

Il linguaggio di programmazione

PASCAL

Caratteristiche del linguaggio Pascal

Progettato e realizzato da N. Wirth del Politecnico di Zurigo (inizio anni '70).
Linguaggio di tipo generale adatto sia per applicazioni scientifiche che gestionali.

Non è un linguaggio innovativo: tutti i concetti in esso contenuti si possono trovare in linguaggi precedenti (FORTRAN, COBOL, PL/I).
Per il corso si farà riferimento alla versione Turbo Pascal 7.0 della Borland.

Struttura di un programma in Pascal:

program nome_del_programma;

const

< dichiarazione delle costanti >

type

< dichiarazione dei tipi >

var

< dichiarazione delle variabili >

< dichiarazione delle procedure e delle funzioni >

begin

< corpo del programma >

end.

Le procedure e le funzioni sono dei sottoprogrammi che possono essere chiamati (e quindi fatti eseguire) da un punto qualsiasi del programma principale. Non sempre sono presenti in un programma. La loro struttura verrà esaminata in seguito.

Per inserire un commento questo deve essere racchiuso o tra parentesi graffe o da parentesi tonde seguite o precedute da '*'.
Esempio:

(* questo è un commento *)

TIPI DI DATI

Nello svolgimento di un programma si deve essere in grado di immettere nel sistema dei dati (input) e poi estrarre da esso le informazioni ottenute elaborando opportunamente i dati immessi (output). I dati da utilizzare vengono memorizzati nel sistema contrassegnandoli con opportuni identificatori (costanti o variabili).

Un tipo di dati determina:

- ❑ L'insieme dei **valori** che un oggetto (variabile o costante) di quel tipo può assumere.
- ❑ L'insieme di **operazioni** che possono essere effettuate su oggetti di quel tipo.
- ❑ La **rappresentazione in memoria** degli oggetti del tipo.

Type checking

Con questo termine si intende la verifica a tempo di compilazione della compatibilità tra il tipo dell'oggetto e l'operazione che viene eseguita su di esso.

Astrazione

Il tipo di dato consente di astrarre da :

- ❑ La rappresentazione in memoria dell'oggetto
- ❑ L'implementazione delle operazioni (che dipende dalla rappresentazione)

➤ Tutte le variabili devono essere dichiarate:

var x, y : integer;

var c : real;

var m, n : char;

➤ Una variabile deve appartenere ad uno ed un solo tipo (non sono ammesse conversioni implicite di tipo)

➤ La compatibilità di tipo deve valere nella istruzione di assegnazione

< variabile > := < espressione >

Eccezione: ad una variabile real può essere assegnato un valore di tipo integer

➤ Esiste la possibilità di dichiarare le costanti

```
const alfa = 1;
```

```
const pi_greco = 3.1415;
```

I tipi predefiniti

Il programmatore ha a disposizione, come si è detto, dei tipi di dati la cui definizione è implicitamente data dal linguaggio (dati standard):

- I tipi numerici
- Un tipo logico
- Il tipo carattere e stringa

Numeri interi senza virgola

Possono essere distinti in cinque sottogruppi (nel Pascal standard è presente solo l'integer):

<i>tipo</i>	<i>da</i>	<i>a</i>
byte	0	255
shortint	-128	127
integer	-32768	32767
word	0	65535
longint	-2147483648	2147483647

Numeri reali in virgola mobile

Possono essere distinti in quattro sottogruppi (nel Pascal standard è presente solo il real):

<i>tipo</i>	<i>da</i>	<i>a</i>
real	2.9×10^{-38}	1.7×10^{38}
single	1.5×10^{-45}	3.4×10^{38}
double	5.0×10^{-324}	1.7×10^{308}
extended	1.9×10^{-4951}	1.1×10^{4932}

Esempi di numeri reali:

```
num1:= 3.14;          num2:= 2.767E-5;
```

Col secondo esempio si è utilizzato un altro modo per rappresentare i numeri reali: la lettera E seguita da un numero corrisponde alla forma esponenziale ed equivale a “dieci elevato a ...” (nel caso precedente 10^{-5})

Tipo caratteri (char)

Il tipo char si usa per il singolo carattere (lettera alfabetica, cifra numerica da 0 a 9 o qualsiasi altro simbolo rappresentabile con il P.C.)

Esempi di char:

```
carattere1:= 'a';      carattere2:= '3';
```

Come si vede i caratteri vanno racchiusi tra due apici semplici.

Tipo booleano (boolean)

I dati boolean possono assumere solo due valori: vero (**true**) e falso (**false**).

I soli operatori consentiti sono:

not, and, or, xor.

Esempio:

```
var k, m, n: boolean;
```

```
k:= m and n;
```

Tipo stringa (string)

Non presente nel Pascal standard. Un dato string è un insieme di caratteri.

Esempi di string:

```
nome:= 'Marco';      data:= '12/03/1998';
```

N.B. Anche le stringhe vanno racchiuse tra due apici.

Il Pascal mette a disposizione del programmatore la possibilità di definire tipi di dati diversi di quelli predefiniti, secondo le proprie esigenze. Tale definizione viene effettuata utilizzando la parola riservata **type**.

La definizione di tipo viene effettuata prima di quella delle variabili.

Esempio:

```
type giorno = (lunedì, martedì, mercoledì, giovedì, venerdì, sabato, domenica);
```

```
    colore = (rosso, giallo, verde);
```

```
    secondi = 0..59;
```

```
var domani: giorno;  
    vernice: colore;  
    tempo: secondi;
```

Nell'esempio sono stati definiti nuovi tipi di dati: giorno, colore, secondi. Per ognuno di essi sono elencati gli elementi appartenenti al dato tipo. I primi due tipi sono detti enumerativi mentre il terzo tipo è detto subrange. Nei tipi subrange sono assegnati i valori inferiore e superiore di un certo intervallo di valori.

La dichiarazione di variabili successiva indica che all'identificatore *domani* potrà essere attribuito, nel corso del programma, solo uno dei valori del tipo *giorno*.

