

# **Introduzione al Turbo Pascal**

## Alcuni richiami

### **Situazioni problematiche:**

Che cos'è un problema? Con quali metodi si affrontano i problemi?

La parola “problema” può assumere, ai giorni nostri, diversi e numerosi significati: si passa dai problemi legati alla vita di tutti i giorni, già di per sé numerosissimi e di natura diversa, a problemi legati alla matematica o alle scienze in generale.

Per questa lunga serie di motivi preferiamo parlare di *situazione problematica*.

Le risorse per risolvere un determinato problema sono sempre in numero finito. Ad esempio alcune situazioni problematiche non sarebbero tali se avessimo a disposizione un tempo infinito o mezzi a capacità infinita per la sua risoluzione. Per esempio ci troviamo in una situazione problematica quando dobbiamo prendere un treno e ci accorgiamo che mancano soli cinque minuti e siamo imbottigliati con la macchina nel traffico. Questa stessa situazione non rappresenterebbe un problema qualora il treno dovessimo prenderlo cinque ore dopo.

### **Formalizzazione:**

Ma il problema e la situazione problematica sono due modi diversi per dire la stessa cosa?

Apparentemente potrebbe sembrarci così, ma in verità non lo è.

Diciamo che si passa da una situazione problematica al problema dopo una presa di coscienza razionale che prende il nome di *formalizzazione*.

Vediamo velocemente quali sono gli ingredienti per una buona formalizzazione:

- Individuare lo scopo da raggiungere che, generalmente si chiama soluzione del problema.
- Identificare il repertorio delle azioni ammissibili (che chiameremo operatori) applicabili formalmente in una qualsiasi situazione del problema.
- Riconoscere quali tra i dati di partenza, disponibili all'inizio del procedimento risolutivo, sono effettivamente significativi.
- Prestare attenzione allo stato in cui si trova il problema (alcuni dati vengono fuori solo in alcuni stati del problema).
- Saper scomporre il problema in sottoproblemi (risolvibili in forma autonoma e che facilitano la risoluzione del problema dato).

Il passaggio più complicato per una buona formalizzazione del problema è sempre l'individuazione dei dati effettivamente significativi.

Dobbiamo notare che quando cerchiamo di formalizzare bene il nostro problema, non facciamo altro che trovare un modo per rappresentare la soluzione del problema stesso.

### **Gli algoritmi:**

Che cos'è un algoritmo, a cosa serve?

Un algoritmo può essere visto come un insieme finito di istruzioni che dobbiamo eseguire per risolvere un problema, ma anche tutti quelli che sono ad esso simili. L'esecuzione di queste istruzioni, che devono essere chiare e non ambigue, deve avvenire in un tempo comunque finito.

Facciamo un esempio di algoritmo che possa risolvere un certo problema, per esempio quello di calcolare il triplo di un numero naturale  $n$  dato:

1. Considera il numero naturale  $n$ ;
2. moltiplica questo numero per tre.

E' ovvio che queste due istruzioni danno vita ad un algoritmo che ci permette di risolvere il problema che ci eravamo posti. Logicamente possiamo modificare il nostro algoritmo in modo da renderlo più generico, capace di risolvere tutti i problemi di questo tipo, cioè calcolare un multiplo qualsiasi del numero  $n$  dato:


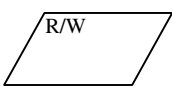
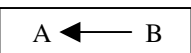
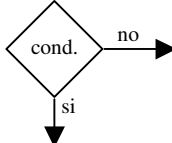
1. Considera il numero naturale  $n$ ;
2. Considera il numero  $m$  di volte che  $n$  deve essere sommato a se stesso;
3. Moltiplica  $m$  per  $n$ .


Con una semplice istruzione in più abbiamo generalizzato il problema precedente.

### **I diagrammi a blocchi (DaB):**

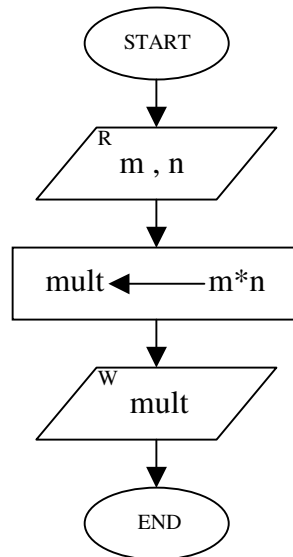
Il DaB fa parte dei cosiddetti linguaggi di progetto; essi servono a scrivere gli algoritmi sotto forma di diagrammi di flusso. E' una fase importante del percorso che dal *problema* ci porta al *processo*.

Nella tabella seguente riportiamo i costrutti di controllo e i componenti di un DaB:

	Indica l'inizio del DaB.
	Indica l'operatore di lettura ( R ) o scrittura ( W ).
	Indica l'operatore di assegnazione (nell'esempio il valore di B viene passato ad A).
	Indica il costrutto di selezione a due vie. A seconda del verificarsi o meno della condizione si seguono due differenti percorsi. Questo costrutto è usato anche per le iterazioni.

	Indica la fine del DaB.
---	-------------------------

Ora facciamo un piccolo esempio e scriviamo il DaB dell'algoritmo presentato in precedenza, cioè quello del multiplo di un numero naturale:



In questo esempio per primo vengono letti i valori delle due variabili  $m$  e  $n$ , poi viene assegnato alla variabile `mult` il risultato del prodotto tra  $m$  e  $n$ ; in fine viene scritto tale valore ed il programma ha termine.

### **Dal DaB al linguaggio ad alto livello:**

Dal DaB, che ricordiamo essere la codifica in linguaggio di progetto dell'algoritmo risolutore di un problema, si passa alla stesura del codice sorgente in un linguaggio di programmazione ad alto livello, attraverso una fase di codifica.

