Introduzione a

I LINGUAGGI DI PROGRAMMAZIONE

a cura di Francesco Galgani
www.galgani.it

Avvertenza:

Con la presente pubblicazione, mi limiterò a introdurre sinteticamente alcuni concetti fondamentali e di base per la programmazione, mediante esempi tratti dal C++. Per chi volesse approfondire l'argomento e cimentarsi nello studio di un linguaggio, può trovare nei seguenti siti alcune buone guide, dalle quali ho anche tratto parte degli esempi qui proposti:

Guide alla programmazione (C, Visual Basic, Guida di base, Java, C++, Delphi, ecc.)
http://programmazione.html.it

Programmare In... (C, C++, Java, Perl, Cobol, Pascal, Matlab, Fortran, Javascript, ecc.)
http://net.supereva.it/programmarein/

Corso su Alcuni Elementi di Base del C++
http://www.math.unipd.it/~sperduti/CORSO-C++/Corso%20C++.htm

Che cos'è un linguaggio di programmazione?

Un linguaggio di programmazione è uno mezzo per poter scrivere "programmi", allo scopo di controllare il computer, facendogli eseguire determinati compiti.

I linguaggi del computer, allo stesso modo delle lingue umane, hanno un loro "*lessico*" (*insieme di parole*) e una loro "sintassi" (*modo con cui le parole sono messe insieme*).

Quanti linguaggi esistono?

Così come le persone possono comunicare usando tante lingue diverse, anche i computer possono essere istruiti mediante tanti linguaggi diversi.

A parte qualche rara eccezione, tutti i linguaggi di programmazione si basano su strutture simili: imparare uno dei linguaggi significa avere gli strumenti giusti per impararli tutti.

Pieter Bruegel, *La Torre di Babele*, 1563

La Babele dei linguaggi...

ASSEMBLER, FORTRAN, ALGOL, COBOL, BASIC, PASCAL, C, C++, LISP, ADA, SMALLTALK, LOGO, JAVA, ASP, PHP, PERL, DELPHI, DBASE, MODULA, MATLAB, SQL, MUMPS, PYTHON, OBERON, OCTAVE, EIFFEL, PARADOX, ecc.

Quale linguaggio scegliere?

Ogni linguaggio è stato creato e ottimizzato per certi tipi di applicazioni. Anche se nei nostri esempi useremo il C++, i concetti basilari di programmazione sono perlopiù comuni a tutti i linguaggi.

In sintesi:

Applicazioni Matematiche: Fortran, Algor, Matlab, C++
Sistemi operativi: Assembler, C, C++
Programmi gestionali: Visual Basic, C, C++
Programmi multipiattaforma: Java
Programmi didattici: HTML, Pascal, C++
Giochi: C++, Assembler, Java, Flash
Sviluppo siti dinamici: Perl, Asp, Php

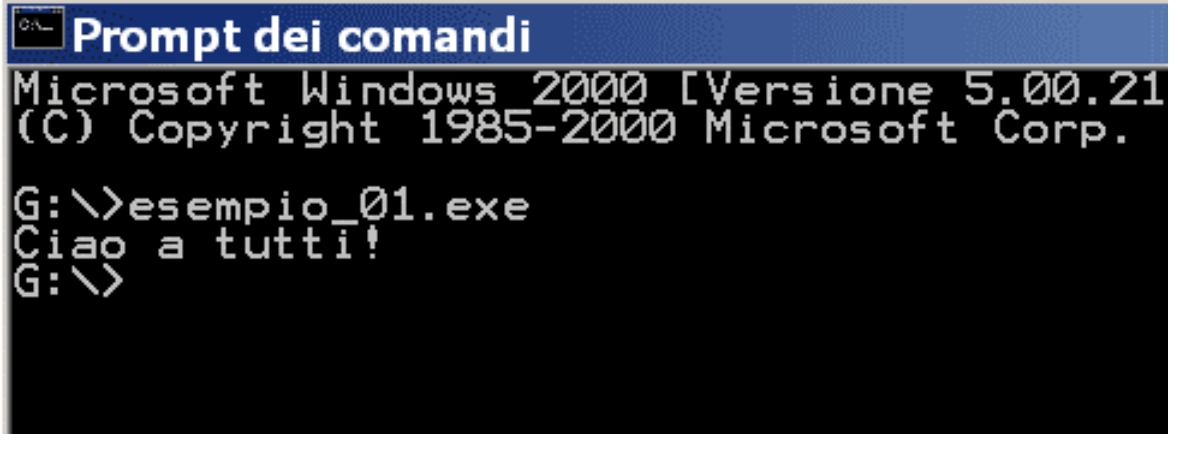
Qual è l'aspetto delle istruzioni di un programma?

