply pin. * /

float cm = 0;                 // a float type variables, called "cm", in which is stored the distance from
                              //  the obstacle, in centimeters. Its initial value is zero.

void misurazionedistanza (void)     //****starting point of “misurazionedistanza” routine ******

{                             // open bracket, which marks the beginning of the routine instructions
  digitalWrite(2, LOW);       // put OFF, on pin2, the ultrasound beam launch (if it was active)
  delayMicroseconds(2);       // waits 2 microseconds (to let off any echoes in ambient)
  digitalWrite(2, HIGH);      // put ON, on pin 2, the ultrasound beam launch
  delayMicroseconds(10); // waits 10 microseconds (time required by HC-SR04 to receive and
                              // process the return signal (echo))
  digitalWrite(2, LOW);       // put OFF, on pin2, the ultrasound beam launch
  cm = pulseIn(3, HIGH) / 58.8; // detects, on pin 3, the return signal, converts it into centimeters
                             // and stores it in "cm" variable. The 58.8 divider is a constant obtained by
                             // this formula: time for a meter = 1/(sound speed/2) . Was used the half
                             // speed sound because distance is twice: from the ultrasound generator
                             // to the obstacle and from the obstacle to the receiving sensor.
}                            // close bracket, which marks the end of the routine instructions

void comunicazionedistanza (void)  // **** starting point of “comunicazionedistanza” routine *******
{                             // open bracket, which marks the beginning of the routine instructions
  Serial.print(cm);           // show, on serial monitor, the value contained in "cm" variable
  Serial.println(" cm");      // show the word "cm" and places the cursor on the next line
}                             // close bracket, which marks the end of the routine instructions

void setup()                  // "setup section" starting point, and that means instructions executed
                              // only once, at the beginning of work
{                             //  open bracket, which marks the beginning of the setup instructions
  Serial.begin(9600);         // initialize the serial monitor
  pinMode(2, OUTPUT);  // declare the digital pin 2  (connected to “trigger”) as an output pin
  pinMode(3, INPUT);   // declare the digital pin 3  (connected to “echo”) as an input pin
}                             // close bracket, which marks the end of the setup instructions

void loop()                       // “Loop section” starting point. The “loop” is the main part of the
                                  // program, repeated continuously until you do not interrupt power
                                  // supply or press the reset button
{                                 // open bracket, which marks the beginning of the loop instructions
  misurazionedistanza ();         // launches the "distance measurement" routine
  comunicazionedistanza ();       // launches the "distance communication" routine
  delay (2000);                   // wait 2 seconds ( 2000 milliseconds) before loop restart
}                                 // close bracket, which marks the end of the loop instructions
```

The program observation, allows certain considerations

First, the part sequence (the initial notes, variables, routines, the setup and loop sections), whose location within the program must be respected, not only to obtain a correct operation, but also to facilitate the understanding and maintenance.

Then the notes, apparently exaggerated, widespread and sometimes redundant. In this example the notes have been deliberately expanded in an effort to make more easy reading the program, but also in normal programs is appropriate to specify the meaning of each single pass. Notes seem a pointless exercise, but a program, especially if complex, is subject to changes and notes are essential to avoid damage being edited.

The last consideration are the brackets. They are essential and mark the beginning and the end of: setup, loop, routines and condition subjected codes. They are not only indispensable, but contribute to make easier the program understanding.

In any case, indipendent from code interpretation that, with current knowledge can be difficult, you can, simply by reading comments and titles and observing brackets, realize how the program has been divided into parts and which functions are carried out on each part.

Program structure

A "wiring" program normally consists in three parts separated by two declarations (*void setup ()* and *void loop ()*). This is the standard structure:

```
/ * First part, for libraries declaration, variables, constants, and for routines (code that is executed only when is called by a specific instruction) * /
/* -----( Explanatory notes on program purpose and operation) -----*/
/*-----( Declaration of any libraries used by program)-----*/
/*-----( statement of used appliance, such as servo motors or liquid crystal display)---*/
/*-----( constants declaration )-----*/
/*-----( variables declaration )-----*/
/*-----( routines )-----*/

void setup()   /* second part or "setup", performed only at the starting program time * /
{  /* ----( start setup )------*/
   /*  ----( input/output pins declaration )----*/
   /*  ----( statements to be executed at the starting time )----*/
}/*   ----( end setup)---*/

void loop()   /* Third part or "loop", the main part of the program, which is performed and repeated until power shut off or until reset button pressed*/
{ /* ----( start loop )----*/
   /* ----( program statements )-----*/
} / *----( end loop )----*/
```

Some programmers inserts routines also at the end of the loop section or between the setup and loop section. This practice is permitted but reprehensible, since makes more difficult and dispersive the program interpretation.

Spelling, grammar and syntax

The programming language has rules that must be strictly followed:

- Each statement ends with a “**;**”
- The parentheses and square brackets delimits the operators of an instruction while braces delimits a set of instructions related to a condition, to a procedure or to a program part. If from an instruction depends the execution of other instructions, the "subordinated" instructions are normally enclosed in braces;
- At every open parenthesis must match a closing parenthesis. The absence of a closing parenthesis sometimes prevents the program compiling (and thus the execution) and in any case makes results unpredictable;
- The combination “**/***” indicates the start of a comment, which can extend over more rows and which must necessarily be closed by the combination “***/**”
- The combination “**//**” indicates a comment start, which extends to the end of the line;
- The indentation, not mandatory, is nonetheless useful to make more easy understand the program structure. In the IDE "tools" menu' there is the "automatic format" option, useful precisely to get the automatic code indentation;
- Variables and constants must be declared before their use (in terms of physical location into program) . For this reason is a good practice insert them at the program beginning, before "setup" and routines.

The keywords

The compiler recognizes some keywords and assigns them a specific meanings. The most importants are:

- ***HIGH*** and ***LOW*** Are keywords used to operate on Arduino pins and for a variable digital management. ***LOW*** means that the variable is associated to value 0 and/or on the related pin circulates a tension lower than 1,5 volt while ***HIGH*** imply that variable is associate to value 1 and/or on the related pin circulates a tension greater than 3,5 volt.

- ***INPUT*** and ***OUTPUT*** are keywords used to declare if a pin should be considered as input (a pin connected to a sensor) or as output (a pin connected to an actuator).

- ***true*** and ***false*** are keywords used to monitor the outcome of a condition.

The instructions

In writing a program, normally are used:

- **variables** and that means memory areas where data are stored;
- **instructions** and that means commands, in turn classified into:
    - **structure** and **control** instructions**;**
    - **math, boolean or computational** operators;
- **functions**, in turn classified into:
    - **input/output** functions;
    - **comunication** functions;
    - **time** functions;
    - **math** functions;
    - **random numbers generation** functions;

Variables

Are memory zones where the data, used by program, are stored.

Programmer assigns to each variable a name and a type definition code.

As already mentioned, variables must be defined before use. For that reason is appropriate define them in the initial area, before void setup () and before routines.

Note *(a bit 'tricky, to be read maybe when you know a little better the programming language): the variables defined within a section (in the setup, or within the loop brace, or even inside of a "for" loop or inside a routine) retain their meaning only within said cycle. Outside these cycles can be re-defined and take new and different meanings.*

This particularity is useful when you "assemble" a program by using codes and routines taken from other programs. In this case each routine, if accompanied by definition of variables used, can be easily incorporated into any program.

It is not especially useful if you write a program starting from the beginning and, especially in complex programs, is harbinger of errors and confusion.

How was told, you should always declare variables at the program beginning and use anywhere, but only for the purpose for which they were declared.

If stated that a variable is, for example, used as an index of an array, is should, for program intelligibility purpose, use it exclusively in that way, and not, for example, as a temporary storage for information that have nothing to do with the above array.

The most common types of variables are:

byte uses one byte and can contain an unsigned number, between 0 and 255.

int uses 2 bytes and can contain a number between -32768 to 32767.

unsigned int uses 2 bytes and can contain only positive numbers between 0 to 65535.

long uses 4 bytes and can contain a number between -2.147.483.648 to 2.147.483.647.

unsigned long uses 4 bytes and can contain a number between 0 to 4.294.967.295.

float uses 4 bytes and can store floating point numbers. Floating-point numbers can be as large as 3.4028235E+38 and as low as -3.4028235E+38.

char uses one byte. If you use it as a number it can contain a value that ranges from -128 to +127, but if you use it as text can contain any ASCII character. If you add two square brackets: ***char []*** becomes a ***string*** type variable in which can store a text. It uses one byte for each text character plus a NULL character that indicates the end of the text.

Example:

```
char saluto[] = "ciao";   // The string variable named "saluto" contains the word "ciao" and occupies
                          //   4 characters of text + a null character, 5 characters in total.
```

NOTE: in the sample the sign "=" was used to assign the value "ciao" to the "saluti" variable. The "=" sign, if not accompanied by another mathematical or conditional operator, is always interpreted as an assignment and therefore, in our case, assigns the value "ciao" to "saluti" variable. If it is to be used in a condition (to check, for example: if "saluti" is equal to 10), the "=" sign must be doubled, and so it must be used the combination "==" and then: if (saluti == 10).

Within square brackets, in a type char variable, you can insert a number to proactively define the length of the variable. This option is useful when you want to define an initially empty variable.

Example:

```
char area [10];      // A string type variable called "area" is empty, but makes available a memory
//                      zone which can be stored words of up to a maximum of 10 characters
```

array This is not a variable type, but the conventional name used to reference a matrix, a list of variables, accessible through an index.

An array is a type ***int*** or ***char*** variable, followed by a open square bracket, a numeric value (the number of elements) and a close square bracket. You use the type ***int*** when you want to define an array where each elements can contain an integer number from from -32768 to 32767, while using the type ***char*** if you want to define an array that contains characters.

It is also possible define in advance the values of each array item, by following the type ***int*** or ***char***, the square brackets with elements number, the equal sign and the values, separated by a comma and enclosed in brackets.

If, for example, we wish store four values, we can create an array like that:

```
int tab1[4]={10,25,50,100};      // the array called "tab1" is 5 bytes long (4 numbers + the ending
                                 //  null) and contains values accessible through an index. In this
                                 // example, the index can assume a value ranging from 0 to 3;
                                 // With zero leads to the first value (10) and with 3 leads to the
                                 // fourth value (100).
```

To refer values stored in an array, simply use the index associated to the array name. Example:

```
int val;          // defines a variable called “val” that will contain an integer
val = tab1 [2];   // store in "val" the value contained in the third element on the above array,
                  // and so 50 (remember: with index equal zero index leads the first element, with
                  // index equal to 1 leads the second element and with index equal 2 access the
                  // third  element ...)
```

#define nome value this is not a variable definition, but only value definition, which will be used by the compiler everywhere, in replacement of the word “***nome***”

Example:

```
#define pinled 5 //  during compilation, the compiler will substitute each "pinled" occurrence,
                 //  with the value: "5"
```

The structure instructions

Are two instructions or rather two statements, which delimit the program parts.

setup () Associated to the definition ***void*** indicates the start of the initialization area

loop () Associated to the definition ***void*** indicates the start of the loop area (the main code)

Among the structure instructions should also be included the keyword ***void*** which, although classified as a data type, flags each routine beginning.

In this last case its structure is:

void nome_della_routine ***(void)***
{….. routine instructions ….. ; */* A routine is executed only when the path followed by program meets the routine launch instruction (this is the format: **nome_della_routine ();**). At this point, the program executes the routine's instructions and, when finished, returns to the normal procedure, executing the instruction that follow the routine launch instruction. */*
}

To call (or better, to launch) a routine is enough write the routine name followed by an open and a closed parenthesis.

Example:
nome_della_routine ***();***

Note: at launch time, within brackets can be inserted some variables that routine uses or modifies. This option does not seem particularly useful in a program that is not born from assembling routines or functions, and therefore is ignored in this guide.

The control instructions

Are instructions that, upon a condition occurrence, launches a specific code execution.

if.. else…

Allows you to make decisions. The ***if*** statement must be followed by a condition enclosed by parenthesis and by one or more statements enclosed in braces. If the condition is true will run the instructions in braces while if the condition is false will perform the instructions (always in braces) immediately after the keyword ***else***. In both cases will then be carried out the instructions subsequent the ***else*** dependent instructions . You can use the ***if*** statement without the keyword ***else***. In the latter case, if the condition is true, will perform the statements enclosed in braces following the ***if***; if is false you will pass directly to subsequent instructions after the closing brace.

Structure:

```
If (..condition..)
  { ..code to execute if condition is true..; }
else
  { ..code to execute if condition is false..; }
…. code that will be executed in any case ….
```

Example:

```
if (val==1)                    // if the "val" variable contains “1”
{
   digitalWrite(3, HIGH);     /*puts the actuator which in the initialisation phase has been associated to pin 3 (for example a LED) in "HIGH" status (and that means' active)*/
 }

else
{
   digitalWrite (3, LOW);     /* but if "val" contains a value other than "1", places in "LOW" state (turns off) the actuator associated to pin 3 */
}
```

Note: As already mentioned, the combination of signs "==" is necessary to differentiate a condition from an assignment. More specifically, the ***val = 1*** expression is an assignment and that mean "enter 1 in the val variable", while the ***val == 1*** expression, preceded by a condition (such as an ***if***) is interpreted as a condition and then "if val is equal to 1"

The ***if*** dependent instruction can be used without braces; in this case, if condition is true, will be executed the immediately next instruction. If the condition is false, is skipped the next instruction and the second next will be executed.

Example

```
if (val==1)                    // if the "val" variable contains “1”
   digitalWrite(3, HIGH);      // puts the actuator which in the initialisation phase has been associated
                               // to pin 3 (for example a LED) in "HIGH" status (and that means
                               //  “active”)
Serial.print (“ciao”);         //  if "val" contains a value other than "1" or after the pin 3 activation (in
                              // any case) , the word “ciao” will be printed on serial monitor.
```

for…

the ***for*** statement repeats a series of instructions, until a condition is true.

Structure:

for (..initializes a variable..; ..condition..; ..modify variable..)
{ .. code to be executed and repeat until the condition is true.. }

Example:

```
for ( i=0; i <10;  i++ )
  {
  Serial.print (“Ciao”);
  Serial.println (“ for 10 times”);
  }
/ * i = 0 is an assignment: assigns the initial value zero to variable "i". "i" is an integer that must
have been preset in the variable declaration area. If not, you can specify it at the time and then
write “int i = 0” instead of “i = 0”;

i <10 is the condition: if "i" contains a value less than 10 are performed the following instructions,
enclosed in braces

i ++ is a modify statement that changes the variable "i". "++" is a computational operator and
means: "increases by 1 the contents of the variable"

 {  in brakets the code to execute and repeat until "i" is less than 10
 Serial.print (“Ciao”);  print “Ciao” on serial monitor
 Serial. Println (“ for 10 times”); Print “for 10 times” on serial monitor and then go to next line
  }  */
```

Note: The ***for*** dependent instruction can be used without braces; In this case, if the condition is true runs the statement immediately following the for closing parenthesis and then returns to the loop management instruction (in our example the ***i ++***) and at test.

Example