Il codice sorgente va scritto su un file (o più files), cioè abbiamo bisogno di un editor di testo che ci permettere di tradurre il DaB utilizzando una corretta sintassi, che può variare da linguaggio a seconda del linguaggio di programmazione che si utilizza.

Tramite questa sintassi particolare, il computer è capace di leggere ciò che noi vogliamo comunicargli.

Nella successiva fase di compilazione, il nostro codice sorgente viene tradotto in un codice intermedio (scritto in linguaggio macchina, cioè in binario, perché il computer è in grado di ragionare solo in termini di bit) e poi in codice eseguibile (il cosiddetto file \*.exe). Si giunge così al processo.

## ***Introduzione ai linguaggi di programmazione***

“Il computer è una macchina che elabora delle informazioni; in quanto macchina, però, non è capace di gestirle da sé, ma necessita di un operatore esterno che istruisca l’elaboratore e gli insegni come trattare queste informazioni.”

Questa definizione racchiude in sé il significato della parola programmazione.

Esistono due principali tipi di linguaggi con cui programmare una macchina: i programmi interpreti e i compilatori. Alla prima categoria appartengono numerosi programmi tra i quali spicca il BASIC (Beginners All-Purpose symbolic Instruction Code = Codice simbolico di istruzioni per tutti gli usi, per principianti), chiamato così grazie alla sua intuitività. Il suo funzionamento è semplice: il codice sorgente (quello introdotto dall’operatore esterno) viene tradotto in linguaggio macchina (ASCII) riga per riga ogniqualvolta il programma stesso viene eseguito.

Oltre agli interpreti però, come già detto, esistono anche i compilatori. PASCAL, FORTRAN (FORmula TRANslate = Traduzione di formule) e C, sono soltanto alcuni dei più diffusi linguaggi compilati.

I programmi compilatori funzionano nel modo opposto degli interpreti: dato il solito codice sorgente, il compilatore lo trasforma in blocco in codice eseguibile.

Questi due metodi di programmazione sono entrambi validi ma ciascuno offre vantaggi e svantaggi. I compilatori infatti hanno il vantaggio di essere più veloci rispetto agli interpreti, ma quest’ultimi sono più soggetti ad errori.

## ***Il Turbo Pascal***

In questa sede ci occuperemo estesamente di un particolare tipo di linguaggio di programmazione compilato, il Turbo Pascal. Il Pascal prende il suo nome dal matematico francese Blaise Pascal che fu il primo ad ideare una macchina calcolatrice: la Pascalina.

Questo linguaggio, però fu messo a punto nel 1960 da Niklaus Wirth, docente all' università di Zurigo. Esso si diffuse dapprima negli ambienti scientifici ma, successivamente, grazie alla sua versatilità ed alla sua semplicità ,si diffuse un po' dappertutto.

Passiamo ora alla parte operativa.

Una volta avviato il TP (Turbo Pascal N.d.A.) si presenterà un'interfaccia a menu abbastanza intuitiva. Compariranno vari menu, ma quelli principali sono tre: File, Run, Compile. Il primo è formato da New, per creare un nuovo documento di lavoro; Load, per aprire lavori già salvati; Save, registra un lavoro. Nel menu Compile l' unico comando che ci riguarda è Compile, che compila il programma. Spesso può risultare utile conoscere il comando Destination (nel menu Compile) per decidere se compilare il programma nella Ram oppure in un file eseguibile (nome\_programma.exe).

Una volta compilato il programma, però, dobbiamo eseguirlo, e perciò è necessario selezionare dal menu Run il comando Run.

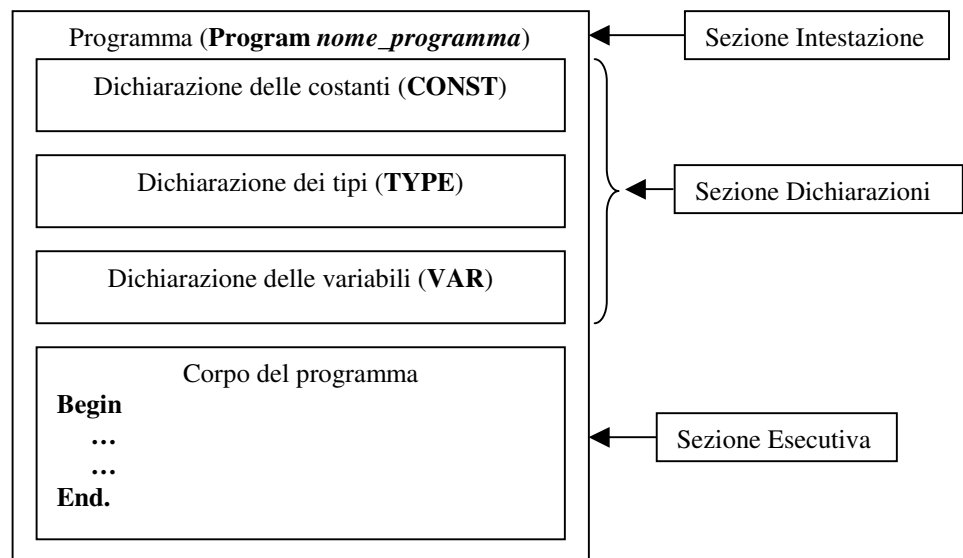
## Struttura generale e sintattica di un programma

Un programma scritto in TP consta in tre sezioni principali: *sezione intestazione*, *sezione dichiarazioni* e *sezione esecutiva*. Nonostante questi terribili nomi esse sono semplicissima comprensione: nella prima sezione si inserisce il nome del programma, nella seconda sezione vengono digitate le variabili e la loro tipologia, ed infine viene scritto il programma vero e proprio.

Per intenderci possiamo fare un banalissimo esempio: possiamo considerare il TP come un teatro in cui si inscena un'opera: il titolo dell'opera è dato dalla sezione delle intestazioni, gli attori che prendono parte all'opera vengono presentati nella sezione dichiarativa, mentre lo svolgimento dello spettacolo è rappresentato dal corpo del programma.

