GUIDA DI BASE

A cura di *Fabrizio Ciacchi*

Introduzione al mondo della programmazione. La guida base alla programmazione vi fornirà gli strumenti logici e pratici per poter affrontare lo studio dei linguaggi di programmazione con maggiore sicurezza. Dalla storia dei linguaggi all'uso delle funzioni.

TORNA SU

GUIDA DI BASE

LEZIONE 1: *Storia della programmazione*

Prima di iniziare a programmare bisogna conoscere e capire la storia e le funzionalità dei vari linguaggi di programmazione. Questo perché è importante conoscere quali eventi hanno contribuito a creare e sviluppare un linguaggio, rendendoci quindi più semplice scegliere quello più adatto alle nostre esigenze e capacità.

All'inizio, negli anni '40, l'unico metodo per programmare era il **linguaggio macchina**, in altre parole il lavoro del programmatore era quello di settare ogni singolo bit a 1 o 0 su enormi computer che occupavano stanze intere e pesavano decine di tonnellate; si capisce che questo tipo di procedimento, non solo era faticoso, ma era riservato ad una cerchia ristretta di persone.

Pochi anni dopo il 1950, con il progresso tecnologico e la riduzione delle dimensioni e dei costi dei calcolatori, nacquero due importanti linguaggi di programmazione, il **FORTRAN** (FORmula TRANslator), il cui utilizzo era ed è prettamente quello di svolgere in maniera automatica calcoli matematici e scientifici, e l'**ALGOL** (ALGOrithmic Language), altro linguaggio per applicazioni scientifiche sviluppato da Backus (l'inventore del FORTRAN) e da Naur, i quali oltretutto misero a punto un metodo per rappresentare le regole dei vari linguaggi di programmazione che stavano nascendo.

Nel 1960 venne presentato il **COBOL** (COmmon Business Oriented Language), ideato per applicazioni nei campi dell'amministrazione e del commercio, per l'organizzazione dei dati e la manipolazione dei file.

Nel 1964 fa la sua comparsa il **BASIC**, il linguaggio di programmazione per i principianti, che ha come caratteristica fondamentale quella di essere molto semplice e, infatti, diventa in pochi anni uno dei linguaggi più utilizzati al mondo.

Intorno al 1970, però, Niklus Wirth pensò bene di creare il **PASCAL** per andare incontro alle esigenze di apprendimento dei neo-programmatori, introducendo però la possibilità di creare programmi più leggeri e comprensibili di quelli sviluppati in basic. Si può affermare con certezza che Wirth ha centrato il suo obiettivo, considerando che ancora oggi il Pascal viene usato come linguaggio di apprendimento nelle scuole.

Pochi anni più tardi fa la sua comparsa il **C** (chiamato così perché il suo predecessore si chiamava B), che si distingueva dai suoi predecessori per il fatto di essere molto versatile nella rappresentazione dei dati. Il C, infatti, ha delle solide basi per quanto riguarda la strutturazione dei dati, però può apparire come un linguaggio assai povero vista la limitatezza degli strumenti a disposizione. Invece la sua forza risiede proprio in questi pochi strumenti che permettono di fare qualsiasi cosa, non a caso viene considerato "il linguaggio di più basso livello tra i linguaggi ad alto livello", per la sua potenza del tutto paragonabile al linguaggio macchina, mantenendo però sempre una buona facilità d'uso.

Ma la vera rivoluzione si è avuta nel 1983 quando Bjarne Stroustrup inventò il **C++** (o come era stato chiamato inizialmente "C con classi") che introduceva, sfruttando come base il C, la programmazione Orientata agli Oggetti (OO - Object Oriented) usando una nuova struttura, la classe.
Il concetto di classe è semplice, dividere l'interfaccia dal contenuto, ottenendo in questo modo tanti "moduli" interagibili tra loro attraverso le interfacce, permettendo così al programmatore di cambiare il contenuto di una classe (se sono stati trovati errori o solo per introdurre delle ottimizzazioni) senza per questo doversi preoccupare di controllare eventuale altro codice che richiami la classe. Come si può intuire tale linguaggio ha completamente stravolto il modo di programmare precedente ad esso che, sostanzialmente, si riduceva ad una programmazione procedurale che lasciava poco spazio al riutilizzo del codice; basti pensare all'utilizzo del *GOSUB* nel Basic che su migliaia di righe di codice creava naturalmente confusione, o al Pascal nel quale una variabile (e il suo tipo) deve essere dichiarata prima di essere usata.
Insomma il C++ è riuscito a creare un nuovo modo di programmare, o meglio di progettare un programma, rendendo il codice scritto più chiaro e soprattutto "riutilizzabile", ed è grazie a lui che oggi possiamo usare le finestrine colorate di Windows che ci piacciono tanto.

Altri linguaggi di programmazione da menzionare sono il **LISP** (1959), l'**ADA** (1970), lo **SMALLTALK** (1970) e il **LOGO**. Negli ultimi cinque anni, inoltre, si sono fatti strada linguaggi di programmazione come il **JAVA**, linguaggi di scripting come l'**ASP** e il **PHP**; i "vecchi" linguaggi, come il basic, il Pascal o il C++, sono diventati "VISUALI" e hanno aperto la strada ad una programmazione più intuitiva e veloce, ma di questo parleremo un'altra volta.

Lezione successiva

[Sommario]

TORNA SU

GUIDA DI BASE

LEZIONE 2: *Come e quale linguaggio scegliere*

Adesso è opportuno parlare delle varie applicazioni dei linguaggi di programmazione che, per loro natura, possono fare benissimo alcune cose e male altre. Per questo qui di seguito verranno menzionati i maggiori campi in cui viene applicata la programmazione e i linguaggi più adatti a tale campo:

- **Applicazioni Matematiche o di Ricerca**: per questo tipo di programmazione i linguaggi più adatti sono ancora il Fortran e l'Algol, non tanto perché non si possano creare linguaggi più versatili o potenti, ma solo perché dalla loro creazione è stato scritto ogni genere di programma matematico basato e quindi non è economicamente pensabile convertire tutto il codice già esistente per un altro linguaggio di programmazione. Ad oggi esistono però molti applicativi (il più famoso di tutti è il Matlab) usati in ambito universitario che hanno potenzialità maggiori in termini di prestazioni e di rappresentazione grafica, ma rimangono solamente ad uso e consumo dei pochi interessati che studiano materie scientifiche/ingegneristiche. Naturalmente per soluzioni o sviluppi proprietari viene molto usato anche il C++.