```
for (i=0; i<10; i++)
  Serial.println (“Ciao to you”);        // print on serial monitor “ciao to you” until
                                         //  “i” is less than 10
… next instruction…                      // instruction to be performed when the “ for” loop is ended
```

switch… case… break… default

Is a statement composed of elementary instructions. Is used to perform code depending on value contained in a variable whose name is between brackets, after the word ***switch***.

Structure:

switch (*..variable..* ***)***
{ case *XX:* *…code..* ***; break; case*** *YY:* ***..****code..* ***; break; default:*** *..code..* ***}***

Example:

```
switch(sensor)            / * Switch start instruction; the following instructions (until the closing brace) are subject to the value contained in the variable named "sensor" */

{
  case 38: digitalwrite(12, HIGH);break;   /* If the sensor variable contains 38, active the actuator connected to pin 12 (digitalWrite (12, HIGH);) and then exit from switch execution (break;) and jumps at the closing brace next instruction*/

  case 55: digitalwrite(13, HIGH);break; /* If the sensor variable contains 55, active the actuator connected to pin 13 (digitalWrite (13, HIGH);) and then exit from switch execution (break;) and jumps at the closing brace next instruction*/

  default: digitalwrite(12, LOW); digitalwrite(13, LOW); /* In any other case (if the sensor variable contains neither 38 nor 55) components connected to pins 12 and 13 are placed in LOW state*/

}

… next instruction…
```

while…

Performs a series of instructions enclosed in braces and repeats them until a condition is true.

Structure:

While (*..condition..* ***)***
{ *.. code to repeat untill the condition is true..* ***}***

Example:

```
while(sensore<500)       /* If the value contained in "sensore" is less than 500, executes
instructions contained in braces, otherwise continues starting from the closing brace next
instruction*/
{
  digitalWrite(13, HIGH);       // activated actuator connected to pin 13
  delay(100);                   // wait for 100 milliseconds
  digitalWrite(13, LOW);        // deactivated actuator connected to pin 13
  delay(100);                   // wait 100 milliseconds
  sensore=analogRead(1);   // put in "sensore" variable the value released by component
//                                connected to the analog pin 1 and returns to while statement
}
```

The example coding can be used to raise an alarm connected to pin 13 (a led or a buzzer) if the component connected to the analog pin 1 (for example, a temperature detector) returns a value smaller than 500. The alarm stops (and the program continues its path) only if the sensor provides a value equal to or greater than 500

do…while

Is an instruction identical to the "while", except that code is performed also before the condition is verified. It is used when you want to execute code at least once before the condition is evaluated.

Structure:

do
{** .. code to be repeated until the while condition is true.. **}
while (** ..condition.. **);

Example:

```
do
 {
        digitalWrite(13,HIGH); delay(100);
        digitalWrite(13,LOW); delay (100);
        valoresensore=analogRead(1);
 }
while (valoresensore < 500);
```

The code in example turns on and off a device (maybe a led) until the value provided by an analog sensor is less than 500. The difference, compared to the previous example, is that the flashing starts before checking value provided by sensor. Practically the flashing plays at least once, regardless of the value provided by the sensor

Break

This statement allows you to exit from a *for*, or *do ... while* or *while* loop and continue to run code that follow the cycle. It is also used in the switch statement, to interrupt the the conditions analysis.

Continue

Used in a *for .., do… while* or *while ...* cycle lets you interrupt execution of cycle internal code and return to the condition verification.

Example:

```
for(lum=0; lum<200; lum++)
{
        if((lum>120) && (lum<180)) continue;
        analogWrite(porta1, lum);
        delay(20);
}
```

in example we assumes that on output pin named porta1 is connected a led whose brightness varies progressively from 0 to 200. But the progression stops when the value brightness is between 120 to 180 (lum> 120 && lum <180).

Note: the combination "***&&***" is used to represent the boolean operator "and"

Maths operators

In program writing you can use the normal mathematical operators:

+ for sum

- for subtract

/ for divide

***** for multiply

= to assign results

There is also a strange operator: the "**%**" that, instead calculate a percentage (as you might expect), returns the remainder of a division.

Eample:
```
x = 19%7        // x contains 5, the rest of 19 divided by 7
```

Note: If the result of a division is assigned to a float type variable, the number of decimals on result is equal to the divider decimal number

Example:
```
float risultato;            // Define a float type variable, that will contain a division result
risultato = (11/3.00);      //By effect of the divisor decimal places, the result is limited to two decimal
//                            places, and then the “risultato” variable contains 3.67
```

Conditional operators

Are operators used in conditional instructions.

== equal to (double equal sign is essential to differentiate a condition from an assignment)

> greater than

< less than

!= different from

<= less or equal to

>= greater or equal to

Boolean operators

Are operators used to combine multiple conditions. For example, if we want to check if the "sensore" value is between 1 and 5 just write:

```
if(sensore>=1) && (sensore<=5)
   { . code to be executed if the value is greater than or equal to 1 and less than or equal to 5 .. }
else
   { … code to execute if the "sensore" value is less than 1 or greater than 5… }
```

There are three boolean operators

&& is the “and”,

|| is the “or”,

! is the “not”.

Note: Boolean operators, and mostly "or" and "not" combined together, are difficult to use. If improperly used, can lead to unpredictable results. In principle is always advisable to avoid using the "or" operator in conjunction with the "not" operator because the result is almost always different from what a hasty or unprepared user could expect.

Computational operators

are operators used to simplify some basic operations, such as increment or decrement a variable.

++ add 1 to variable placed before the computational operator

-- subract 1 from variable placed before the computational operator

Example:
val++ is like *val = val + 1* (increment *val* by 1)
val-- is like *val = val – 1* (decrement val by 1)

Should anyone use a value other than 1 you will have to use the following computational operators;

+=, -=, *=, /=

Eample: the following expressions are equivalent:

```
val=val+5;      // add 5 to val
val+=5;         // add 5 to val
```

Libraries and functions

functions are instructions or, better, macros through which you can interact with sensors and actuators, or perform some activities (typically calculations) that return a value.

In addition to the standard ones, provided by the Arduino IDE, there are a myriad of other functions managed by libraries, often written by users, that can be used to perform particular activities, especially in presence of complex to use sensors or actuators (as a stepper motor or an LCD display).

To use a library and its functions, you must declare it in the initial part of program (before setup and before the first routine).

To include a library in a program you must first insert it (if not already present) in the Arduino libraries file, and then recall it in program, by using

#include <library-name.h>

to be placed, as already stated, at the program beginning.

To insert a new library in the IDE libraries file, we need to go in the IDE, follow the path Sketch-> #include Library-> add library, select the folder or the compressed file (.zip) containing the library, and then press the button "open ".

After loading a new library you should close and reopen IDE, to ensure that the library and its new features are "visible" to program.

As already said there are libraries, specific for almost any kind of device. Such libraries are normally available on line, and are found simply by searching information (datasheets or even examples and usage notes) about the device you will use.

INPUT and OUTPUT functions

The programming language includes features for the Arduino pin management. Through these functions, is possible indicate the direction of use of a pin (INPUT or OUTPUT), enable or disable a digital pin, detect the signal from an analog pin or activate a digital pin in "PWM" mode (a mode that generates an analog output signal).

pinMode (pin, use-direction);

This istruction is used to configure a digital door; in "pin" you must enter the digital door number (from 0 to 13 on Arduino uno) that you want to configure and in "use-direction" the type of use (***INPUT*** or ***OUTPUT***).

Example:

```
pinMode(13,INPUT);      // set the digital pin  13 as an INPUT door
pinMode(12,OUTPUT);  // set the digital pin 12 as an OUTPUT door
/ * Note: In place of value 12 or 13 you can obviously uses a variable containing the pin number * /
```

digitalWrite(porta,valore);

active (***HIGH***) or deactive (***LOW***) an OUTPUT digital pin. The activation implies that the pin is fed with a 5 volts power while the deactivation means that no power circles through the pin (practically enables or disables the pin connected actuator)

Example:

```
digitalWrite(7,HIGH);       // active (ie puts in "HIGH"  state, in "on" state) the component connected
//                                to digital pin 7.
digitalWrite(led1, LOW);  // Disables (puts "LOW" state, in off) the component connected to the
//                                pin, whose number is stored in a variable named led1
```

variabile = digitalRead(pin-number);

It detects the state of an INPUT digital pin. Arduino returns value 1 (HIGH) in “variabile” if detects a voltage greater than 3 volts, places 0 (LOW) if detects a voltage lower than 1.5 volts and does not change the “variabile” value if it detects a voltage between 1, 5 and 3 volts.

Example:

```
int val = 0;                //define a type “int” variable called “val” with an initial value = 0
…….
…….
val=digitalRead(7);      // Detects voltage supplied by sensor connected to pin 7. If detects an
//                              higher than 3 volts tension, returns 1 (HIGH) in "val"; if it detects a  lower
//                              than 1.5 volts tension, returns 0 (LOW) while if it detects a between 1.5
//                              and 3 volts tension, leaves unchanged the value of "val"
```

variable = ***analogRead(***pin***);***

detects tension on an analog pin and returns in a variable, a number between 0 and 1023, proportional to the detected voltage. 0 corresponds to a 0 volts while 1023 corresponds to a 5 volts tension.

Example:

```
int val = 0;              // define a type “int” variable called “val” with an initial value = 0
…….
…….
val=analogRead(0);    // Detects tension on the analog pin 0 and returns, in “val” variable, a value
//                          between 0 and 1023, proportional to the detected tension
```

analogWrite(*porta****,****value****);***

With this statement, you can use a digital pin in a PWM way, and that means as an OUTPUT analog pin. The instruction is able to supply a voltage (between 0 and 5 volts) proportional to the number (between 0 and 255) present in the *value* variable.

Note: On Arduino Uno pin usable as analog OUTPUT are only pins: 11, 10, 9, 6, 5, e 3.

Example of use of input and output instructions (this is a complete program, for a led brightness management):

```
/ * hardware: a variable resistor, a led and a 220 ohm resistor. Schematic: connect the led positive
leg to a 220 ohm resistor, in turn connected to pin 9; connect the led negative leg to ground;
connect the center pin of a variable resistor to analog pin 3; connect the variable resistor extreme
pins: one to ground and the other to 5 volts power supply

Turning the variable resistor knob, vary the luminous intensity of the LED / *

int ledPin      = 9;   // defines a variable called "ledPin", which value is 9
int analogPin = 3;   // defines a variable called "lanalogPin", which value is 3
int val          = 0;   // defines a variable called "val", which value is 9

void setup()
{
pinMode(ledPin, OUTPUT);   // define digital pin 9 as an OUTPUT pin
}

void loop()
{
val = analogRead(analogPin);   // Arduino detects tension supplied from the variable resistor
//                                    connected to the analog pin 3 (the variable “analogPin” contains
//                                    the value 3)  and put in "val" a value proportional to the sensed
//                                   voltage (a  value between 0 and 1023)

analogWrite(ledPin, val / 4);      // Send to “ledPin” (pin 9) a tension that varies from 0 to 5 volts, in
//                                    function of value contained in val / 4 (more exactly a
//                                    (5/255) * (val / 4) volts)
}
```

Serial communication functions

Arduino, through the Serial communication functions, can communicate with serial monitor (the pc monitor and keyboard) within IDE (the development environment installed on your PC) or with other devices connected via serial ports (serial ports are represented by the usb port and by pins 0 and 1 (TX ed RX)).

There are numerous serial functions (you can find the complete list in IDE “help menu” guide), and among these the most used are:

```
Serial.begin(velocita); // Instruction, normally placed in setup section, that prepares Arduino to
//                         send and receive data via the serial port. In the "velocita" (speed)
//                         normally insert 9600 (9600 bits per second) but you can also use other
//                         values, up to a maximum of 115200.