Esistono delle parole, dette riservate, a cui non può essere dato il valore di variabile, esse infatti rappresentano dei comandi che il TP può eseguire. Mi spiego meglio: tornando all'esempio del teatro, non possiamo chiamare un personaggio SIPARIO oppure SCENA, esse infatti sono parti specifiche del teatro e non possono essere perciò dei personaggi.

Ricapitoliamo quanto detto con l'ausilio di un semplice schemino:



Dopo questa breve introduzione al TP possiamo cominciare ad analizzare le più frequenti parole riservate e a capirne l'uso all'interno di un programma.

## **Abbozziamo un programma**

Ora cominceremo con l'abbozzo di un programmino semplice di cui abbiamo già descritto l'algoritmo risolutore in precedenza: calcolare il triplo di un numero dato.

D'ora in poi le parole riservate verranno scritte in maiuscolo. E' da ricordare che alla fine di qualsiasi istruzione bisogna inserire il punto e virgola (;). Anche se non è obbligatorio andare a capo dopo di esso, è preferibile farlo per rendere più agevole la lettura e la comprensione del listato.

Inoltre per inserire un commento che non interferisca con l'esecuzione del programma bisogna inserire il commento tra asterischi, e mettere il tutto tra parentesi tonde. Esempio (\* Questo è un commento \*).

Ecco qui il programma:

```
PROGRAM triplo; ( * Sezione dell' intestazione * )
VAR n:INTEGER; ( * Sezione delle dichiarazioni * )
begin ( * Corpo del programma * )
    writeln(' Questo programma calcola il triplo di un numero dato ');
    writeln;
    writeln('Introduci un numero intero ');
    readln(n);
    n:=n*3 ( * Vedremo più avanti l'operazione di assegnazione * );
    writeln;
    writeln(' Il triplo è ',n);
    writeln;
    writeln(' Premi enter per finire');
    readln;
end.
```

Nella parte dichiarativa abbiamo comunicato all'elaboratore che n è una variabile (VAR) che nel corso dell'esecuzione potrà avere soltanto un valore INTEGER, cioè di numero intero positivo o negativo. Esaminando successivamente il listato vedremo che la parte esecutiva è introdotta da BEGIN (= inizio). Poi troviamo una nuova parola riservata, READLN ( abbreviazione Read Line = leggi riga). Essa serve a leggere i dati di input.

La sua sintassi è molto semplice :

*READLN(nome variabile da leggere).*

Inoltre l'istruzione READLN non seguito da alcun parametro, crea un ciclo di attesa che dura finché, come si vuole in questo esempio, non viene premuto Invio.

Questa istruzione è d'obbligo alla fine di un listato, prima dell' END, poiché “congela” lo schermo quando dobbiamo leggere il risultato dell' elaborazione. Ad esempio, se eseguiamo il programma precedente senza READLN finale, non potremmo esaminare i risultati della nostra operazione, perché il programma si arresterebbe troppo in fretta.

Un'altra fondamentale istruzione è WRITELN ( abbreviazione Write Line = scrivi riga); essa serve a stampare i dati di output sullo schermo.

La sua sintassi è :

```
WRITELN('Stringa di testo');
```

Se consideriamo l'esempio sopra vedremo che il testo tra apici (') verrà stampato sullo schermo; nel caso in cui WRITELN venga usato senza parametri l'elaboratore lascerà una riga vuota nell'elaborazione.

Un ulteriore esempio sull'uso di WRITELN:

```
WRITELN(a)
```

In questo caso verrà visualizzato sullo schermo il valore assunto in quel momento dalla variabile *a*. Dunque, se *a* vale 3, allora sullo schermo uscirà il numero 3. E' possibile anche combinare stringhe e variabili, mettendo una virgola tra il secondo apice e la variabile:

```
Writeln(' Io sono ',a);
```

Alla fine del programma, troviamo END seguito da un punto. Esso indica la fine del flusso delle informazioni e il conseguente arresto del programma.

## ***Gli operatori booleani e i segni di operazione***

Come già visto in precedenza spesso in TP ci serviamo di operatori quali AND oppure OR (letteralmente “e” ed “o”). Essi, introdotti dal matematico inglese George Boole, servono a congiungere due proposizioni. L’operatore AND serve a dettare più condizioni e farle avverare solo se sono simultaneamente soddisfatte. L’operatore OR, invece, serve a dettare più condizioni e farle avverare se almeno una di queste è vera.

I segni di operazione in TP, sono i seguenti:

“+” ADDIZIONE

“-“ SOTTRAZIONE

“\*” MULTIPLICAZIONE

“/” DIVISIONE

“^” ELEVAMENTO A POTENZA

MOD = RESTO DELLA DIVISIONE TRA INTERI

DIV = PARTE INTERA DELLA DIVISIONE TRA NUMERI INTERI

Esistono altri operatori molto usati che hanno come dominio numeri reali: ABS(x), SQR(x), SIN(x), COS(x), TAN(x), ARCTAN(x), LN(x), EXP(x), SQRT(x).

Inoltre per indicare il rapporto fra due cifre ci serviamo dei seguenti simboli matematici :

“<” MINORE

“>” MAGGIORE

“=” UGUALE

“=>” MAGGIORE O UGUALE

“<=” MINORE O UGUALE

“<>” DIVERSO DA ...

Un’operazione tra due o più dati di uno stesso tipo si dice interna se il risultato ottenuto è del medesimo tipo dei dati utilizzati nell’operazione. Se ciò non succede l’operazione si dice esterna.

Per esempio, l’operazione DIV è sempre interna in quanto rappresenta l’operazione di divisione di interi, ed il risultato è un intero (o comunque la parte intera). Se operiamo la divisione tra due numeri interi utilizzando l’operatore “/” il risultato sarà comunque un numero reale, quindi questa operazione è esterna.