Le istruzioni sono contenute in un file di testo, ad esempio:

```
// il mio primo programma in C++
#include <iostream.h>
void main() {
cout << "Ciao a tutti!";
}
```

CODICE SORGENTE DI UN PROGRAMMA

COMPILAZIONE



PROGRAMMA ESEGUIBILE

Come possiamo utilizzare le istruzioni di un certo linguaggio?

Ogni linguaggio, così come ogni lingua umana, è formato da "parole" che devono essere messe insieme secondo determinate regole (sintassi) al fine di ottenere un significato (semantica).

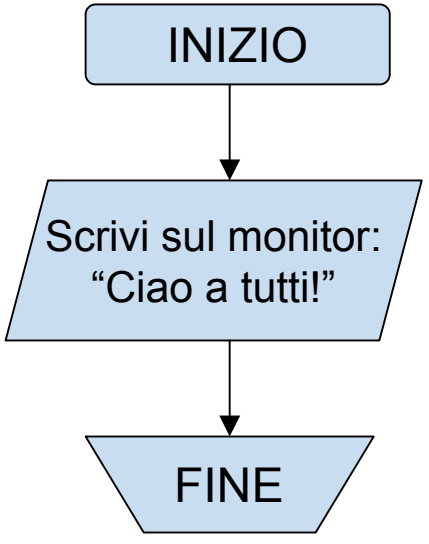
Disporre le parole rispettando una regola non è sufficiente per dare loro un significato:

regola:	**soggetto + verbo + complemento**	
es. 1:	Il mio cane corre sul prato	(sintassi corretta, significato valido)
es. 2:	Il tuo gatto salta sulla macchina	(sintassi corretta, significato valido)
es. 3:	Il fiume beve sul mio cane	(sintassi corretta ma priva di significato)
es. 4:	Il prato corre nella mia macchina	(sintassi corretta ma priva di significato)

Allo stesso modo, dobbiamo stare attenti a NON SCRIVERE programmi "formalmente corretti" ma senza senso!

Com'è possibile che la "Macchina-Computer" capisca il significato di ciò che scrivo?

Semplice diagramma di flusso



Innanzitutto **il computer è una macchina stupida**: può soltanto fare ciò che gli comandiamo, senza capire il significato di ciò che sta facendo (*esulano da questa affermazione e dagli argomenti qui trattati i problemi di intelligenza artificiale*).

L'abilità del programmatore è quella di risolvere determinati problemi con un numero finito di operazioni (**algoritmo**) e far fare al computer tali operazioni in maniera corretta, usando un linguaggio di programmazione.

Un algoritmo può essere rappresentato con **un diagramma di flusso**.

Come possiamo costruire un algoritmo?

I due strumenti che ci vengono in soccorso sono l'Algebra di Boole (utile per le operazioni sui dati) ed i diagrammi di flusso (utili per rappresentare in maniera ordinata i processi logici).