Procedure di immissione e visualizzazione dati

readln (num);

Quando, nel corso dello svolgimento di un programma, viene incontrata questa procedura il sistema si arresta e rimane in attesa che venga immesso un dato il cui valore viene assegnato alla variabile individuata dall'identificatore *num*. Il tipo di dato che viene introdotto dalla tastiera deve essere compatibile con il tipo che si è assegnato alla variabile *num* all'inizio del programma. La chiamata alla suddetta procedura inserisce un ritorno a capo automatico.

read (num);

In tal caso non viene inserito il ritorno a capo dopo la chiamata alla procedura.

writeln (num);

In questo caso viene visualizzato sul video il valore che ha la variabile *num* al momento della chiamata della funzione. Viene inserito un ritorno a capo e quindi eventuali altri dati verranno visualizzati nella riga successiva. Per visualizzare una stringa di messaggio questa va inserita tra due semplici apici. Per visualizzare più valori con una sola procedura le variabili o le stringhe di messaggio devono essere separate dalla virgola.

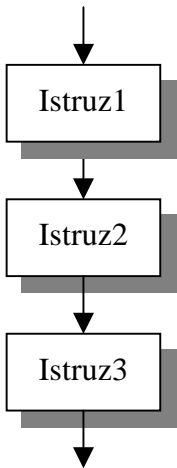
write (num);

Non effettua il salto di riga.

STRUTTURE DI CONTROLLO

Qualsiasi algoritmo, comunque formulato, è sempre riconducibile ad un algoritmo contenente le strutture logiche di controllo di seguito indicate.

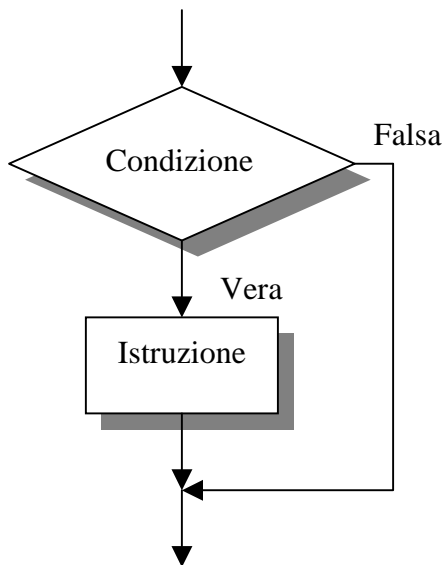
Sequenza



```
begin  
Istruz1;  
Istruz2;  
Istruz3  
end;
```

Strutture condizionali

Selezione



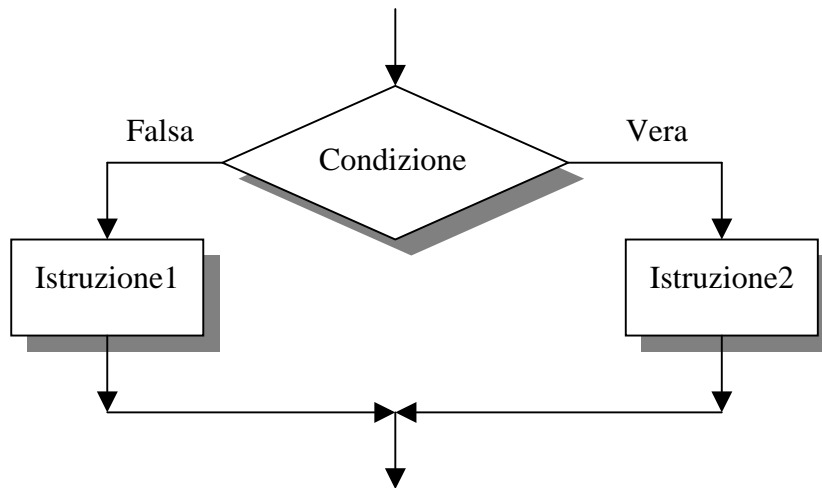
La sintassi della struttura è:
if condizione **then** istruzione;

esempio:

```
if a < b then  
a := a + b;
```

```
if a < b then  
begin  
a := a + b;  
b := 10  
end;
```

Le istruzioni che vengono eseguite quando è vera la condizione possono essere più di una, in questo caso bisogna racchiuderle tra **begin** e **end**.



La sintassi in Pascal di questa struttura è:

```

if Condizione then
    Istruzione1
else
    Istruzione2;
  
```

Al posto di *Istruzione1* oppure *Istruzione2* possono esserci più istruzioni racchiuse tra **begin ... end**

Esempio:

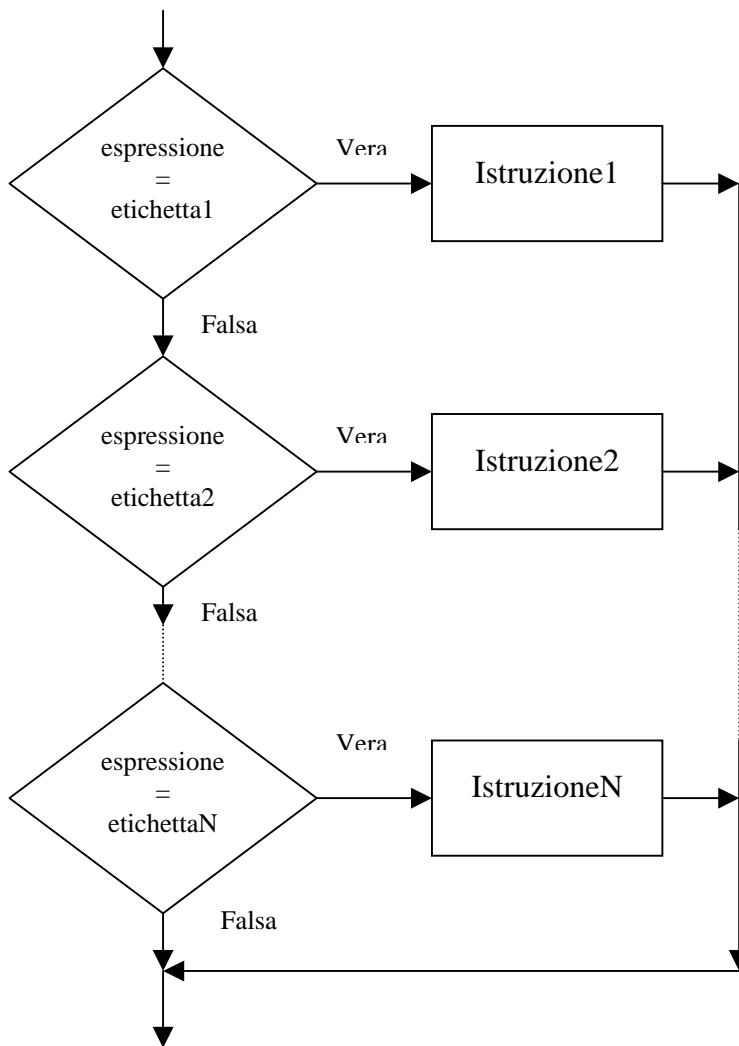
```

if a > b then
    a := a - b
else
    b := b - a;
  
```

N.B.: L'istruzione prima di **else** non richiede il ";" in quanto il sistema riconosce dove termina l'istruzione dalla presenza della parola riservata "else".