## Operazione di assegnazione

L'operazione di assegnazione di un qualsiasi valore ad una variabile è molto importante in un qualsiasi linguaggio di programmazione ad alto livello. Essa viene esplicitata in questo modo:

$$\langle \text{variabile} \rangle = \langle \text{espressione} \rangle$$

dove nell'espressione possono essere contenute variabili, costanti e operatori.

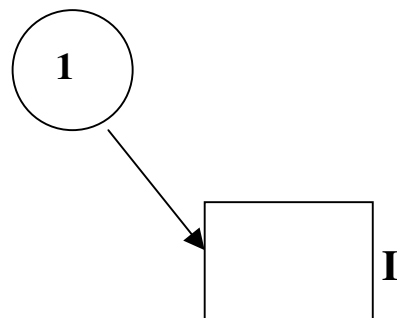
Non dobbiamo leggere il simbolo di uguaglianza (=) nel modo usuale della matematica in quanto non si tratta di uguaglianza, ma di tutt'altra cosa.

Consideriamo il seguente esempio di assegnazione:

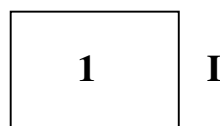
$$I = I + 1$$

E' evidente che matematicamente ciò non avrebbe alcun senso. Quell'uguale sta per assegnazione, cioè letteralmente si dice all'esecutore di assegnare alla variabile I il valore I+1, cioè il valore che aveva in precedenza aumentato di uno.

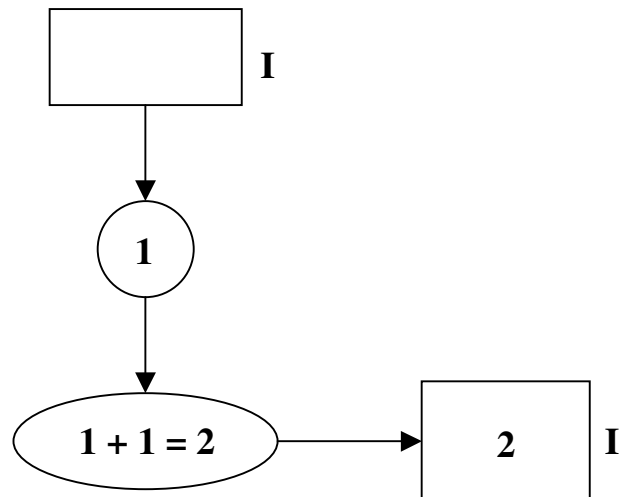
L'idea ci risulta più semplice se consideriamo la variabile come una scatola, nell'esempio di nome I, nella quale viene posto un valore. Facciamo un semplice schemino:



Alla variabile I, rappresentata da una scatola, viene assegnato il valore 1. Alla fine di questa operazione di assegnazione, nella scatola ci sarà il valore 1:



A questo punto la scatola può anche essere svuotata e successivamente riempita con altri valori. Infatti, quando facciamo un'assegnazione del tipo  $I = I + 1$  non facciamo altro che svuotare la scatola e successivamente riempirla col nuovo valore  $I + 1$ :



Quindi abbiamo tirato fuori dalla scatola I il valore 1 e lo abbiamo sommato ad 1 (come nell'esempio) ed il valore ottenuto, cioè 2, lo abbiamo messo nella scatola I, che adesso non conterrà più 1, ma 2.

Dobbiamo dunque pensare all'operazione di assegnazione in questi termini e non in senso matematico.

In Pascal l'operazione di assegnazione avviene antepoendo all'uguale il simbolo dei due punti (:). Se facciamo riferimento all'esempio precedente scriveremo:

$$I := I + 1.$$

## I tipi di dati

Come già visto in precedenza, nel settore dichiarativo di un programma bisogna definire il tipo di variabili che intendiamo utilizzare nel corso del programma stesso.

Esistono vari tipi di variabili numeriche, e sono raggruppate nella seguente tabella:

Nome tipo variabile	Intervallo consentito	Commenti
<b>Numeri Interi</b>		
SHORTINT	Da -128 a +127	Numeri interi (positivi e negativi)
INTEGER	Da -32768 a +32767	Numeri interi (positivi e negativi)
BYTE	Da 0 a 255	Numeri interi positivi
WORD	Da 0 a 65535	Numeri interi positivi
LONGINT	Da -2147483648 a +2147483647	Numeri interi (positivi e negativi)
COMP	Da $-9.2 \times 10^{18}$ a $9.2 \times 10^{18}$	Numeri interi (positivi e negativi)
<b>Numeri Reali</b>		
REAL	Da $2.9 \times 10^{-36}$ a $1.5 \times 10^{35}$	Numeri reali (positivi e negativi)
SINGLE	Da $1.5 \times 10^{-45}$ a $3.4 \times 10^{38}$	Numeri reali (positivi e negativi)
DOUBLE	Da $5 \times 10^{-324}$ a $1.7 \times 10^{308}$	Numeri reali (positivi e negativi)
EXTENDED	Da $3.4 \times 10^{-4932}$ a $1.1 \times 10^{4932}$	Numeri reali (positivi e negativi)

Questa tabella dimostra come ci siano vari tipi di dichiarazione dei dati. Non è detto, però, che i tipi di dati corrispondano soltanto a cifre; potremmo infatti trovarci di fronte a un problema che necessita di dati alfanumerici, cioè di numeri e lettere.

In questo caso dovremmo introdurre due nuovi tipi di variabile:

1) **STRING[x]** : Dichiarando così un tipo di dato assumeremo che quel dato è formato da un numero intero di x caratteri alfanumerici (compresi gli spazi). Usato senza parametri non definisce la lunghezza della stringa. Inoltre nel caso in cui si dichiara una lunghezza minore del dato inserito, l'elaboratore non considererà i caratteri che eccedono. Ad esempio, se dichiarassimo la variabile a di tipo **STRING[5]** e al momento di inserire i dati digitassimo la parola "folletto", essa risulterebbe troncata in "folle", poiché i caratteri in eccesso non verrebbero considerati nell'elaborazione.