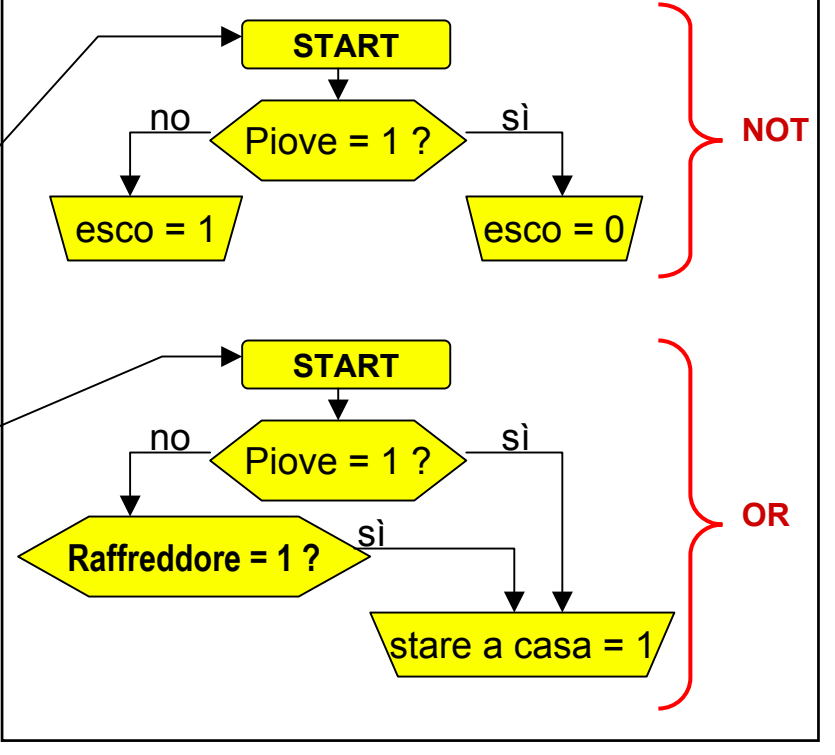
Algebra di Boole

rappresentazione delle informazioni:
bit: "0" o "1", "falso" o "vero"

operazioni fondamentali:
AND, OR, NOT

esempi:
- Se piove non esco, altrimenti esco (**NOT**)
- Se ho soldi e se è bel tempo, vado al mercato (**AND**)
- Se piove o se sono raffreddato, sto a casa (**OR**)

Diagrammi di flusso: esempio NOT e OR



Quali sono i concetti basilari per scrivere qualunque programma?

Per scrivere dei buoni programmi è quantomeno necessaria una mentalità logico-matematica. Affronteremo i principali elementi di base mediante esempi in C++.

In sintesi, ci occuperemo di:

- Linguaggi di basso e di alto livello
- Linguaggi interpretati e linguaggi compilati
- Struttura di un programma e le funzioni
- Parole chiave, operatori, separatori
- Costanti, variabili, tipi di dati
- Programmazione condizionale
- Programmazione iterativa

Linguaggi di basso e di alto livello

Un linguaggio di "**basso livello**" è più comprensibile (vicino) alla macchina che al programmatore, viceversa, uno di "**alto livello**" è più intuitivo e comprensibile per il programmatore.

Linguaggi di basso livello:
- lessico e sintassi elementari
- diversi per ogni processore
- più complessi da usare

Linguaggi di alto livello:
- descrizione dei problemi semplificata
- migliore approccio a problemi specifici
- livello di astrazione più alto

Esempio: sommare i numeri naturali da 1 a 20 e memorizzare il risultato nella variabile "somma"

Assembler ST6* (linguaggio di basso livello)

```
a       .def    0ffh
somma   .def    084h
main    ldi     wdog,0ffh
        clr     a
        ldi     somma,20
op      add     a,somma
        dec     somma
        jrz     exit
        jp      op
exit    ld      somma,a
```

C++ (linguaggio di alto livello)

```
void main() {
int somma=0;
for (int i=1; i<=20; i++) somma+=i; }
```

Ci sono casi particolari in cui la programmazione di basso livello viene ancora usata perché offre la possibilità di creare programmi con velocità di esecuzione e utilizzo di memoria migliori.