- **Sistemi Operativi**: La programmazione di un sistema operativo richiede il controllo di tutte le risorse del computer e ciò è possibile solamente scrivendo il motore del sistema operativo nel linguaggio più vicino al linguaggio macchina, ovvero l'Assembler il quale è estremamente potente, ma anche estremamente difficile da imparare e, soprattutto, da usare bene; le parti restanti del sistema operativo (ad esempio l'interfaccia con l'utente) generalmente vengono scritte in C o in C++.

- **Programmi Gestionali**: La programmazione di gestionali segue due filoni ben distinti; da una parte c'è chi scrive i programmi utilizzando Visual Basic, e che quindi girano solo su Windows; dall'altra parte, invece, c'è chi utilizza il C/C++ su sistemi operativi Unix (o come si chiama ora, OpenServer) o Linux.
  La scelta su quale linguaggio usare è totalmente libera, bisogna però considerare quale sistema operativo useranno maggiormente i nostri clienti.

- **Programmi per Windows**: Oggi come oggi per creare un programma Windows esistono molti linguaggi, primo tra tutti il Visual Basic, i più famosi poi sono il Visual C++, Delphi (l'evoluzione visuale del Pascal) e Java. Insomma per Windows la scelta è libera vista l'ampia fetta di mercato ricoperta da questo Sistema Operativo.

- **Programmi MultiPiattaforma**: Ad oggi l'unico vero linguaggio di programmazione MultiPiattaforma rimane il Java, che pur con le sue pecche, permette di creare programmi che possono girare indistintamente su Windows, su Macintosh, su Linux (con molti problemi), in remoto su pagine internet o, addirittura, su

elettrodomestici o cellulari. Il punto di forza di Java rimane la **J**ava **V**irtual **M**achine che, una volta installata su una periferica o un sistema operativo permette di eseguire i programmi Java senza problemi legati all'hardware o alla diversa impostazione dei sistemi operativi.
Bisogna però dire che si stanno affacciando sul mercato possibili realtà concorrenti come Kylix, il Delphi per Linux, o la piattaforma .NET, nata da e per Windows, ma che sta subendo il porting su Linux.

- **Programmi di Apprendimento**: Nei licei e nelle università per insegnare a programmare utilizzano principalmente tre linguaggi, l'HTML, il Pascal e il C++. Premesso che ciò varia molto da scuola a scuola e da università ad università, si può dire che questi linguaggi sono ottimi per imparare a programmare, soprattutto il Pascal, che come abbiamo detto la precedente lezione, è nato proprio per insegnare le basi della programmazione.

- **Creazione di Giochi**: Lo sviluppo di videogiochi avviene principalmente utilizzando il C++ e in alcune occasioni (per migliorare la velocità della grafica) anche l'Assembler. Bisogna dire però che possono essere creati semplici videogiochi anche utilizzando il Java ed il Flash, anche se quest'ultimo non può essere considerato un vero linguaggio di programmazione.

- **Sviluppo Siti Dinamici**: Per lo sviluppo di siti dinamici, invece, si possono utilizzare molti linguaggi, come il Perl, l'Asp o il Php. Recentemente stanno emergendo anche Python, Zope (che utilizza il Python) e Jsp, un linguaggio di scripting che utilizza le librerie di java.

Il succo della storia è che la scelta è vostra e và ponderata considerando "cosa" e "per chi" volete iniziare a programmare.

Lezione successiva

[Sommario]

TORNA SU

GUIDA DI BASE

LEZIONE 3: *Il linguaggio di Backus-Naur*

Prima di andare avanti permettetemi di affermare che uno degli aspetti più importanti quando si sviluppa un programma è quello di poter (e saper) scrivere correttamente un programma. Da quando sono nati i linguaggi di programmazione sono stati scritti libri e sono stati fatti corsi e seminari sul "come" scrivere programmi sensati. Quindi questa lezione e la prossima trattano (in maniera volutamente poco approfondita) il linguaggio di Backus-Naur e l'indentazione, il primo utile per scrivere programmi formalmente corretti ed il secondo importante per organizzare in maniera chiara il codice scritto.

Quando si esamina un linguaggio di programmazione occorre considerare due aspetti, la sintassi e la semantica: la prima rappresenta le regole che permettono di scrivere in maniera corretta le frasi del linguaggio, la seconda specifica il significato delle frasi, distinguendo tra quelle che hanno un significato e quelle che, invece, non ne hanno.
Per definire questi due aspetti occorre utilizzare un metalinguaggio, la cui definizione per la semantica risulta assai complessa, mentre per la sintassi il metalinguaggio è in genere costituito da un insieme di notazioni (non ambigue), simboliche o grafiche, che possono essere spiegate facilmente con il linguaggio naturale.
Nella fattispecie parleremo del linguaggio di Backus-Naur (BNF, Backus-Naur Form) utilizzato a livello internazionale nel campo della programmazione per definire la sintassi dei linguaggi. Quella che presentiamo qui è solamente una piccola infarinatura sul funzionamento e sulle ampie possibilità descrittive di questo linguaggio.