2) **CHAR** : E' meno usata di **STRING**, ma qualche volta può risultare utile. Viene usata per definire variabili formate da un solo carattere (eccetto numeri).

Se per un attimo torniamo alla sezione delle dichiarazioni, notiamo che il TP ci consente di definire dei tipi personalizzati di variabili tramite l'istruzione **TYPE**.

## **Le strutture iterative**

Il termine iterare deriva dal latino e significa “ripetere”. Questa struttura (che è presente in tutti i linguaggi programmatici, non solo in Pascal) agisce su una operazione che deve essere ripetuta più volte. Ne esistono vari tipi:

**Iterazione enumerativa** - Come dice la parola stessa esegue un’operazione, o un blocco di operazioni,  $n$  volte. In Pascal è indicata dal comando:

```
FOR a := ni TO nf DO( * Niente punto e virgola * )
    istruzione 1;
    istruzione 2;
    istruzione ...;
```

END;

Nel nostro caso  $ni$  sostituisce il numero intero dal quale si parte a enumerare e  $nf$  sostituisce il numero fino al quale si continua a enumerare. Dopo questa istruzione di iterazione si inseriscono le istruzioni da ripetere racchiudendo il tutto con un END seguito da un punto e virgola (;), poiché il flusso di informazioni non è del tutto terminato.

E' possibile anche procedere a ritroso, contando alla rovescia da un numero  $n$  fino ad arrivare ad un numero  $t$ : *FOR a := n DOWNTO t DO*

Ecco un semplice esempio:

```
PROGRAM potenza;
USES crt;
VAR a,b,c,i:INTEGER;
BEGIN
    WRITELN('Inserisci un numero');
    READLN(A)
    WRITELN(' Inserisci esponente ');
    READLN(B);
    FOR I:= 1 TO b DO
        c:=a*a;
    END;
    WRITELN('Il risultato della potenza è ',c);
    READLN;
END.
```

**Iterazione per falso** – In questo caso si esce dal ciclo racchiuso tra REPEAT e UNTIL se, e solo se, la condizione espressa da UNTIL è vera, in caso contrario continua a ripetere le istruzioni. Questo costrutto fa sì che le istruzioni all'interno del ciclo vengano eseguite almeno una volta. E' importante utilizzare tale costrutto quando si vuole operare una verifica su dati in input.

*REPEAT*

*istruzione 1;*

*istruzione 2;*

*istruzione ...;*

*UNTIL proposizione;*

Ci dobbiamo ricordare di inserire proposizioni possibili dopo UNTIL, altrimenti l'uscita dal ciclo ripetitivo non si avrà mai! ( in gergo si dice che il computer si "impianta" e bisogna riavviarlo).

Esempio:

*PROGRAM ripetizione;*

*USES crt;*

*VAR a,b:INTEGER;*

*BEGIN*

*REPEAT*

*WRITELN(' Inserisci un numero minore di 30 e maggiore di 10');*

*READLN(A);*

*UNTIL (A<30) AND (A>10);*

*READLN;*

*END.*

**Iterazione per vero** - Esegue le informazioni racchiuse tra WHILE DO e l' END; (il punto e virgola indica che l'END è parziale) finché la condizione presa in esame non sia falsa.

*WHILE a = 3 DO*

*istruzione 1;*

*istruzione 2;*

*istruzione...;*

*END;*

Questo semplice esempio spiega come il blocco di istruzioni presente tra WHILE DO e END; venga ripetuto finché si rispetti la condizione  $a = 3$ . Nel caso in cui il valore di  $a$  fosse stato diverso da 3, infatti, il blocco di istruzioni non sarebbe stato mai eseguito.

## Le strutture selettive

A questo punto è lecito domandare se è possibile operare delle scelte con il TP. La risposta è affermativa. Esistono due tipi di selezione: la *selezione binaria* e la *selezione multipla*.

**Selezione binaria** – E' la più semplice: se consideriamo un numero naturale qualsiasi e ci domandiamo se esso è pari, la risposta potrà avere solo due alternative: Si oppure No. In TP c'è la possibilità di eseguire una scelta binaria tra due condizioni con il seguente comando:

```
IF <proposizione1> THEN <istruzione1> ELSE <istruzione2>
```

Che tradotto fa più o meno:

```
SE <proposizione1>(è vera) ALLORA(esegui) <istruzione1> ALTRIMENTI(esegui) <istruzione2>
```

Questo è quanto serve per operare (o far operare al computer) una scelta.

Per chiarire ulteriormente questo importante processo presentiamo un esempio. Costruiamo un programma che calcoli l' area di un rettangolo SE E SOLO SE l'area stessa è minore di un numero  $n$  ( per esempio 20) e maggiore di un numero  $m$  (per esempio 10):

```
PROGRAM area_rettangolo;
USES crt;
VAR a,b,c:INTEGER;
BEGIN
    CLRSCR; ( * E' un nuovo comando.E' l' abbreviazione di CLear SCreen=
              cancella schermo. Pulisce lo schermo. * )
    WRITELN(' Inserisci base');
    READLN(a);
    WRITELN('Inserisci altezza');
    READLN(b);
    c:=a*b;
    IF (c>20) OR (c<10) THEN
        BEGIN
            WRITELN('Area maggiore di 20 oppure minore di 10. ');
            READLN;
        END ( * L' END prima dell' ELSE senza punto e virgola(;)* )
    ELSE
        BEGIN
            WRITELN(' AREA = ',c);
            READLN;
        END;
END;
```

*READLN;*

*END.*

E' un listato semplice, e l'unica osservazione da fare è che se dopo IF...THEN c'è solo un'istruzione può essere scritta terminando il tutto con il solito punto e virgola(;). Tuttavia ,se ce ne sono più di una è necessario aprire un altro BEGIN e cominciare un sottoprogramma il quale, però va terminato con l' END; che, se seguito da ELSE non accetta il punto e virgola(;).