** "ST6" è un microprocessore, prodotto dalla SGS-THOMSON, in commercio dal 1988 e tuttora in uso, 17 anni dopo, per l'elettronica hobbistica e professionale. Se volete saperne di più, potete trovare una miriade di siti Internet dedicati al ST6.*

Linguaggi interpretati e linguaggi compilati

I linguaggi di programmazione, soprattutto quelli di alto livello, ci permettono di usare forme espressive "umanamente" comprensibili. Il computer, invece, "capisce" un suo linguaggio, detto "**linguaggio macchina**", composto da sequenze di cifre binarie:

10010111011100011101110010110001001111100101111010001 01

Tutti i linguaggi di programmazione hanno lo scopo di convertire le istruzioni in linguaggio macchina e, a seconda del metodo utilizzato, si dividono in due categorie: **compilati** (*usano un compilatore*) e **interpretati** (*usano un interprete*).

Linguaggi interpretati (il programma funzionerà con "codice sorgente + interprete"):
l'interprete legge una riga del codice sorgente → controlla lessico e sintassi → traduce → esegue

Linguaggi compilati (il programma funzionerà mediante un "file eseguibile", indipendente dal codice)
il compilatore legge tutto il codice → controlla lessico e sintassi → traduce → crea eseguibile

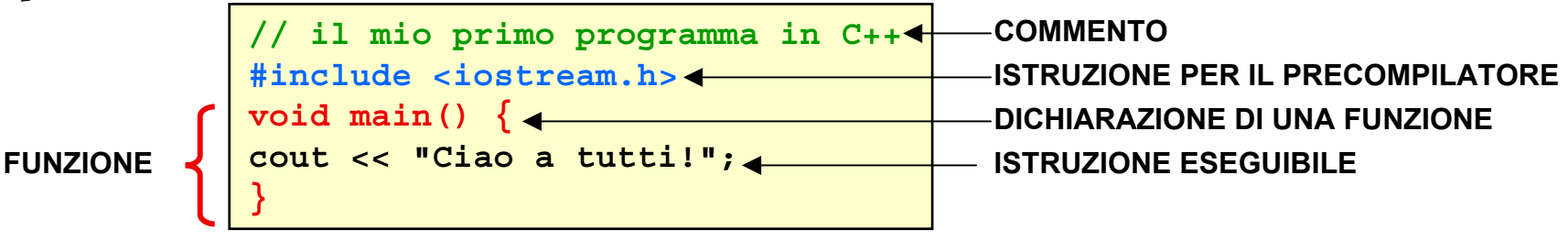
---

Esempio reale di compilazione di una riga di codice Assembler:

ldi somma,20 → lessico e sintassi OK → 0D rr nn (3 byte) →
→ 0D 84 14 → 00001101 10000100 00010100

Struttura di un programma e le funzioni

Non tutte le righe di un programma denotano un'azione: vi sono righe che contengono soltanto commenti, righe che contengono istruzioni per il precompilatore, righe che iniziano la dichiarazione di una funzione e infine righe che contengono istruzioni eseguibili, contenute nel corpo della funzione.

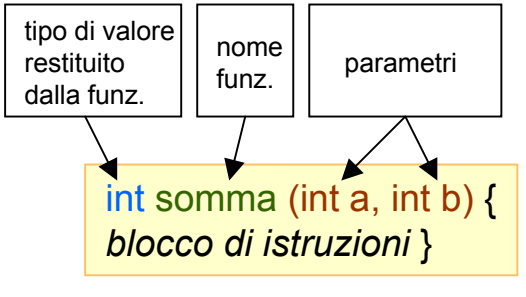


In matematica, una funzione ci permette di ricavare il valore di una variabile *dipendente* da quello di una *indipendente*; nella programmazione accade la stessa cosa: una funzione è un blocco di istruzioni che, accettando in input dei parametri (**costituiti da variabili**), "restituisce" (**return**) un valore in uscita.