Per spiegare il corretto funzionamento prendiamo un piccolo sottoinsieme della lingua italiana scritta, in cui la sintassi del linguaggio è definita da alcune regole, ognuna delle quali descrive una struttura (o categoria sintattica). Proviamo a formalizzare la sintassi di una frase, attenendoci alla struttura seguente:

```
frase:
   soggetto  verbo  .
soggetto:
   articolo  nome
articolo:
  one of
     il  la
nome:
  one of
     cane  gatto  fiume  macchina  prato
verbo:
  one of
     corre  beve  salta
```

Queste regole stabiliscono che una frase deve essere obbligatoriamente formata da un articolo, un nome, un verbo e dal segno di interpunzione ".";

questo naturalmente vale per le regole che abbiamo definito, le quali possono essere estese anche in vari modi, vediamo sotto qualche esempio:

Elemento Opzionale - Un elemento opzionale viene generalmente contraddistinto dal pedice *opt* e indica appunto che tale elemento non è indispensabile per la costruzione di una "frase".

```
frase:
  soggetto  aggettivo opt  verbo  .
```

Elemento Ripetuto (o Ricorsivo) - A volte abbiamo bisogno che alcuni elementi si ripetano in maniera ricorsiva, questo per poter definire più elementi dello stesso tipo uno dopo l'altro.

```
frase:
  soggetto  verbo  complemento-list opt  .
complemento-list:
  complemento
  complemento  complemento-list
complemento:
  articolo  nome
  preposizione  nome
```

Tale lista rappresenta una lista generica, ma esiste anche la lista separata da una virgola, il cui nome termina con il suffisso *-clist*, dove *c* rappresenta il termine inglese *comma* (virgola). Ripetendo l'esempio precedente

```
frase:
  soggetto  verbo  complemento-clist opt  .
complemento-clist:
  complemento
  complemento  ,  complemento-clist
complemento:
  articolo  nome
  preposizione  nome
```

Elemento Incompleto - Quando abbiamo una lista di regole in forma incompleta, sarà usata l'espressione "...more..." per indicare che alcune alternative sono state omesse, secondo lo schema seguente:

```
element:
  alternative
  ... more ...
```

A questo punto possiamo proporre la nuova sintassi estesa, basandoci su ciò che abbiamo appena illustrato:

```
frase:
  soggetto  verbo  complemento-list opt  .
soggetto:
  articolo  aggettivo opt  nome
articolo:
 one of
   il  la
aggettivo:
 one of
   mio  tuo
nome:
 one of
   cane  gatto  fiume  macchina  prato
verbo:
 one of
   corre  beve  salta
```

```
complemento-list:
  complemento
  complemento  complemento-list
complemento:
  articolo  nome
  preposizione  nome
preposizione:
 one of
    nella  sul
```

Proviamo a costruire alcune frasi con questa sintassi:

- Il mio cane corre sul prato.
- Il tuo gatto salta sulla macchina.
- Il fiume beve sul mio cane.
- Il tuo prato corre nella macchina.

Come possiamo vedere tutte le frasi sono "sintatticamente" corrette, ma solo le prime due sono corrette dal punto di vista semantico. Questa è la cosa a cui il programmatore deve fare più attenzione, infatti un compilatore è in grado di trovare tutti gli errori sintattici, ma difficilmente individuerà una struttura programmativa senza senso, come ad esempio un ciclo che non finisce, sintatticamente corretto ma errore grave da evitare per il corretto funzionamento del programma e per una futura modifica del codice da parte di un'altra persona.
Quindi studiare la sintassi è indubbiamente molto utile e il linguaggio di Backus-Naur permette di comprendere le regole essenziali per scrivere un programma, ma il buonsenso e l'organizzazione logica di un programma sono cose che difficilmente si possono insegnare tramite un libro, ma si acquisiscono programmando giorno dopo giorno.

Questa lezione è liberamente tratta dal libro "Introduzione alla Programmazione ed elementi di strutture dati con il linguaggio C++" - A. Domenici, G.Frosini

Lezione successiva

[Sommario]

TORNA SU

GUIDA DI BASE

LEZIONE 4: *Impariamo ad Indentare*

Come abbiamo detto nella precedente lezione, imparare a scrivere programmi "sensati" è alquanto difficile e ci vuole tempo per padroneggiare le strutture più complesse. In questo caso, però, potrebbe venirci in aiuto la forma con la quale scriviamo i programmi, avvantaggiandoci per lo sviluppo e per eventuali modifiche successive.
Stiamo parlando della "maniera di scrivere" un programma, ma questa volta non dal punto di vista sintattico, bensì facendo attenzione a come quello che scriviamo venga presentato visivamente piacevole e, soprattutto, chiaro per chi legge il programma. Chi legge il programma infatti possiamo essere noi durante lo sviluppo dello stesso, ma potrebbero essere anche altre persone che tempo dopo vogliono modificare qualcosa; se la struttura è chiara si potranno, infatti, facilmente individuare errori sia sintattici che semantici.
Ed è soprattutto l'ultimo punto quello che a noi interessa, perché non esiste compilatore che possa dirci gli errori "logici" che stiamo facendo.

La tecnica che viene generalmente usata si chiama indentazione e consiste solamente nell'inserire spazi o tabulazioni (generalmente ignorati dal compilatore) per mettere subito in luce eventuali gerarchie dei cicli o delle funzioni. Ad esempio creando una semplice tabella in HTML possiamo scriverla in questo modo:

```
<TABLE> <TR> <TD>a</TD> <TD>b</TD> <TD>c</TD> </TR>
<TR> <TD> <TABLE> <TR> <TD>a1</TD> </TR> <TR>
<TD>a2</TD> </TR> </TABLE> </TD> <TD>b1</TD> <TD>c1</
TD> </TR> </TABLE>
```

ma, francamente, non si capisce esattamente come verrà presentato il testo nel documento HTML; per questo con l'inserimento di alcuni spazi si riesce a rendere più chiaro per noi e per gli altri quello che stiamo scrivendo,

```
<TABLE>
 <TR>
  <TD>a</TD>
  <TD>b</TD>
  <TD>c</TD>
 </TR>
 <TR>
  <TD>
  <TABLE>
   <TR>
    <TD>a1</TD>
   </TR>
   <TR>
    <TD>a2</TD>
   </TR>
  </TABLE>
  </TD>
  <TD>b1</TD>
  <TD>c1</TD>
```

```
 </TR>
</TABLE>
```

Si consiglia anche di "commentare" il codice scritto in maniera da rendere più chiare le operazioni logiche che stiamo svolgendo. Possiamo commentare un qualsiasi codice in svariati modi, qui sotto ne elenchiamo alcuni di esempio presi dai più comuni linguaggi di programmazione:

```
// o #
   Posto dopo un'istruzione, tutto ciòche compare sulla
stessa linea dopo questi simboli verràinterpretato come
commento.

/* */ o <!-- -->
   Ogni carattere compreso tra questi simboli
verràinterpretato come commento
```

Inoltre poiché qualsiasi monitor è limitato in larghezza sarebbe buona norma, quando si scrive un programma od anche un semplice file HTML, non superare, mentre si scrive, le 80/90 colonne, poiché oltre tale limite siamo costretti ad usare la scrollbar dell'editor per continuare a leggere il testo, perdendo così del tempo e rischiando di perdere il filo logico del documento.