**Selezione multipla** - Se domandiamo ad una persona quali sono i piatti che preferisce essa sceglierà fra un'ampia gamma di pietanze fino a scegliere quella che soddisfi i suoi gusti. Infatti, al contrario di prima, la selezione multipla ci fornisce illimitate possibilità di risposta. Il comando che in TP ci permette di operare un selezione multipla è il seguente:

*CASE n of ( \* Se n è stata dichiarata INTEGER \* )*

*valore 1:BEGIN*

*istruzione1;*

*istruuzione2;*

*istruzione...*

*END;*

*CASE n of ( \* Se n è stata dichiarata CHAR \* )*

*valore x:BEGIN*

*istruzione1;*

*istruzione2;*

*istruzione...;*

*END;*

*ELSE*

*BEGIN*

*istruxione 1;*

*istruzione2;*

*istruzione...;*

*END;*

*END;*

Nonostante la sua sintassi spaventosamente ricca, questa struttura è di facilissima comprensione. Innanzitutto dobbiamo decidere se dichiarare la variabile n come tipo INTEGER oppure CHAR. Se n è CHAR dovremo usare UNA lettera seguita dai due punti(:) e dal rispettivo sottoprogramma delimitato da BEGIN e END; . Se invece n è INTEGER dovremo usare un numero naturale al posto di n e mettere dopo i due punti(:) il solito sottoprogramma delimitato da BEGIN e END; .

Proviamo ora con un esempio: un programma che generi un numero a caso tra 1 e 10 e lo comunichi all'operatore solo se esso è uguale a 7, 9 o 3.

```
PROGRAM numeri_casuali;
USES crt;
VAR a,b,c:INTEGER;
BEGIN
  CLRSCR;
  RANDOMIZE; ( * Inizializza, ossia azzera il generatore di numeri casuali * )
  c:=RANDOM(10); ( * Questo comando permette all' elaboratore di generare un
                 numero a caso compreso tra 0 e il numero indicato in
                 parentesi. Il massimo consentito è 255 * )
  CASE c OF
    7:BEGIN
      WRITELN(' Il numero è ',c);
    END;
    9:BEGIN
      WRITELN(' Il numero è ',c);
    END;
    3:BEGIN
      WRITELN(' Il numero è ',c);
    END;
  ELSE
    BEGIN
      WRITELN(' Il numero uscito è diverso da 3 7 e 9);
    END;
  END;
```

Anche se con questo ultimo esempio abbiamo terminato il paragrafo riguardante le strutture dei dati al lettore potrebbe sembrare che quelle poche strutture analizzate debbano essere affiancate da molte altre per costruire un programma relativamente complesso, e invece secondo il teorema di Bohm - Jacopini: “Le tre strutture matematiche che servono per creare un qualsiasi programma sono: iterative, selettive e sequenziali.”

Le strutture sequenziali sono le più lineari, poiché il flusso di dati scorre senza ripetizioni né selezioni che abbiamo utilizzato tranquillamente nel primo programma (quello del triplo di un numero) senza saperlo.

## ***Procedure e funzioni***

All'inizio di questo lavoro abbiamo posto l'accento sull'importanza della scomposizione di un problema in sottoproblemi le cui soluzioni possono portarci alla risoluzione del problema di partenza.

Possiamo quindi inquadrare le procedure e le funzioni in quest'ottica; esse, infatti, possono essere definite come moduli di codice sorgente. Il loro compito è quello di suddividere programmi vasti e complessi in unità di modeste dimensioni. Sia le funzioni che le procedure hanno una struttura simile a quella di un programma; possono essere presenti dichiarazioni di costanti, di tipo, di variabili, funzioni o procedure annidate internamente proprio come in un programma.

Sia le procedure che le funzioni devono essere inserite tra il settore dichiarativo ed il corpo del programma (per intenderci, prima del BEGIN iniziale).

La struttura di una procedura è la seguente:

```
procedure <nome_procedura>;  
    <definizioni di costanti>  
    <definizioni di tipo>  
    <definizioni di variabili>  
    <definizioni di procedure annidate>  
begin  
    <corpo della procedura>  
end;
```

Una funzione si comporta come una procedura ad eccezione del fatto che essa restituisce sempre un valore; tipicamente il valore di ritorno di una funzione rappresenta il risultato di calcoli basati su valori passati alla funzione nella fase di esecuzione del programma.

## ***Chiamate a procedure e funzioni***

Una procedura viene chiamata esplicitamente attraverso una istruzione costituita solo dal nome della procedura; una funzione viene chiamata in modo implicito utilizzando il suo nome in una espressione all'interno di una istruzione (spesso si tratta di una istruzione di assegnazione).

Ad esempio :

```
Program ProcsAndFuncs;
var
    N : integer;
procedure Ciao;
begin
    Writeln ('Ciao!');
end;
function F17: integer;
begin
    F17 := 17;
end;
begin
    Ciao; (* chiamata alla procedura Ciao*)
    N := F17; (* chiamata della funzione F17 *)
end.
```

Una procedura o funzione restituisce il controllo al programma chiamante quando viene completata l'esecuzione dell'ultima istruzione o, in alternativa, quando viene incontrata l'istruzione EXIT.

Qualsiasi variabile dichiarata in una procedura o funzione si dice che è *locale*. Perciò queste variabili vengono create in fase di chiamata della procedura o funzione e vengono distrutte appena il controllo passa al programma chiamante. Inoltre, come nel caso delle *variabili globali*, il loro valore è indefinito nel momento in cui si entra in una procedura o funzione.

Le variabili globali sono quelle definite nel programma principale e che pertanto possono essere definite statiche.

Le variabili che figurano nella definizione di una procedura o funzione sono dette *formali*.