Esempio: f(z)=a+b

```
// esempio di funzione
#include <iostream.h>
int somma (int a, int b) {
  int r;
  r=a+b;
  return r; }
```

```
int main () {
  int z;
  z = somma (5,3);
  cout << "Il risultato e' " << z;
  return 0; }
```



Parole chiave, operatori, separatori

Le "**parole chiave**" sono quei termini che, in base al linguaggio scelto, hanno funzioni precise: il compito del programmatore è quello di impararle ed usarle in maniera appropriata. Qui di seguito riportiamo, come esempio, le parole chiave del C++, molte delle quali vengono usate anche in altri linguaggi:

asm auto break case catch char class const continue default delete do double else enum extern float for friend goto if inline int long new operator private protected public register return short signed sizeof static struct switch template this throw try typedef union unsigned virtual void volatile while

Gli "**operatori**" sono caratteri speciali per controllare il flusso delle operazioni che dobbiamo eseguire (operazioni aritmetiche, operazioni logiche, trasferimento dati, ecc.).
I "**separatori**" sono invece simboli di interpunzione che permettono di chiudere un'istruzione o di raggruppare degli elementi.

```
operatori:  + -  * /  % ^  & |  ~ !  = < > += -= *= /= %= ^= &= |= << >>
            <<= >>= == != <= >= && || ++ -- , ->* -> .* :: () [] ?:

separatori: ( ) , ; : { }
```

Riferendoci al precedente esempio, osserviamo questa semplice funzione:

```
int somma (int a, int b) {
  int r;
  r=a+b;
  return r; }
```

5 parole chiave (evidenziate in rosso)
2 operatori (evidenziati in verde)
8 separatori (evidenziati in blu)

Costanti, variabili, tipi di dati (p. 1/2)

Pensiamo a quando salviamo un numero di telefono di un nostro amico sul cellulare: se vogliamo chiamare il nostro amico, basterà inserire il suo nome (**nome della variabile**) ed il cellulare comporrà automaticamente il numero di telefono (**valore della variabile**). La comodità risiede nel poter usare un nome per valori, che possono essere numeri o lettere, di grande entità o difficili da ricordare. Un altro vantaggio, da non sottovalutare, è la possibilità di usare il nome della variabile al posto del suo valore per eseguirvi sopra delle operazioni, con la possibilità, in seguito, di modificarne il valore come e quante volte vogliamo.

Supponiamo adesso di voler calcolare con molta precisione la lunghezza di una circonferenza; in questo caso, avremmo la necessità di far corrispondere a PI (**nome della costante**) un numero e uno soltanto (**valore della costante**):

3,14159265358979323846264338327950288419716939937510582097494459230781640628620899862803 4...

I nomi delle variabili e delle costanti, per quanto riguarda l'hardware del nostro computer, si riferiscono semplicemente a celle di memoria, ma è evidente che la memoria che utilizzeremo per memorizzare i nostri dati dovrà essere adeguata ai **tipi di dati** che vorremo memorizzare!

Nei linguaggi di basso livello, il programmatore è libero di utilizzare la memoria come meglio crede, stabilendo cosa scrivere o leggere e in quali indirizzi.
Nei linguaggi di alto livello, il programmatore non ha più la necessità di scegliersi personalmente gli indirizzi, poiché delega questo compito al compilatore.

Costanti, variabili, tipi di dati (p. 2/2)

Riferendoci ad un precedente esempio, osserviamo come avevamo dichiarato la variabile "somma":

Assembler ST6 (linguaggio di basso livello)

```
a        .def    0ffh
somma    .def    084h
main     ldi     wdog,0ffh
         ...
```

"somma .def 084h" significa che il compilatore deve associare la variabile "somma" ad 1 byte, con indirizzo 084 (espresso con un numero in base esadecimale). Il valore iniziale della variabile deve essere specificato nelle istruzioni successive.