Selezione multipla (struttura case)

Quando deve essere operata una scelta fra vari casi viene utilizzata la struttura **case** in sostituzione di istruzioni **if...then** multiple. Se una delle etichette case risulta uguale al valore che assume *espressione* viene eseguita la corrispondente istruzione. Le etichette devono essere dello stesso tipo di espressione.



La sintassi della struttura è:

```
case espressione of
  etichetta1: Istruzione1;
  etichetta2: Istruzione2;
  .....
  etichettaN: IstruzioneN;
```

oppure:

```
case espressione of
  etichetta1: Istruzione1;
  etichetta2: Istruzione2;
  .....
  etichettaN: IstruzioneN;
else
  IstruzioneM
end;
```

Nel caso che non si verifichi l'uguaglianza per nessuna delle etichette il programma prosegue senza che sia stata effettuata alcuna scelta, se però è presente l'**else**, viene eseguita l'istruzione ad essa riferita.

Esempio:

case k of

1, 3, 5, 7, 9 : **write**('numero dispari');

2, 4, 6, 8 : **write**('numero pari')

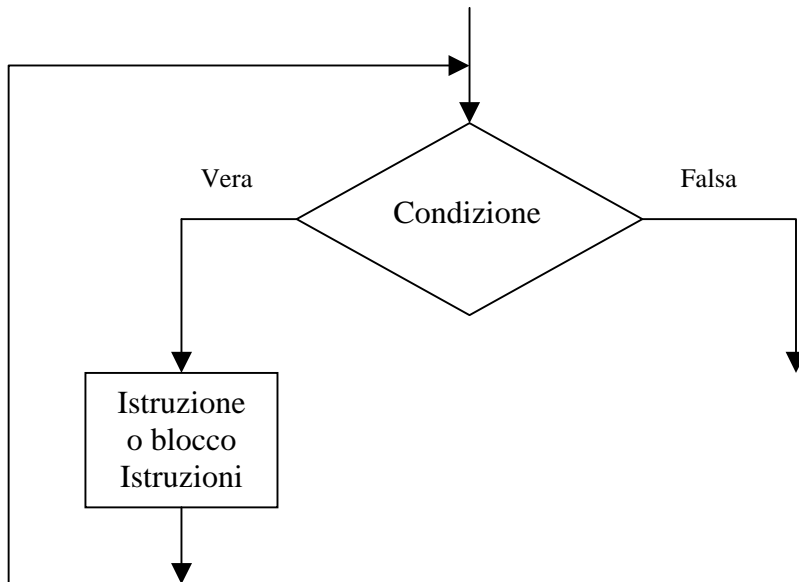
end;

Strutture iterative

Le strutture iterative permettono di ripetere l'esecuzione di una determinata parte di programma controllando che una certa condizione posta si mantenga vera (*while... do*) o di ripetere il gruppo di

istruzioni fino a quando la condizione non diviene vera (*repeat... until*) o per un numero assegnato di volte (*for...*).

La struttura while... do



La sintassi è:

```
while Condizione do
    Istruzione;
(* oppure begin
    blocco istruzioni
end;      *)
```

esempio:

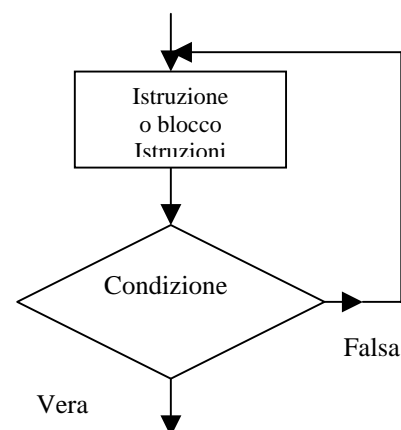
```
while r >= y do
begin
    q := q+1;
    r := r - y
end;
```

La condizione viene esaminata all'inizio del ciclo, quindi se essa non è vera prima di entrarvi, l'istruzione (o il blocco di istruzioni) che segue il **do** non viene svolta neanche una volta. Se l'istruzione (o le istruzioni) che seguono **do** non modificano prima o poi la condizione rendendola falsa, non si esce mai dal ciclo.

La struttura repeat... until

La sintassi della struttura è:

```
repeat Istruzione
(* oppure
    blocco Istruzioni    *)
until condizione;
```



In questo caso la condizione viene esaminata alla fine del ciclo, e quindi esso viene eseguito sempre almeno una volta. L'uscita avviene questa volta quando la condizione diviene *vera* (quindi essa deve essere *falsa* per eseguire ancora il ciclo): anche in questo caso l'istruzione (o le istruzioni) che segue il **repeat** deve avere la possibilità di modificare la condizione, rendendola *vera*, per uscire dal ciclo.

Esempio:

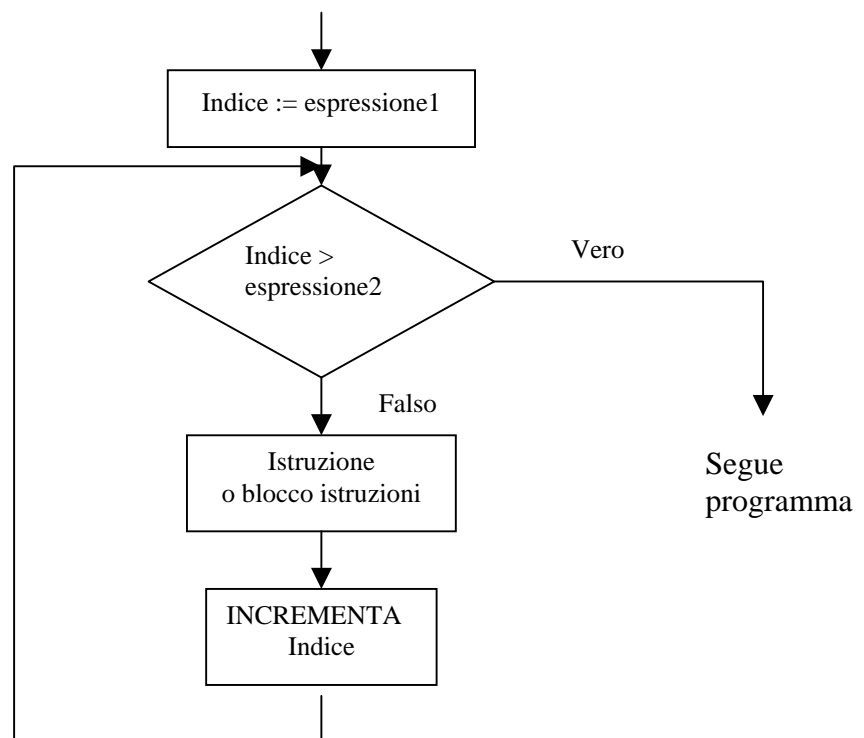
repeat

$h := h + 1/n;$

$n := n - 1$

until $n = 0;$

La struttura for...