Lezione successiva

[Sommario]

TORNA SU

GUIDA DI BASE

LEZIONE 5: *Introduzione alla Logica e Diagrammi di Flusso*

A questo punto dobbiamo spiegare meglio come riuscire a formalizzare in maniera logica i concetti che vogliamo trasformare in programma; questa operazione è forse la più importante e complessa, perché è su questa che poi si baserà il lavoro di scrittura del codice che andremo a mettere in atto.
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 nostri processi logici).

Apriamo una piccola parentesi sull'Algebra di Boole; questo tipo di logica non è correlata esclusivamente al mondo della programmazione, ma fa parte della vita di tutti i giorni. Pensiamo ad esempio a quando vogliamo uscire di casa, se piove non possiamo uscire; questo processo mentale, cioè il verificare se piove, può assumere, di fatto, due stati, o è vero (e quindi non usciamo) o è falso (e quindi usciamo). Si può capire che se questo tipo di logica risulta utile a noi come esseri umani, lo è ancora di più per un computer poiché il suo linguaggio è fatto solo di bit ed è possibile associare al vero il valore **1** ed al falso il valore **0**.

L'**Algebra di Boole** verrà qui solamente accennata e sostanzialmente consiste nel prendere come valori "vero" e "falso" (o 1 e 0); posti questi valori andiamo a vedere come operazioni logiche producano dei risultati nuovi e sensati.

AND - Congiunzione

falso AND falso	risultato falso
falso AND vero	risultato falso
vero AND falso	risultato falso
vero AND vero	risultato vero

Riusciamo facilmente a comprendere che l'AND restituisce un valore vero "se e solamente se gli altri due valori sono veri", questo vuol dire che anche in una successione di più operazioni AND basta che un valore sia falso ed anche il risultato lo sarà.

OR - Disgiunzione

falso OR falso	risultato falso
falso OR vero	risultato vero
vero OR falso	risultato vero
vero OR vero	risultato vero

L'OR invece restituisce un valore vero "se e solamente se almeno uno dei due valori risulta vero"; in poche parole anche in una successione di più operazioni OR basta che un valore sia vero ed anche il risultato lo sarà.

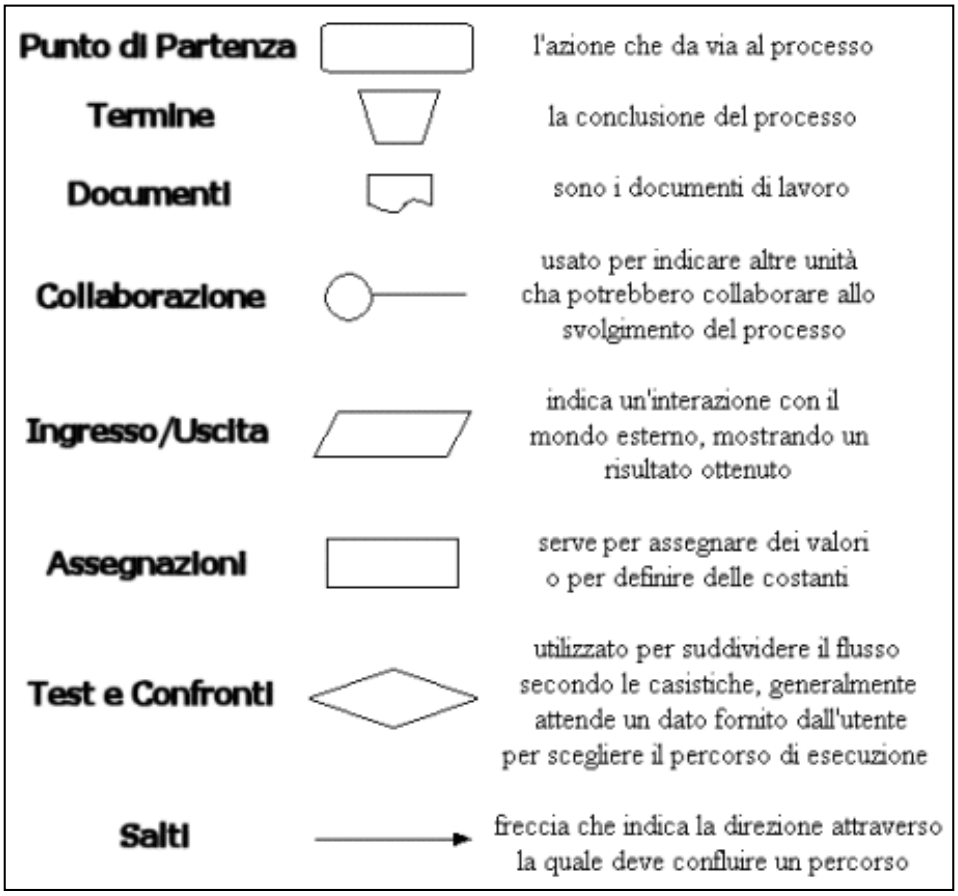
NOT - Negazione

NOT falso	risultato vero
NOT vero	risultato falso

Il NOT si riduce ad una semplice operazione di negazione del valore acquisito.

Si può evincere che grazie all'Algebra di Boole è possibile formalizzare i nostri pensieri riuscendo così a creare strutture complesse di rapido utilizzo, ma che non erano direttamente connesse ai dati iniziali. Basti pensare che i microprocessori stessi si basano sulla semplice logica dell'Algebra di Boole, con la quale è possibile formalizzare qualsiasi tipo di ingresso.