Il passaggio di parametri fra il programma e la procedura o funzione può avvenire in due modi:

- *Per valore* (senza la parola chiave VAR);
- *Per indirizzo* (con la parola chiave VAR).

Quando in fase di esecuzione di un programma viene passato ad una Procedura o Funzione un parametro per valore, viene trasferita una copia del parametro (non il parametro stesso); in effetti

---

questa copia diviene a tutti gli effetti una variabile locale della routine chiamata. All'interno della routine tale variabile può subire modifiche senza che queste influenzino il valore originario assunto a livello di programma principale chiamante.

Al contrario, quando in fase di esecuzione di un programma viene passato ad una Procedura o Funzione un parametro per indirizzo, viene trasferito l'indirizzo del parametro. Ciò significa che le modifiche subite dalla variabile all'interno della procedura o funzione si ripercuotono sul valore originario della variabile assunto a livello di programma principale chiamante. Solamente le variabili (né costanti e nemmeno espressioni) possono essere utilizzate come parametri VAR; solo per le variabili infatti ha senso parlare di indirizzo di memoria.

## **Vettori e matrici**

Un vettore o “array” è un insieme ordinato di variabili tutte dello stesso tipo; ciascuna di esse viene individuata attraverso il nome dell’array e la posizione o le coordinate della variabile entro l’array.

Consideriamo la dichiarazione:

```
Var Y : array[1..100] of real;
```

essa significa che Y rappresenta un vettore costituito da 100 elementi reali Y[1], Y[2], Y[3], ..., Y[100].

Ciascuna di queste 100 variabili può essere usata alla stessa stregua di una comunissima altra variabile di tipo “real”. Inoltre, e questo è ciò che rende facile l’uso degli array, il valore dell’indice può essere rappresentato da una qualsiasi espressione che dia come risultato un valore compreso fra 1 e 100.

Dobbiamo tuttavia prestare attenzione ed evitare che i valori assunti dagli indici degli array provochino uno sconfinamento al di fuori del campo di definizione dato in sede di dichiarazione.

Ad esempio, se la variabile J è di tipo integer, i valori assunti dalla J in questo ciclo iterativo *For* *for J := 1 to 100 do Y[J] := 0.0;*

ci permettono di azzerare tutto l’array.

Gli array possono anche essere di tipo multi dimensionale; cioè in altri termini si possono avere arrays di arrays (di arrays...). Supponiamo di avere un vettore Alfa bidimensionale (o matrice Alfa) costituito da 50 x 10 elementi reali; la dichiarazione può essere stesa in due modi distinti:

### **Modo A:**

```
Var
```

```
    Alfa : array[1..50,1..10] of real;
```

### **Modo B:** Da preferire

```
Var
```

```
    Alfa : array[1..50] of array[1..10] of real;
```

È evidente che per individuare un solo elemento sarà necessario dare i due valori degli indici che forniscono le coordinate dell’elemento entro l’array (matrice a due dimensioni).

```
Alfa[30,5] = -2.5E-10
```

Gli elementi di un array a più dimensioni vengono collocati in memoria con il seguente ordine:

```
Alfa[1,1], Alfa[1,2], ..., Alfa[1,10],
```

```
Alfa[2,1], Alfa[2,2], ..., Alfa[2,10],
```

```
.....
```

```
Alfa[50,1], Alfa[50,2], ..., Alfa[50,10].
```

Notiamo come varia più rapidamente l'ultimo indice.

Dobbiamo fare alcune riflessioni sulle dichiarazioni di Tipo che possono dare problemi quando vengano in modo sistematico usati gli ARRAY. Consideriamo il programma riportato di seguito

*Program UNO;*

*var*

*A : array[1..30] of byte;*

*B : array[1..30] of byte;*

*Begin*

*...*

*A := B;*

*...*

*End.*

L'istruzione *A := B* viene considerata errata dal compilatore in quanto le variabili A e B vengono considerate diverse.

*Program DUE;*

*type*

*Y = array[1..30] of byte;*

*var*

*A,B : Y;*

*Begin*

*...*

*A := B;*

*...*

*End.*

In questo caso l'istruzione *A := B* viene considerata corretta dal compilatore in quanto le variabili A e B vengono considerate dello stesso tipo.

Notiamo anche come l'indice di un array può avere un insieme di definizione diverso dal classico [1..N] come possiamo osservare nella dichiarazione seguente:

*Z : array[-5..+5] of real;*

## **Record**

I Record sono un tipo di variabile strutturato costituito da un insieme di variabili di tipo diverso fra loro (gli elementi costituiscono i vari campi del record). Possiamo accedere al record per intero o individualmente ai suoi elementi componenti chiamati campi. La forma sintattica per la dichiarazione di tipo è la seguente:

*Type*

```

Nome_record = record
    <campo_1>: <tipo>;
    <campo_2>: <tipo>;
    ...
    ...
    <campo_n>: <tipo>;

```

*end;*

Dopo aver definito una variabile di tipo record è possibile assegnare ai vari campi quantità determinate specificando il nome della variabile seguito da un punto e dal nome del campo. È evidente che le quantità che vengono assegnate al campo di un record devono essere compatibili con la dichiarazione di tipo del campo.

Per esempio :

*type*

```

alfa = record
    nome : string[40];
    eta : integer;

```

*end;*

*var*

```

    Studente : alfa;

```

*begin*

```

    Studente.nome := 'Marco Rossi';
    Studente.eta := 17;

```

*end.*

L'istruzione Pascal WITH ci consente di accedere in modo abbreviato ai vari campi di un record. All'interno dell'istruzione WITH (che può comprendere un blocco delimitatore Begin/End) non è più necessario utilizzare il nome del record per accedere ai suoi campi.