Serial.print (valore);     // Send to serial monitor the "valore" content and keeps
//                          the monitor cursor on the current line.

Serial.println (valore); // Send to serial monitor the "valore" content, followed by a carriage
//                          return, so as to expose the subsequent messages on a new line

val = Serial.available(); //  Put in "val" (an "int" variable) the incoming message length
//                           (message generally typed on the PC keyboard, connected via USB)

val = Serial.read();       // Reads and inserts in val (normally a "chr" type variable) the incoming
//                           character
```

As we have seen from the above instructions, the serial port operates both incoming and outgoing. Through this door you can export data and messages as well as receive data, used by Arduino program.

Time functions

Wiring includes some time management functions. The most interesting seem to be:

millis();

Provides the passed milliseconds since the program was started.

Example:

```
tempo=millis();    // Inserts into  "tempo" (a "long" type variable), the milliseconds elapsed since
//                    Arduino was powered or resetted
```

delay(pausa);

Pause the program for the milliseconds specified in "pausa".

Example:

```
delay(1000);      // stop program for one second (1000 milliseconds)
```

Math functions

The language includes mathematical functions some of which are here represented. The functions full list is available in the IDE "help" menu'.

*gamma = **min (**alfa**,**beta**);***

fits into "gamma" the smaller of values contained in "alfa" and "beta"

Example:
```
val=min(50,30);          // “val” contains 30
```

A similar reasoning applies to the following functions:

```
val = max(x,y);          // inserts in val the greater of values contained in x and y.

val = abs(x);            // inserts in val the x absolute value (removes the x sign)

val = constrain(x,a,b);  // x is a variable that contains a value; "a" is the minimum
//                          acceptable value and "b" is  the maximum. The function “constrain”:
//                          puts x in val if x has a value between a and b;
//                          puts a in val if x is less than a;
//                          puts b in val if x is greater than b.

val = pow(base,esponente); // Exponentiation: inserts in "val" the "base" elevated to
//                           "esponente". Warning: "val" must be a "double" type variable

val = sqrt(x);           // Calculates the square root of x. Warning, “val” must be a "double" type
//                          variable

val =map( x, daMinimo, daMassimo, aMinimo, aMassimo);   // Puts in "val" a value
//                          between aMinimo and aMassimo, calculated, keeping the
//                          proportion between the x value and daMinimo and daMassimo.
//                          In practice re parameterise  x  on a different scale and between
//                          aMinimo and aMassimo
```

Random numbers generation functions

Wiring is equipped with a random numbers generator:

val = ***random*** *(max)* *// Puts in val a random number between 0 and the value contained in max*
//

Since the generated numbers are actually pseudo random (are derived from a huge pre-defined sequence), to avoid repeating the same sequence each time you start the program, you should initialize the random number generator using the statement:

randomSeed(*seme***);** *// initialize with a "seme" (seed) number, the random number generator*

By entering into "***seme***" an always different number derived, for example, by a time function applied to human action (such as the time elapsed between the program start and a button pushed) or the value supplied by an analog unused pin, the random () function will return random numbers in more and different sequences.

Conclusions

Here it concludes this brief guidw about the Arduino programming language.

You can found at specialized bookstores some more exhaustive manuals on the subject.

Since the language used for writing sketches is called "Wiring", the research should be directed towards a "Wiring" manual.

"Wiring" is a C ++ "contaminated" by JAVA. So is even perhaps possible use a C ++ and / or Java manual (this is anyway a not advisable hypotheses, given the size of these two volumes and because "Wiring" is only a subset of these languages).

Probably the best way to investigate an instruction or function characteristics, is the IDE guide or an internet search.

Last, but not least, it should be specified that also "Wiring" is equipped with its own development environment (its IDE), that you can find and download here:

http://wiring.org.co/download/index.html

The "Wiring" IDE can be used just as well as the Arduino IDE, to write, compile and load programs.