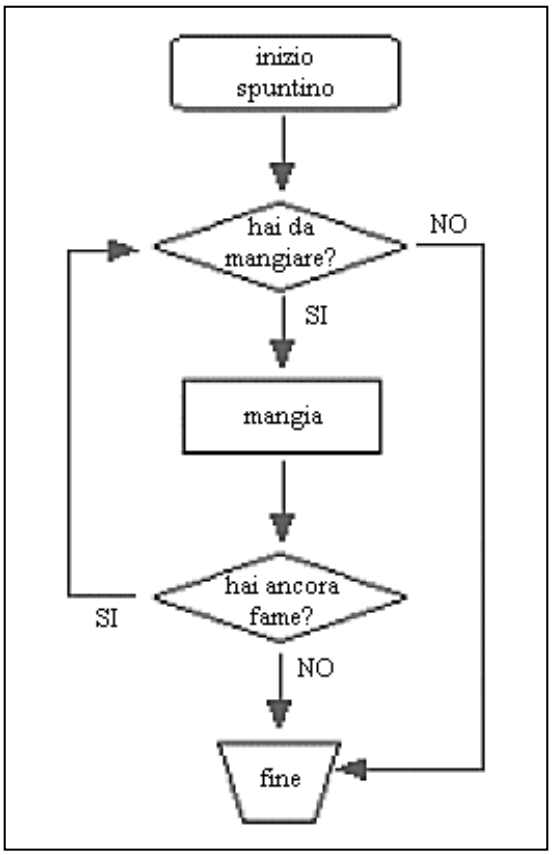
I **diagrammi di flusso** sono dei disegni che rappresentano graficamente un nostro ragionamento rappresentandolo come algoritmo, permettendo così, primo, una comprensione immediata del funzionamento del nostro percorso logico, secondo, un controllo accurato sulla funzionalità e la casistica del ragionamento.
Prima di procedere con un esempio pratico occorre definire i "simboli" utilizzati all'interno dei diagrammi di flusso per identificare i vari tipi di azione. Ci viene qui in soccorso la tabella sottostante che mostra i simboli ed il relativo significato per schematizzare il flusso del nostro ragionamento.



Si può facilmente capire che la schematizzazione di un algoritmo risulta quindi molto semplice, visto che ogni simbolo viene assegnato ad una fase del percorso logico che facciamo. Prendiamo in esame un semplice ragionamento che verifica se è possibile fare uno spuntino, tale programma viene spiegato come segue:

- Inizia lo spuntino
- Hai a disposizione qualcosa da mangiare?
- Mangia
- Hai ancora fame?
- Fine dello spuntino

Esposto in questo modo l'algoritmo risulta di difficile comprensione, ma rappresentandolo con un diagramma di flusso possiamo comprendere l'esatto funzionamento del ragionamento.



Naturalmente la teoria dei diagrammi di flusso risulta essere più complessa rispetto all'esempio sopraesposto, ma i dati forniti servono comunque per farsi un'idea di come uno strumento apparentemente semplice, possa essere usato per formalizzare anche concetti di grande entità e complessità.

Lezione successiva

[Sommario]

TORNA SU

GUIDA DI BASE

LEZIONE 6: *Parole Chiave ed Operatori*

Con questa lezione prendiamo in considerazione le parole ed i simboli utilizzati dal linguaggio di programmazione per eseguire le operazioni più disparate, come scrivere qualcosa a video od eseguire una semplice operazione matematica. Queste parole (e simboli) quindi non possono essere utilizzate per definire il nome di una variabile o di una funzione.
I termini utilizzati all'interno di un linguaggio si suddividono in "Parole chiave", "Espressioni letterali" e "Operatori e separatori" e ci accingiamo a spiegarli in dettaglio per permetterne una comprensione ed un uso più sapiente.

- **Parole chiave**
  Le parole chiave risultano molto utili perché sono le parole, o meglio i termini composti da caratteri alfanumerici, riservate al linguaggio di programmazione. Il creatore del linguaggio di programmazione stabilisce a priori quali termini riservare e quale sarà la loro funzione, il compito del programmatore è quello di impararle ed usarle in maniera appropriata. Fortunatamente l'uso improprio di tali termini viene generalmente rilevato durante la fase di compilazione di un programma.
  Qui di seguito vengono riportare, come esempio, le parole chiave del C++, molte delle quali vengono comunemente usate nei più disparati linguaggi di programmazione.

```
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
```

Si capisce come risulti quindi discutibile assegnare ad una funzione il nome "new" o ad una variabile il nome "int", in quanto diventerebbe a quel punto impossibile (per il compilatore) distinguere tra l'uso di quella parola come elemento fondamentale del linguaggio di programmazione o semplice nome da utilizzare per memorizzare dati.
Inoltre, tanto per la cronaca, tra i termini citati, ad esempio, IF viene usato anche in Basic, Pascal, PHP, ASP, Javascript, Java; questo per testimoniare che una volta apprese le parole chiave basta veramente poca accortezza per non usarle in maniera impropria.

- **Espressioni letterali**

Le espressioni letterali vengono usate per rappresentare dei valori costanti ed al di là dell'uso per caratteri, stringhe, interi o reali, risultano molto utili grazie alle "sequenze di escape".
Le sequenze di escape infatti permettono di rappresentare i caratteri di controllo come la tabulazione o il ritorno carrello, ma soprattutto permettono di inserire (ad esempio all'interno di una variabile stringa) il carattere apice (') e le virgolette (") che altrimenti potrebbero essere interpretate come delimitatori di costanti

carattere. Per rappresentarle basta, infatti, farle precedere da una barra invertita (o backslash), così l'apice diventa \' e le virgolette \".

- **Operatori e separatori**

In combinazione con le parole chiave vengono spesso usati alcuni caratteri speciali per controllare il flusso delle operazioni che dobbiamo eseguire, questi caratteri vengono chiamati **operatori** e hanno delle proprietà che esporremo tra poco.
I **separatori** invece sono simboli di interpunzione che permettono di chiudere un'istruzione o di raggruppare degli elementi.
Riportiamo qui di seguito alcuni operatori secondo la notazione di Backus-Naur.

```
operatore:
  one of
+    -    *    /    %    ^    &    |    ~    !    =    <
>    +=   -=   *=   /=   %=   ^=   &=   |=   <<   >>   <<=
>>=  ==   !=   <=   >=   &&   ||   ++   --  ,    ->*  ->   .
*   ::   ()   []   ?:
separatore:
  one of
(    )    ,    ;    :    {    }
```

Alcuni simboli possono essere usati indistintamente come separatori o come operatori (ad esempio la virgola), l'utilizzo nella maniera più appropriata si evince dal contesto.