C++ (linguaggio di alto livello)

```
void main() {
int somma=0;
for (int i=1; i<=20; i++) somma+=i; }
```

"int somma=0" significa che il compilatore sceglierà l'indirizzo a cui associare la variabile "somma", sapendo che il suo valore è di tipo "int", cioè corrisponde ad un numero intero rappresentato da 4 byte; inoltre il valore iniziale della variabile "somma" è "0".

In C++, i principali tipi di dato sono:

bool → 1 byte → valore booleani → "true" o "false", "1" o "0"
char → 1 byte → carattere alfanumerico o numero intero di 8 bit → da 0 a 255, da –128 a +127
short → 2 byte → intero di 16 bit →da 0 a 65535, da –32768 a +32767
int → 2 o 4 byte → equivale a "short" o "long" in base al s.o. (su Win9x,2000,NT equiv. a "long")
long → 4 byte → intero di 32 bit → da 0 a 4294967295, da – 2147483648 a +2147483647
float → 4 byte →numero in virgola mobile → 3.4e±38 (7 cifre decimali)
double → 8 byte →numero in virgola mobile con doppia precisione → 1.7e±308 (15 cifre decimali)

Programmazione condizionale: "*if*" ed "*else*"

Le istruzioni condizionali "if" ed "else" permettono di eseguire del codice a seconda che una condizione sia vera o falsa: se la condizione è vera, viene eseguito il codice subito dopo "if"; se è falsa, quello dopo "else". La struttura "if + else" può anche essere concatenata per verificare più condizioni, ad esempio:

```
if (x > 0)
    cout << "x e' positivo";
else if (x < 0)
    cout << "x e' negativo";
else
    cout << "x e' 0";
```

L'istruzione di scelta: "*switch*"

Lo scopo dell'istruzione "switch" è quello di confrontare il valore di una espressione con un certo numero di valori **costanti** (*le variabili non possono essere usate*) ed eseguire il blocco di istruzioni associato al valore dell'espressione. I seguenti frammenti di codice sono equivalenti:

```
switch (x) {
  case 1:
    cout << "x e' 1";
    break;
  case 2:
    cout << "x e' 2";
    break;
  default:
    cout << "valore di x ignoto"; }
```

```
if (x == 1) {
  cout << "x e' 1";
  }
else if (x == 2) {
  cout << "x e' 2";
  }
else {
  cout << "valore di x ignoto";
  }
```

Programmazione iterativa (cicli): "*while*", "*do*" e "*for*"

I "***cicli***" hanno lo scopo di ripetere una istruzione o gruppo di istruzioni un certo numero di volte oppure fino a che rimane vera una certa condizione.

ciclo while → ripete un blocco di istruzioni finché una condizione rimane vera
ciclo do-while → come sopra, con l'unica differenza che il blocco di istruzioni viene eseguito almeno una volta
ciclo for → ripete un blocco di istruzioni usando un contatore

Ad esempio, i seguenti programmi sono equivalenti:

ciclo while

```
// Visualizza da 1 a 50
#include <iostream.h>
void main() {
   short n=0;
   while (n<50) {
      n++;
      cout << n << " ";}
}
```

ciclo do-while

```
// Visualizza da 1 a 50
#include <iostream.h>
void main() {
   short n=0;
   do {
      n++;
      cout << n << " ";}
   while (n<50); }
```

ciclo for

```
// Visualizza da 1 a 50
#include <iostream.h>
void main()
{
for (short n=0; n<=50; n++) {
    cout << n << " ";
    }
}
```

Scomponiamo un numero in fattori primi...

Ho realizzato, con due linguaggi e due stili diversi, un programma per scomporre un numero in fattori primi. I sorgenti e gli eseguibili sono allegati insieme al file di Powerpoint.

Il programma "fattorizza.cpp" è scritto in C++ con un stile sintetico e di facile lettura. Il programma "fattori.bas" è invece scritto in Quick Basic 4.5, con un stile più dispersivo e più vicino al linguaggio macchina (potete notare, a tal proposito, il frequente utilizzo dell'istruzione "goto", ormai in disuso nei linguaggi di alto livello).

Francesco Galgani