La sintassi è:

for Indice := espressione1 **to** espressione2 **do**

Istruzione;

(* oppure

begin blocco Istruzioni

end; *)

All'indice viene assegnato un valore iniziale (espressione1); dopodiché il ciclo viene ripetuto tante volte fino a quando il valore dell'indice incrementato automaticamente non raggiunge il valore finale imposto dall'espressione2. Se debbono essere eseguite più istruzioni queste vanno racchiuse tra **begin...end**.

E' prevista anche la seguente sintassi:

for Indice := espressione1 **downto** espressione2 **do**

Istruzione;

(* oppure

begin blocco Istruzioni

end; *)

In questo caso invece di essere incrementata, la variabile indice viene decrementata automaticamente.

Esempi:

for i := 1 **to** n **do**

begin

a := a + i;

write('La variabile a vale: ', a)

end;

for i := m **downto** n **do**

begin

a := a + i;

write('La variabile a vale: ', a)

end;

TIPI DI DATI STRUTTURATI

I dati finora utilizzati sono tutti semplici e predefiniti. E' possibile utilizzare dati più complessi detti strutturati e suddivisi in:

tipo **array**, tipo **record**, tipo **set** e tipo **file**.

Tipo Array

Il tipo **array** definisce una struttura dati i cui componenti appartengono allo stesso tipo; ciascun componente è selezionabile tramite un indice.

Se per esempio si volessero memorizzare un certo numero di valori di tensione presenti ai capi di un condensatore in tempi successivi potremmo utilizzare un array.

Se l'identificatore unico delle tensioni venisse chiamato *Vc*, per distinguere un valore dall'altro potrà essere impiegato un indice, per esempio *ind*, che variando da 0 ad un determinato valore *n*, permetterà di identificare le varie tensioni. Se quindi l'identificatore del vettore è *Vc[ind]*, (l'indice viene posto tra parentesi quadre), i vari valori di tensione saranno:

$$Vc[1], Vc[2], Vc[3], \dots, Vc[n]$$

La dichiarazione degli array viene effettuata in Pascal nel modo seguente:

```
var   nome_array: array [tipo_indice] of tipo_elementi;
```

Esempio:

```
var   Vc: array [1..20] of real;
```

Gli array possono essere dichiarati come tipi; in tal caso bisogna, per referenziarli, dichiarare anche una variabile di tipo array seguendo la seguente sintassi:

```
type  nome_array = array [tipo_indice] of tipo_elementi;
```

```
var   var_array: nome_array;
```

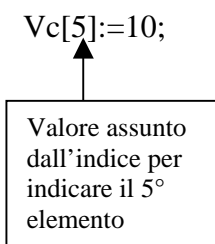
Esempio:

```
type  tensione = array [1..20] of real;
```

```
var   Vc: tensione;
```

definisco cioè un tipo *tensione* come array di venti elementi reali e poi nella sezione di definizione delle variabili definisco la variabile *Vc* del tipo *tensione*.

Per assegnare un valore ad un componente dell'array, bisogna selezionarlo assegnando il valore opportuno all'indice; ad esempio per assegnare al 5° elemento del vettore *Vc* il valore di tensione pari a 10:



Rappresentazione grafica di un array di interi:

1	2	3	4	5	6	7	8	9	10
127	78	23	452	34	71	39	6	26	11

127 è il valore contenuto dall'array e situato nella sua prima locazione

E' possibile dichiarare anche un array a più dimensioni; ad esempio un array a due dimensioni di interi, che viene detto anche *matrice* si dichiara in questo modo:

var num: **array** [1..4, 1..3] **of** integer;

che possiamo immaginare come una tabella con 4 righe e 3 colonne:

Esempio di tabella:

123	45	32
-10	12	5
8	9	30
27	34	128

Supponendo di aver inizializzato gli elementi dell'array con i valori dati nella tabella, gli elementi dell'array sono accessibili con un identificatore formato dal nome del vettore e due indici, uno per le righe ed uno per le colonne: num[r, c].

Esempio:

num[1, 1] corrisponde al valore 123,

num[4, 2] corrisponde al valore 34.

Quindi con il primo indice vengono indicate le righe e con il secondo le colonne. Una struttura dati di questo tipo viene utilizzata quando si lavora con tabelle.

Ecco un esempio di programma che calcola la traccia di una matrice definita come un array bidimensionale. Per traccia di una matrice con lo stesso numero di righe e di colonne si intende la somma degli elementi della diagonale che hanno lo stesso indice di riga e di colonna:

```

Program Traccia_Matrice;
uses CRT;

Type indice=1..3;
    matrice= array[indice,indice] of integer;

const max = 3;

var mt: matrice;
    i, j, tr: integer;

begin
    clrscr;

    (* Lettura degli elementi di una matrice *)
    writeln('Lettura matrice 3 x 3:');
    writeln;
    for i:=1 to max do
        for j:=1 to max do
            begin
                write('mt[');
                write(i,j,']: ');
                readln(mt[i,j]);
            end;
        tr:= mt[1,1]+ mt[2,2]+ mt[3,3];

    (* Sezione di stampa *)
    writeln;
    writeln('La traccia della matrice: ');
    for i:=1 to max do
        begin
            writeln;
            for j:=1 to max do
                write(' ',mt[i,j]);
            end;
            writeln('
                ');
        end;
    writeln('
                ');
    writeln('
                ');
    readln
end.

```

Tipo Record

Un tipo **record** è una collezione di componenti (detti **campi**) di tipo qualsiasi.

Nome	Cognome	Indirizzo	Tel
------	---------	-----------	-----

Questa figura è una rappresentazione di un record.