Gli operatori hanno delle proprietà che ne caratterizzano l'uso semplice o composito, tali proprietà permettono infatti di interpretare in modo unico il significato di un'espressione.
Ecco le proprietà degli operatori:

- ***POSIZONE***
  La posizione di un operatore rispetto ai suoi operandi (detti anche argomenti) è molto importante. Un operatore si dice **prefisso** se viene posto prima degli operandi, **postfisso** se viene posto dopo e **infisso** se si trova tra gli operandi.
- ***ARIETA'***
  L'arietà è il numero di argomenti di un operatore; ad esempio il simbolo "++" (che incrementa di uno) ha un solo argomento, il simbolo "+" e le parentesi hanno, invece, due argomenti e, nel caso del C++, l'operatore "?:" (operatore condizionale, simile all'IF) è l'unico ad avere tre argomenti.
- ***PRECEDENZA (o PRIORITA')***
  Per precedenza si intende l'ordine con il quale verranno eseguiti gli operatori. L'esempio classico è quello dell'espressione **4+7*5** in cui non sappiamo se bisogna eseguire prima l'operazione di somma o di moltiplicazione; premesso che l'operatore di moltiplicazione ha una priorità maggiore rispetto all'addizione, il modo coretto di interpretare l'espressione sarà **4+(7*5)**.
- ***ASSOCIATIVITA'***
  Questa proprietà ci viene in aiuto quando eseguiamo due o più operatori aventi stessa precedenza. Un operatore può essere associativo a **sinistra** oppure associativo a **destra**, questo vuol dire che iniziamo ad eseguire gli operatori partendo rispettivamente da sinistra o da destra.

Lezione successiva

[Sommario]

TORNA SU

GUIDA DI BASE

LEZIONE 7: *Variabili e Tipi di Dati*

Adesso è fondamentale definire il concetto di variabile, visto che nelle prossime lezioni faremo largo uso di questo termine. Pensiamo, ad esempio, 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). Si può vedere quindi che una variabile esiste in funzione del nome e del suo valore corrispondente; la comodità risiede (come nel cellulare) nel poter usare un nome per valori, che possono essere numeri o lettere, di grande entità o difficili da ricordare. Un altro vantaggio, non da 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 modificare il valore come e quante volte vogliamo. Un esempio tipico può essere quello di un programma che scrive a video il quadrato di un numero inserito dall'utente. Senza usare un linguaggio specifico si può vedere la comodità nell'uso della variabile "A" che può assumere un valore a piacere, senza per questo dover riscrivere il programma tutte le volte;

```
scrivi sullo schermo "Ciao Inserisci un numero";
A= -numero inserito da tastiera-;
B = A* A;
scrivi sullo schermo "Il quadrato di " A" è" B;
```

* Anche B è una variabile e viene usata per registrare il risultato finale

In questo esempio si può vedere come se ad esempio inseriamo come numero il valore "4", sullo schermo verrà scritto "Il quadrato di 4 è 16", ma se scriviamo un altro numero cambierà anche la scritta.
E cosa succede se invece di un numero digitiamo un carattere? Generalmente il programma darà un errore e si fermerà questo perché si aspetta un numero e non un carattere; infatti riuscire a classificare la tipologia di dati che devono essere inseriti e poi utilizzati all'interno di un programma è molto importante per evitare errori o per rendere il programma più veloce.
Qui di seguito ci rifacciamo ai tipi di dati presenti nel C++ e generalmente adottati dai più comuni linguaggi di programmazione:

- **Tipo Booleano**
  Il tipo **`booleano`** è il tipo più semplice esistente, perché può assumere solamente i valori **vero** (o 1) e **falso** (o 0). Gli operatori su tali valori sono quelli dell'algebra di Boole, esposta in una precedente lezione.

- **Tipo Intero**
  Il tipo **`intero`** è costituito dai numeri interi compresi tra un limite inferiore ed un limite superiore; tali limiti dipendono dall'implementazione del compilatore e dal sottotipo (short, long,

unisgned), ma generalmente sono costituiti da valori che vanno da -2^{N-1} a $+2^{N-1}$ dove N è il numero di bit utilizzati per rappresentare gli interi (generalmente 16 o 32). Oltre tali limiti il compilatore genera un errore dicendo che non può rappresentare il numero.

- **Tipo Reale**
  Il tipo **`reale`** è formato da valori che appartengono ad un sottoinsieme dei numeri razionali; la rappresentazione è limitata prima di tutto dai numeri presenti dopo la virgola quindi risulta impossibile rappresentare, ad esempio, numeri periodici e, secondariamente, esiste una limitazione inferiore e superiore come quella del tipo intero ovviamente dipendente dal compilatore.

- **Tipo Enumerazione**
  Il tipo **`enumerazione`** non è presente in tutti i linguaggi di programmazione e serve per rappresentare costanti intere, predefinite dal programmatore, associate a informazioni non numeriche. L'esempio tipico può essere quello di associare un valore ai giorni della settimana per compiervi delle operazioni. Il vantaggio principale, quindi, si riduce al poter eseguire operazioni matematiche su elementi apparentemente non matematici.

- **Tipo Carattere**
  Il tipo **`carattere`** rappresenta i caratteri che normalmente sono visibili sullo schermo e stampabili su carta più alcuni caratteri speciali che dipendono dall'implementazione. La codifica usate più spesso è quella **ASCII** in cui un carattere occupa un byte. Da far notare che nella rappresentazione 'A' risulta diverso da 'a' e che '7' è tutt'altra cosa rispetto all'intero 7.

Naturalmente in ogni linguaggio di programmazione è possibile eseguire operazioni miste o conversioni (sia implicite che esplicite) di tipo.