Le istruzioni di assegnazione, presentate nel precedente esempio, assumono con l'istruzione WITH la seguente forma:

```
begin
    WITH Studente do
        begin
            nome := 'Marco Rossi';
            eta := 17;
        end;
    end.
```

Se combiniamo tra loro vettori e record, possiamo costruire delle *tabelle* intese come vettori di record. Infatti un vettore si distingue dal record per il fatto che deve contenere elementi tutti dello stesso tipo, cosa non necessaria se consideriamo un record.

Facciamo un esempio di tabella:

```
Program tab_studenti;
const
    num_stud = 500;
type
    rec_alunni = record
        cognome:string[50];
        nome:string[50];
        eta:integer;
        indirizzo:string[250];
    end;
var
    alunno = array[1..num_stud] of rec_alunni;
    (*il vettore alunno è inteso come vettore colonna*)
begin (* assegniamo dei valori alla prima riga della tabella*)
    alunno[1].cognome:= 'Rossi';
    alunno[1].nome:= 'Marco';
    alunno[1].eta:=17;
    alunno[1].indirizzo:= 'via risorgimento 31'
end.
```

Logicamente avremmo potuto anche usare l'istruzione WITH, ma forse così rendiamo meglio l'idea.

## **Ordinamento e ricerca**

Spesso è molto importante ordinare i dati inseriti in un vettore in modo crescente o decrescente, ma è altresì importante cercare in modo efficiente un elemento di un vettore.

Esistono alcuni noti algoritmi di ordinamento e di ricerca, ma in questo lavoro ne presenteremo solo alcuni. Per quanto riguarda l'ordinamento di un vettore presentiamo il *BubbleSort* (ordinamento "a bolle") e l'*InsertionSort* (ordinamento per inserimento). Per la ricerca utilizzeremo l'algoritmo di ricerca completa sequenziale.

Nel *BubbleSort* viene ordinato un vettore partendo dall'elemento che si trova più a sinistra, che viene confrontato col suo immediato vicino a destra. Se risulta minore del vicino allora resta al suo posto e viene confrontato con l'altro elemento alla sua destra, cioè il terzo. In questo modo vengono confrontati tutti gli elementi del vettore finché non si troverà il più piccolo di essi che verrà posizionato al primo posto. Si ripete la stessa procedura per il secondo elemento del vettore e così via fino a l'ultimo, che risulterà, ovviamente, il più grande.

Ecco il programma in Pascal:

```

program bubblesort;
const
    MAXEL = 1000;
type
    vettore = array[1.. MAXEL] of integer;
var
    \vet: vettore;
    i, j, n: integer;
    aux: integer;
begin
    repeat
        writeln;
        write('Num. elementi: ');
        readln(n);
    until (n >= 1) and (n <= MAXEL);
    (*immissione dei dati nel vettore*)
    for i:=1 to n do begin
        writeln;
        write('Immettere elemento ',i,'°:');
    
```

```

        readln(vet[i]);
    end;
(*ordinamento*)
    for j := 1 to (n-1) do
        for i := (j+1) to n do
            if vet[j] > vet[i] then begin
(*scambio valori*);
                aux := vet[j];
                vet[j] := vet[i];
                vet[i] := aux
            end;
        end;
(*Visualizzazione*)
        writeln;
        for i:=1 to n do
            writeln(vet[i]);
        readln;
    end.

```

Adesso presentiamo il procedimento di ordinamento per inserimento. In pratica non facciamo altro che considerare il nostro vettore da ordinare come diviso in due parti, una che consideriamo già ordinata e l'altra da ordinare. Un elemento  $x$  passa dalla parte disordinata a quella ordinata nella posizione che gli compete ( $x$  viene confrontato via via con gli elementi alla sua sinistra fino a trovarle una posizione quindi, gli altri elementi vengono traslati a destra).

Ecco il programma in Pascal:

```

program Insertion_sort;
const
    MAX=1000;
var
    a:array[1..MAX] of real;
    i,j,n,p: integer;
    x,min: real;
begin
    write('Insertion Sort');
    repeat
        write('Immetti n:');

```

```
        readln(n);
until n <= MAX;
(*immissione dei dati nel vettore*)
for i := 1 to n do
begin
    write('a['i,','i,']:=');
    readln(a[i]);
end;
(*ricerca minimo*)
min := a[1]; p := 1;
for i := 2 to n do
    if a[i] < min then
        begin
            min := a[i];
            p := i;
        end;
a[p] := a[1];
a[1] := min;
(*ordinamento*)
for i := 3 to n do
begin
    x := a[i]; j:=i;
    while x < a[j-1] do
        begin
            a[j]:=a[j-1];
            j:=j-1;
        end;
    a[j]:=x;
end;
end.
```

Consideriamo adesso il problema della ricerca di un elemento all'interno di un vettore ordinato o non. In questo lavoro presentiamo un solo programma in Pascal che opera una ricerca completa e sequenziale di un elemento. In pratica questo programma, che presenteremo di seguito, controlla il

vettore elemento per elemento, partendo dal primo, finché non trova l'elemento cercato. In caso affermativo una variabile booleana ci avviserà che l'elemento è stato trovato.

Ecco il programma in Pascal:

```
program ricercaCompleta;
const
    MAXEL = 1000;
type
    vettore = array[1..MAXEL] of integer;
var
    vet: vettore;
    i, n, c : integer;
    trovato: boolean;
begin
    (*Immissione*)
        repeat
            writeln;
            write('Numero elementi: ');
            readln(n);
        until (n>=1) and (n<=MAXEL);
    (* Inizializzazione*)
        for i:=1 to n do
            vet[i]:=(37*i+c) mod n;
            write('Elemento target:');
            readln(c);
    (* Ricerca sequenziale*)
        i := 1;
        trovato := FALSE;
        while not trovato and (i<=n) do
            if c=vet[i] then
                trovato := TRUE
            else i := i+1;
        if trovato then begin
            writeln;
            writeln(c,' presente in posizione ', i);
```

```
end  
else  
begin  
    writeln;  
    writeln('non presente!')  
end;  
readln;  
end.
```