La dichiarazione di un record è effettuata nel modo seguente:

type

nome_record = **record**

nome_campo1: tipo;

```

        nome_campo2: tipo;
        .....;
        nome_campoN: tipo;
    end;
var
    var_record: nome_record;

```

Nella sezione **var** definisco una variabile del tipo record sopra definito; verrà riservata un'area di memoria opportuna, la cui grandezza dipenderà dal tipo dei campi che formano il record.

Esempio:

```

type data = record
    giorno: 1..31;
    mese: 1..12;
    anno: 1900.. 2100;
end;

```

```

var d: data;

```

Per accedere ai vari campi del record bisogna ricorrere alla seguente sintassi

```

    var_record.nome_campo

```

separando il nome della variabile di tipo record da quello del campo interessato con un punto.

Esempio:

```

d.giorno := 15;
d.mese := 10;
d.anno := 1998;

```

Il campo di un record può essere rappresentato da un record come nel seguente esempio:

```

type persona = record
    nome: string[20];
    cognome: string[20];
    indirizzo: string;
    nascita: data; (* tipo record definito in precedenza *)
    sesso: (M, F)
end;

```

Poi definisco una variabile di tipo persona:

```

var cliente: persona;

```

Per inizializzare i vari campi:

```
cliente.nome := 'Paolo';
cliente.nascita.anno := 1930;
```

Per accedere ai vari campi posso usare l'istruzione **with** con la seguente sintassi:

```
with nome_record do
  begin ...
    (* le istruzioni che vado a scrivere in questa sezione possono accedere direttamente
    ai campi del record *)
  ... end;
```

Esempio:

```
... writeln(persona.nome);
  writeln(persona.cognome);
  writeln(persona.indirizzo.via);
  writeln(persona.indirizzo.numero); ...
```

Questo può essere scritto in questo modo:

```
... with persona do
  begin
    writeln(nome);
    writeln(cognome);
    with indirizzo do
      begin
        writeln(via);
        writeln(indirizzo);
      end;
  end; ...
```

Posso definire strutture dati sempre più complesse come array di record; ad esempio posso pensare ad un programma di gestione in cui una struttura dati può essere costituita da 1000 clienti che definirò in questo modo:

```
type clienti = array [1.. 1000] of persona;
var reparti: array[1..10] of clienti;
```

In questo esempio ho dieci reparti di una data attività commerciale con 1000 potenziali clienti.

Viene ora presentato un programma che utilizza questo tipo di strutture dati:

```
program media_voti;
uses crt;
type alunni=record
  nome:string[20];
  cognome:string[20];
```

```

    eta:integer;
    med: array[1..5] of real;
end;
var
    mat: array[1..5] of string;
    a: array[1..5] of array[1..4] of integer;
    c: array[1..5] of alunni;
    i,j, k: integer;

begin
clrscr;
(* Sezione di lettura dei dati *)
for i:=1 to 5 do
begin
    write('scrivi il nome del ',i,'° alunno ');
    readln(c[i].nome);
    write('scrivi il suo cognome ');
    readln(c[i].cognome);
    write('la sua età ');
    readln(c[i].eta);

end;
for i:=1 to 5 do
begin
    write('scrivi la ',i,'° materia ');
    readln(mat[i]);

end;
clrscr;
for k:=1 to 5 do
begin
writel('==== Allievo ', c[k].cognome, ' ',c[k].nome,' =====');
for j:=1 to 5 do
for i:=1 to 4 do
begin
    write('scrivi il ',i,'° voto di ',mat[j],' ');
    readln(a[j,i]);

end;
clrscr; (* calcolo della media dei voti per ciascuna materia*)
for i:=1 to 5 do
c[k].med[i]:=(a[i,1]+a[i,2]+a[i,3]+a[i,4])/4;
end;

for i:=1 to 5 do (* Sezione di stampa *)
begin
    writeln('la media voti di ',c[i].nome,' ',c[i].cognome,' di anni',c[i].eta);
    for j:=1 to 5 do
        writeln(' in ',mat[j],' è di ',c[i].med[j]:4:2);
end;

readln;
end.

```

Questo programma dopo aver richiesto i voti riportati in ogni materia calcola la media e visualizza il risultato.

La possibilità di lavorare con dati strutturati consente di affrontare e risolvere problemi di una certa complessità e di mantenere un'organizzazione di dati spesso molto vicina alla situazione reale che rappresenta.

Tipo set

Un dato strutturato di tipo set è formato da tutti i possibili sottoinsiemi di un tipo base ordinale (cioè di tipo enumerativo o subrange), così definito in quanto il tipo è costituito da un insieme ordinato e ad ogni elemento dell'insieme è associato un valore intero a partire da zero. Il tipo base non può avere più di 256 elementi (con estremi 0 e 255), quindi sono ammissibili come tipo base il tipo byte, char, boolean e i tipi di dati ordinali realizzati dal programmatore e precedentemente dichiarati come tipo (ad esempio Giorni_della_Settimana).

La dichiarazione viene fatta nel modo seguente:

```
type <identificatore_del_tipo> set of <tipo base>;
```

Esempio:

```
type colore_primario = (rosso, giallo, blu); (* definizione del tipo enumerativo colore_primario *)  
type colore = set of colore_primario;  
var c: colore;
```

La variabile c può assumere un insieme di valori di tipo colore primario, cioè:

```
[ ] insieme vuoto  
[rosso] [giallo] [blu]  
[rosso, giallo] [rosso, blu] [giallo, blu]  
[rosso, giallo, blu]
```

Gli **operatori** applicabili a variabili di tipo set sono:

```
+   Unione  
*   Intersezione  
-   Differenza  
=, <>, <=, >=  
in   Appartenenza
```

➤ **Operatore Unione (+)**

Consente di inserire elementi in un insieme rappresentato da una variabile di tipo set

```
Var a, b: colore;
```

```
a:= [rosso, giallo];    b:= [rosso, blu];
```

```
b := a + b;    (* il risultato della espressione è [rosso, giallo, blu] *)
```

➤ **Operatore Differenza (-)**

Consente di eliminare elementi in un insieme rappresentato da una variabile di tipo set

`a := [rosso, blu]` `b := [rosso, giallo]`

`a - b` produce il valore `[blu]`

N.B.: `a := a + [x]` non altera a se x è già contenuto in a

`a := a - [x]` non altera a se x non è contenuto in a

➤ **Operatore Intersezione (*)**

`a := [rosso, blu]` `b := [rosso, giallo]`

`a * b` produce il valore `[rosso]`

➤ **Operatore Inclusione (IN)**

E' usato per verificare la presenza di un elemento in un insieme.