Lezione successiva

[Sommario]

TORNA SU

GUIDA DI BASE

LEZIONE 8: *La programmazione Condizionale*

Il modo in cui strutturiamo un ragionamento, generalmente, segue la logica procedurale, cioè avendo delle operazioni da fare le eseguiamo una dopo l'altra in sequenza dalla prima all'ultima; anche nella programmazione avviene la stessa cosa, infatti quando scriviamo un programma dobbiamo pensare a creare una schermata introduttiva, chiedere i dati all'utente e poi fornirgli il risultato. Senza entrare nel dettaglio, perché questa guida è per principianti cercheremo ora di spiegare i costrutti più usati nei linguaggi di programmazione per "incanalare" il flusso del nostro programma verso la soluzione (vedere la lezione che parla dei Diagrammi di Flusso); questi strumenti sono chiamati Istruzioni Condizionali perchè permettono di eseguire del codice a seconda che una condizione sia vera o falsa.

Il costrutto principale, usato universalmente, è l'**IF**, il quale permette di porre una condizione ed eseguire delle scelte in base ai dati fornitigli. Qui di seguito cerchiamo di spiegarlo sempre avvalendoci del linguaggio di Backus Naur e di un meta-linguaggio simile al C++.

istruzione-if :
if (*espressione*) *istruzione*
if (*espressione*) *istruzione* **else** *istruzione*

L'espressione che compare dopo la parola chiave **if** deve essere di tipo logico, se la condizione risulta vera viene eseguita l'istruzione subito seguente (then); nel secondo caso, invece, se la condizione risulta vera si esegue l'istruzione seguente, altrimenti si esegue l'istruzione subito dopo la parola chiave **else**. Poiché le istruzioni all'interno dell'if possono essere anche più di una è opportuno, per non creare confusione, inserirle all'interno delle parentesi graffe '{' e '}', anche se generalmente un else si riferisce sempre all'if che gli sta più vicino. Per più scelte invece si può usare l'**else if** che permette di porre una condizione anche per le alternative, lasciando ovviamente la possibilità di mettere l'else (senza condizioni) in posizione finale.
Qui riportiamo un semplice programma che utilizza il costrutto IF e le sue varianti finora esposte:

```
intero A = 50;
scrivi sullo schermo "Inserisci un numero tra 0 e 100";
intero B = -numero inserito da tastiera-;

if(B <A) {
  scrivi sullo schermo "Il numero inserito èminore di
cinquanta";
} else if(B >A) {
  scrivi sullo schermo "Il numero inserito èmaggiore di
cinquanta";
} else{    //poiché A non è minore o maggiore deve essere
uguale
  scrivi sullo schermo "Il numero inserito ècinquanta";
}
```

Questo programma aspetta che l'utente inserisca un numero tramite la tastiera (B) e poi lo confronta con il valore predefinito (50), a seconda che il valore inserito sia più grande, più piccolo o uguale a tale numero, il programma visualizza sullo schermo una frase che ci dice il risultato di tale confronto.

Lezione successiva

[Sommario]

TORNA SU

GUIDA DI BASE

LEZIONE 9: *La programmazione Iterativa*

Fino ad ora abbiamo visto che la modalità generale di programmazione è quella procedurale, ma esiste un'altra logica, meno comune, ed è quella iterativa; il concetto fondamentale è quello di eseguire un'istruzione (o una serie di istruzioni) continuamente, fino a quando non sopraggiungono delle condizioni che fanno terminare tale computazione. Facciamo l'esempio più semplice, voler stampare a video per mille volte la solita frase; la programmazione procedurale suggerisce di scrivere mille volte il codice, mentre quella iterativa permette di scrivere il comando **una ed una sola volta** e poi ripeterlo per mille volte, dopo le quali una struttura di controllo adeguata (ad esempio un contatore da uno a mille) termine l'esecuzione del ciclo.

I tre costrutti comunemente usati sono il **WHILE**, il **DO** ed il **FOR** che fanno praticamente la stessa cosa solo che il secondo, a differenza degli altri, esegue almeno una volta il ciclo.

L'istruzione **WHILE** viene schematizzata come segue:

```
istruzione-while :
  while ( condizione ) istruzione
```

Con questa istruzione viene prima valutata la condizione racchiusa tra le parentesi tonde, se l'espressione risulta vera viene eseguita l'istruzione all'interno del while e il while viene ripetuto, altrimenti si esce dal ciclo del while e si procede con il resto del programma.

Il **DO** può essere considerato una variante dell'istruzione while ed è strutturato nella maniera seguente:

```
istruzione-do :
  do istruzione while ( condizione ) ;
```

Prima di tutto viene eseguita l'istruzione racchiusa tra **DO** e **WHILE** (quindi si esegue almeno una volta), poi si verifica il risultato dell'espressione, se è vero si riesegue il DO, altrimenti si continua con l'esecuzione del resto del programma.

Il **FOR**, schematizzato qui sotto secondo la notazione di Backus-Naur, inizializza una variabile, pone una condizione (che deve essere vera o falsa) e poi modifica (incrementa o decrementa) la variabile iniziale.

```
istruzione-for :
  for ( inzializzazione ; condizione ; incremento )
    istruzione
```

Volendo schematizzare questo ciclo con il WHILE, ecco la struttura che vi si presenta:

```
inizializzazione
while( condizione)
{  istruzione
   incremento}
```

Il potere del FOR è appunto quello di poter modificare la variabile che serve per verificare la condizione iniziale, la quale decide se eseguire o meno il corpo del ciclo.

Tanto per far capire la potenza di questi strumenti, facciamo un esempio un po' più complesso, supponendo di dover trovare un nominativo all'interno di un elenco telefonico, come procediamo? la programmazione procedurale ci suggerisce di partire dalla A fino ad arrivare alla Z, in questo modo impiegheremmo molto tempo a trovare il numero di telefono, se ad esempio il cognome che volessimo cercare fosse "Valente" (nda :D). Sarebbe molto più comodo aprire l'elenco a metà e verificare se il cognome che stiamo cercando si trova a sinistra o a destra della pagina che abbiamo aperto, una volta scelta la metà opportuna (in questo caso quella di destra) basterà dividere in due parti anche quella e compiere di nuovo l'operazione di scelta. Questo signori è un processo ricorsivo che ci porterà molto più in fretta al numero di telefono che stiamo cercando:

```
carattere DACERCARE = "valente";
intero INDICE = 10000;
carattere NOMI_ELENCO[INDICE] = -variabile che
contiene tutti i nomi-;
carattere NUMERI_ELENCO[INDICE] = -variabile che
contiene tutti i numeri;

for(intero A = 0; A <INDICE;) {
//manca l'ultima opzione, quella di incremento, perché
presente dentro al FOR
   intero B = (A + INDICE) / 2;
   if (DACERCARE < NOMI_ELENCO[B]) {
      INDICE = B;
   } else if (DACERCARE > NOMI_ELENCO[B]) {
      A = B;
   } else {
      scrivi sullo schermo "Il numero cercato è"
NUMERI_ELENCO[B];
      -fine del programma-;    }
}
```