Esempio:

```
type insieme_di_caratteri = set of char;
```

```
var vocali: insieme_di_caratteri;
```

```
    c: char; ...
```

```
vocali := ['a', 'e', 'i', 'o', 'u'];
```

```
readln( c );
```

```
if c in vocali then ...
```

```
    else ...
```

è equivalente a:

```
if (c = 'a') or (c = 'e') or (c = 'i') or (c = 'o') or (c = 'u') then ...
```

➤ **Operatori >= <= =**

- `a >= b` restituisce valore vero (true) se ciascun elemento di b appartiene ad a e falso altrimenti (a include b)
- `a <= b` restituisce valore vero (true) se ciascun elemento di a appartiene ad b e falso altrimenti (b include a)
- `a = b` restituisce valore vero (true) se i due insiemi coincidono e falso altrimenti.

Programmazione: il problema dello scambio.

Spesso per risolvere un problema si potrebbe presentare la necessità di operare uno scambio fra i contenuti di due variabili. Nella pratica questo problema è simile a quello di voler scambiare il contenuto di due bicchieri; per eseguire tale scambio bisogna ricorrere ad un terzo bicchiere che servirà solo da appoggio. E' intuitivo che l'utilizzo di un terzo bicchiere è indispensabile per non perdere o mischiare il contenuto dei due bicchieri. Consideriamo il caso di due variabili intere *num1* e *num2* che contengono rispettivamente i valori 6 ed 8. Per scambiare i valori, come per il caso dei bicchieri, abbiamo bisogno di una variabile temporanea dello stesso tipo che possiamo chiamare *temp*. I passi da compiere sono i seguenti:

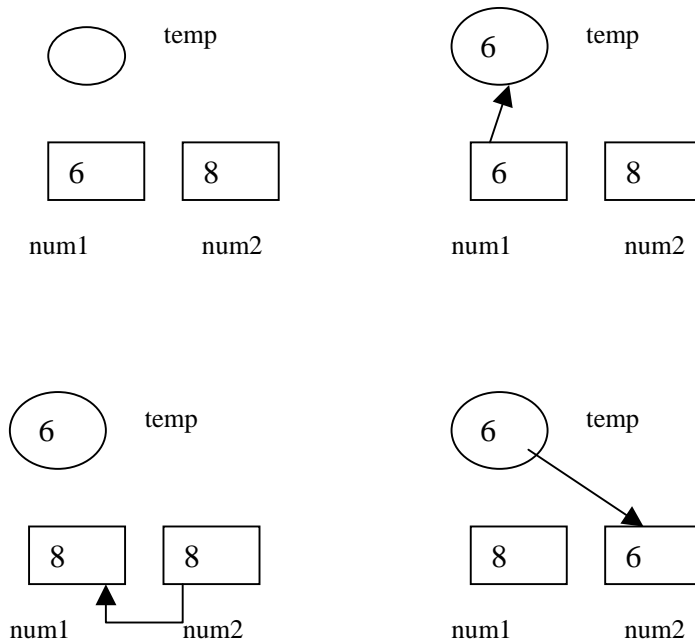
- Ricopiamo in *temp* il valore di *num1*.
- Mettiamo in *num1* il valore di *num2*.
- Facciamo assumere a *num2* il valore contenuto nella variabile *temp*.

A questo punto *num1* vale 8 e *num2* vale 6. In codice Pascal bisogna scrivere:

```
temp:= num1;
```

```
num1:= num2;
```

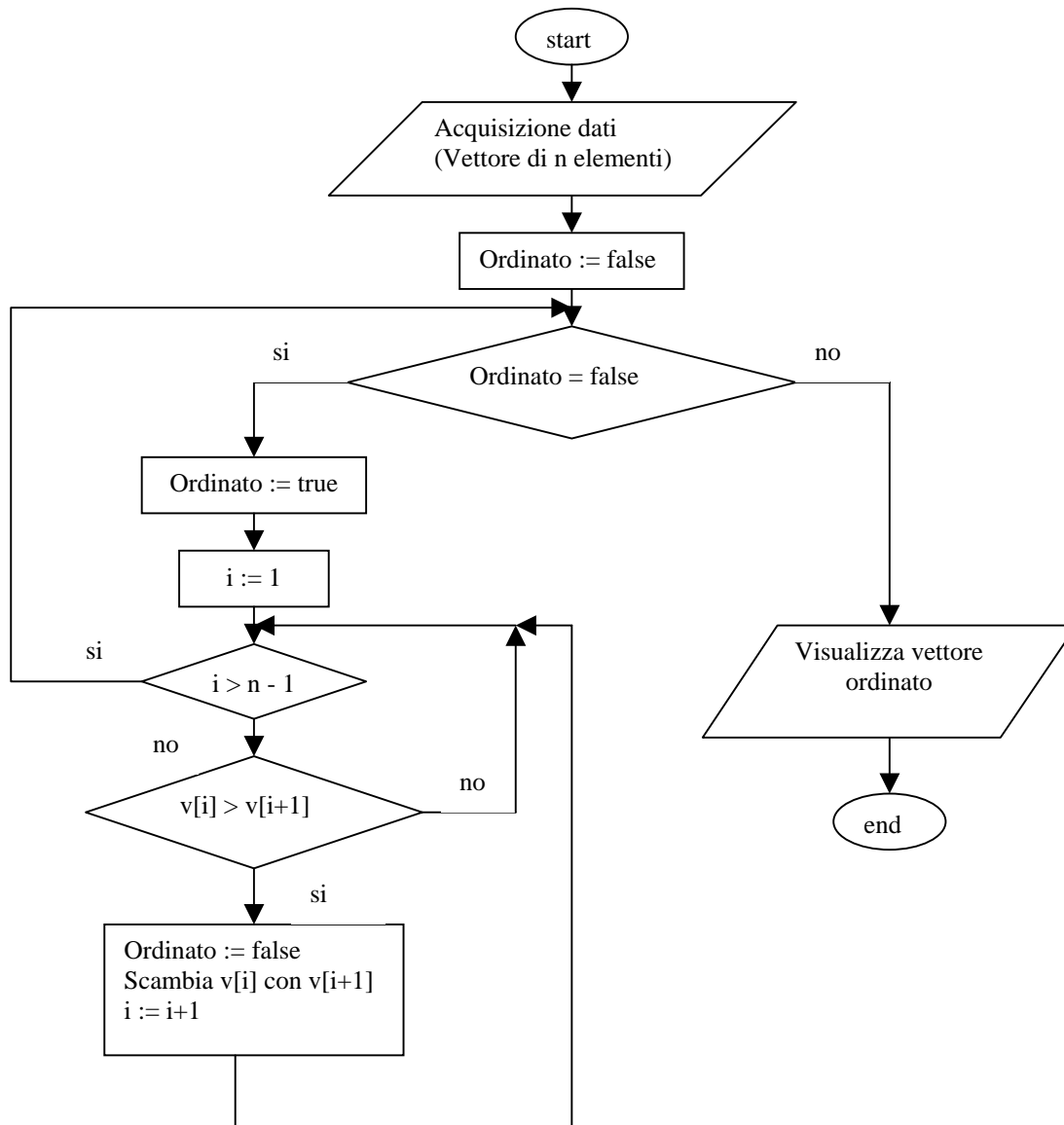
```
num2:= temp;
```



Algoritmi di ordinamento

Rappresentano una classe di algoritmi che, data una sequenza di valori, dopo un certo numero di passi forniscono come risultato la stessa sequenza, ma ordinata in senso crescente o decrescente a seconda della necessità. Una delle strutture dati che meglio si presta per risolvere questo problema è l'array, dal momento che gli elementi in esso contenuti sono individuati tramite un indice che indica la loro posizione.

Semplice algoritmo di ordinamento



L'ordinamento avviene in senso crescente. I passi che esso compie sono i seguenti:

- L'array viene scandito a partire dalla prima fino alla penultima posizione e viene eseguito un confronto di volta in volta fra due elementi adiacenti.
- Se l'elemento attualmente letto ha valore maggiore del successivo, allora si esegue uno scambio fra i contenuti delle rispettive locazioni della struttura.
- I passi precedenti vengono ripetuti finché non si verifica una scansione all'interno della quale non viene effettuato alcuno scambio (ciò significa quindi che l'array è ordinato).

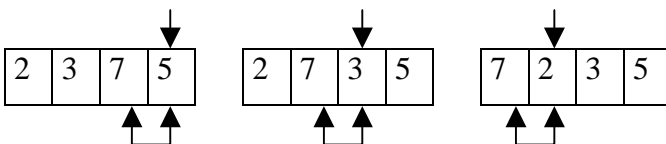
Quindi ad ogni scansione dell'array l'algoritmo tende a sportare i valori più grandi verso il fondo dell'array e, viceversa, i valori più piccoli verso la parte iniziale.

Esempio pratico: (ordinamento decrescente)

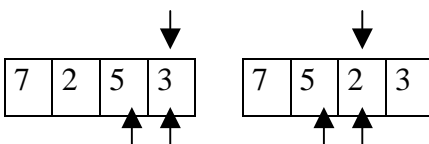
Array iniziale

2	3	5	7
---	---	---	---

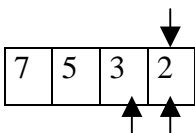
1° ciclo i = 1



2° ciclo i = 2



3° ciclo i = 3



Array ordinato

7	5	3	2
---	---	---	---

La realizzazione dell'algoritmo in linguaggio Pascal è la seguente:

```

program ordina;
const n=10;
var a:array [1..n] of integer;
    i,temp: integer;
    ordinato: boolean;

begin
  for i:=1 to n do
    begin
      write('scrivi a[' ,i, ']: ');
      readln(a[i]);
    end;

    ordinato := false; {inizializzazione per entrare nel while}
    while (ordinato = false) do
      begin
        ordinato := true; (* si presuppone che il vettore sia ordinato *)
        for i:=1 to n-1 do
          if a[i]> a[i+1] then
            begin
              temp:=a[i];
              a[i]:=a[i+1];
              a[i+1]:=temp;
              ordinato := false; {ipotesi di vettore non ancora ordinato}
            end
          end;
        end;

        (* stampa per la verifica di ordinamento avvenuto *)
        writeln('il vettore ordinato è:');
        for i:=1 to n do
          write(a[i], ' ');
        readln;
      end.

```

Il ciclo esterno realizzato attraverso un *while* controllerà, di volta in volta, se l'array è ordinato oppure no. Quindi tale ciclo controlla e determina l'esecuzione del ciclo più interno, il quale si occuperà effettivamente di operare il controllo dei valori letti e di eseguire i vari scambi. La guardia del ciclo *while* effettuerà un controllo su una variabile di tipo booleano, che indicherà lo stato di ordinamento dell'array. Questa variabile viene chiamata **ordinato** ed assumerà valore *false* se ci sono elementi da scambiare (il che significa: "l'array non è ordinato, scandisci di nuovo"), mentre assumerà valore *true* quando risulterà nell'ordine desiderato. In tal caso si uscirà dal ciclo *while* terminando l'algoritmo. Quindi il ciclo più esterno verrà eseguito finché l'array non è ordinato. Il ciclo più interno costituito da un *for* servirà per la lettura dei valori contenuti nel vettore e l'indice del ciclo viene fatto variare dalla prima fino alla penultima posizione, in quanto il confronto (e l'eventuale scambio) viene fatto fra l'elemento corrispondente all'indice corrente ed il successivo. Quando l'elemento attuale è maggiore del successivo, si deve operare uno scambio tra le due locazioni dell'array.

Questo algoritmo ordina perfettamente un vettore di elementi ma non è certamente il più efficiente dal punto di vista di tempo di esecuzione. Un algoritmo più efficiente è il *Bubble Sort*.

Bubble Sort

Il nome, che significa ordinamento a bolle, è dovuto al fatto che, ad ogni passo del ciclo esterno, il valore più grande tra quelli ancora da ordinare viene spostato verso la posizione finale dell'array nel caso di ordinamento crescente, mentre il valore più basso viene posizionato definitivamente nella posizione con indice più basso fra quelle disponibili.

Per realizzare questo algoritmo si utilizzano due cicli *for* annidati. Il ciclo più interno si occupa del confronto fra elementi adiacenti e dell'eventuale scambio, mentre il ciclo esterno serve solo per limitare il numero di elementi su cui operare il successivo ordinamento. Quindi l'indice del *for* esterno deve andare dalla prima alla penultima locazione dell'array. Il ciclo interno viene eseguito in verso opposto, a partire dall'ultima posizione del vettore fino alla posizione immediatamente successiva all'attuale valore assunto dal ciclo esterno. La prima volta il ciclo interno opera il confronto fra gli elementi contenuti fra l'ultima e la seconda posizione. Quindi il valore più basso verrà posizionato nella prima locazione in quanto si tratta di ordinamento crescente. Il ciclo interno successivo sarà effettuato sempre a partire dall'ultima posizione, ma fino ad arrivare al terzo valore, in modo che l'elemento minore fra quelli analizzati occuperà la seconda posizione dell'array. Si procederà in questo modo fino al termine.