In questo programma abbiamo due variabili strane, NOMI_ELENCO e NUMERI_ELENCO, che sono array, ovvero delle variabili che permettono di contenere più valori accessibili tramite un indice; a parte questo bisogna concentrarci sul for, qui viene inizializzata una variabile A con valore 0, questo per permettere di scorrere tutti i valori da 0 a 10000, la seconda parte, la condizione di controllo, permette l'esecuzione del FOR (se A fosse maggiore di INDICE si uscirebbe dal FOR), mentre la terza ed ultima parte del FOR aumenta ad ogni ciclo il valore di A. La variabile B di appoggio servirà per avvicinarsi sempre di più al valore dell'INDICE di DACERCARE; Il corpo del FOR contiene un IF che verifica se DACERCARE sta a sinistra, a destra o è uguale al valore di NOMI_ELENCO con indice B. Per chiarire, mettiamo in questo caso che "valente" corrisponda a NOMI_ELENCO[6250], eseguiremo questi passi:

- verifichiamo cha A (=0) sia minore di INDICE (=10000)

- poniamo B = (0 + 10000)/2 = 5000
- controlliamo che "valente" stia a sinistra di NOMI_ELENCO[5000]
- poiché ciò non accade controlliamo che "valente" stia a destra di NOMI_ELENCO[5000]
- poiché accade poniamo A = 5000
- ricominciamo il ciclo e verifichiamo che A (=5000) sia minore di INDICE (=10000)
- poniamo B = (5000 + 10000) / 2 = 7500
- controlliamo che "valente" stia a sinistra di NOMI_ELENCO[7500]
- poiché accade poniamo INDICE = 7500
- ricominciamo il ciclo e verifichiamo che A (=5000) sia minore di INDICE (=7500)
- poniamo B = (5000 + 7500) / 2 = 6250
- poiché "valente" è uguale (non essendo ne' maggiore ne' minore) a NOMI_ELENCO[6250] visualizziamo sullo schermo il numero di telefono corrispondente in base all'indice e usciamo dal programma

Anche se siamo stati fortunati, si può capire il netto vantaggio nel trovare una cosa cercata con un ciclo e qualche confronto, invece di scorrere tutto l'indice dall'inizio alla fine.

Lezione successiva

[Sommario]

TORNA SU

GUIDA DI BASE

LEZIONE 10: *Le funzioni*

Quando sviluppiamo un programma capita spesso che alcune operazioni debbano essere ripetute più di una volta. Siccome scrivere tutte le volte le medesime operazioni risulterebbe tedioso quanto inutile ci viene in aiuto il concetto di funzione che altro non è che un programmino (o modulo) a se stante il quale prende in entrata dei valori e restituisce un risultato. Altro vantaggio è quello di poter modificare la struttura della funzione (per ottimizzarla o renderla più veloce) senza per questo dover intaccare il codice che la richiama.
Non volendoci addentrare nel dettaglio si può dire che una funzione ha bisogno di essere **dichiarata** e **definita**; cioè vanno specificati i tipi di ingresso e di uscita sui quali la funzione andrà a compiere le proprie operazioni (DICHIARAZIONE) e successivamente dovremo scrivere il "corpo" della funzione vera e propria (DEFINIZIONE). Da notare che se definisco una funzione questa è automaticamente anche dichiarata, mentre invece posso dichiarare una funzione senza definirla.
Per chiarire faremo un esempio pratico rifacendoci all'esempio dell'elevamento a potenza di una lezione precedente e ricordandoci che la funzione utilizza delle variabili proprie che vivono e muoiono al suo interno; infatti noi "passiamo" alla funzione il valore di una variabile, per la quale possiamo utilizzare dei nomi specifici e non quelli originari:

```
intero quadrato(intero ALPHA) {
// questa è la funzione quadrato (che restituisce un intero)
   intero QUAD = ALPHA * ALPHA;
   scrivi sullo schermo "Il quadrato di " ALPHA " è" QUAD;
   ritorna;   //esce della funzione
}

-inizio programma-
scrivi sullo schermo "Ciao Inserisci tre numeri";
intero A = -numero inserito da tastiera-;
intero B = -numero inserito da tastiera-;
intero C = -numero inserito da tastiera-;
quadrato(A); quadrato(B); quadrato(C);
-fine programma-
```

* In questo esempio quando chiamiamo la funzione quadrato(A) la variabile ALPHA assume lo stesso valore di A
** Si consideri che anche il programma principale è in definitiva una funzione che non restituisce valore

Si può vedere che questo programma fa quasi esattamente la stessa cosa del programma proposto precedentemente, cioè calcola il quadrato di un numero inserito da tastiera; direte voi, ma dove sta il vantaggio? il vantaggio su una singola operazione non è tangibile, ma in questo caso possiamo scrivere a video il quadrato di tre numeri usando solo cinque variabili e permettendoci di scrivere il codice per il risultato a video solo una volta.

Se avete letto tutta la guida e siete arrivati fino a questo punto potreste avere le idee confuse, soprattutto perché, forse, non riuscite a capire come utilizzare tutte le informazioni che avete acquisito. Non vi spaventate, è una cosa normalissima, infatti questa guida serve per introdurre nozioni che però approfondirete ed utilizzerete successivamente ed in maniera specifica per il linguaggio di programmazione che avete deciso di imparare.

Come nota finale ringrazio tutti quelli che mi hanno permesso di scrivere questa guida e spero che possa risultare utile e piacevole alla lettura così come è stato per me scriverla.

[S o m m a r i o]

TORNA SU