I passi fondamentali dell'algoritmo sono:

- Si esegue una scansione a partire dal primo fino al penultimo elemento del vettore (utilizziamo una variabile *i* come indice del ciclo)
- Si esegue una scansione interna al primo ciclo a partire dall'ultimo elemento fino all'elemento *i+1*.
- Se l'elemento *j* attualmente letto è minore del precedente *j-1* (maggiore nel caso di ordinamento decrescente), allora si effettua uno scambio.

Vediamo la realizzazione in linguaggio Pascal:

```
program bubble_sort;
const n=10;
var a:array [1..n] of integer;
    i,j,temp: integer;

begin
  for i:=1 to n do
    begin
      write('scrivi a[' ,i, ']: ');
      readln(a[i]);
    end;

    (* ordinamento *)
    for i:= 1 to n-1 do
      for j:= n downto i+1 do
        if a[j] < a[j-1] then
          begin
            temp:=a[j];
```

```

        a[j]:=a[j-1];
        a[j-1]:=temp;
    end;

    (* stampa per la verifica di ordinamento avvenuto *)
    writeln('il vettore ordinato è:');
    for i:=1 to n do
        write(a[i], ' ');
    readln;
    end.

```

Il Bubble Sort si basa sul presupposto che, ad ogni ciclo, si riesca a mettere nella corretta posizione almeno un elemento dell'array. Questo fatto permette di limitare il numero di passi compiuti dal ciclo più interno. Ad esempio per un array di 5 elementi, la prima volta si eseguiranno 4 passi, dopo 3 passi, poi 2 passi, ed infine 1 passo, per un totale di 10 passi. Quindi questo algoritmo risulta più efficiente di quello visto in precedenza.

Vediamo l'applicazione di questo algoritmo all'ordinamento di un vettore di record in base la valore di un campo, nel nostro caso nome di tipo string:

```

program bubble_sort;
const n=10;
type persona = record
    nome: string[30];
    tel: string[15];
end;

var a:array [1..n] of persona;
    i,j: integer;
    temp:persona;

begin
    for i:=1 to n do
        begin
            write('scrivi nome[' ,i, ']: ');
            readln(a[i].nome);
            write('scrivi il numero di telefono: ');
            readln(a[i].tel);
        end;

        (* ordinamento *)
        for i:= 1 to n-1 do
            for j:= n downto i+1 do
                if a[j].nome < a[j-1].nome then
                    begin
                        temp:=a[j];
                        a[j]:=a[j-1];
                        a[j-1]:=temp;
                    end;
            end;

        (* stampa per la verifica di ordinamento avvenuto *)
        writeln('il vettore ordinato è:');
        for i:=1 to n do
            writeln(a[i].nome, ' --> ', 'con numero di telefono: ',a[i].tel);
        readln;
        end.

```

Funzioni e procedure

Spesso può capitare, scrivendo un programma, di dover eseguire lo stesso gruppo di istruzioni in punti diversi del programma. Si sente quindi la necessità di raggruppare tale sequenza di istruzioni in una forma di “modulo autonomo” scritto una sola volta, un vero e proprio sottoprogramma che può essere richiamato in più punti del mio programma. E’ quello che viene fatto con la calcolatrice dotata di tasti funzione quando ad esempio premo il tasto che calcola il quadrato di un numero. Non disponendo di questa funzione bisogna digitare il numero due volte per moltiplicarlo per se stesso, invece di digitarlo una sola volta seguito dal tasto funzione. Questo semplice esempio serve per capire che anche nella programmazione conviene ricorrere a funzioni che raggruppano più operazioni di vario tipo e che posso richiamare scrivendo il nome della funzione e potendo di volta in volta passare gli operandi come input e la funzione ritornerà poi il risultato.

In Pascal esistono due tipi di sottoprogrammi, che anche se simili, hanno struttura e compiti diversi: le *funzioni* e le *procedure*.

Una funzione è un sottoprogramma che restituisce, una volta eseguito, un solo valore al programma chiamante. Quando invece un sottoprogramma svolge dei compiti senza restituire alcun valore al programma chiamante si parla di procedura. Normalmente quando si chiama una funzione o una procedura bisogna passare dei dati su cui operare. Questi dati vengono detti *parametri*.

Una funzione è strutturata all’interno come un generico programma: può contenere dichiarazioni di costanti, di variabili, definizioni di tipi di dati, eventuali procedure e funzioni interne e poi il corpo principale. Le variabili dichiarate all’interno sono dette **locali** e non sono visibili esternamente.

Struttura di una funzione:

```
function nome_funzione(lista parametri): tipo_funzione;  
    const ...; (* definizione di eventuali costanti locali *)  
    type ...; (* definizione dei tipi locali *)  
    var ...; (* dichiarazione delle variabili locali *)  
  
    < eventuali funzioni o procedure interne >  
  
begin  
    istruzioni della funzione;  
  
end;
```

Vediamo adesso l’esempio di una funzione che chiameremo *somma* e che sarà in grado di sommare due numeri interi. E’ necessario specificare il tipo del risultato che nel nostro caso sarà un intero e

che la funzione restituirà al programma chiamante. Si è già detto che la funzione necessita di un input, rappresentato dalla lista dei *parametri formali*. Questi sono a tutti gli effetti delle variabili e come tali vanno dichiarate attribuendo loro un nome, e specificando il loro tipo di appartenenza. Vediamo l'esempio di come può essere scritta la funzione somma:

```
function somma (num1, num2: integer): integer;
begin
    somma := num1 + num2;
end;
```

All'atto di una qualsiasi chiamata, si deve specificare il valore che in quel determinato momento i parametri formali devono assumere: sarà proprio con questi valori che la funzione andrà a lavorare. Ad esempio nel programma principale si può scrivere:

```
risultato := somma(2, 3);
```

In questo caso vogliamo che la somma venga calcolata fra i numeri due e tre, e che tale valore venga attribuito alla variabile *risultato*. Al parametro num1 viene assegnato il valore 2 e al parametro num2 il valore 3. Questi numeri che sono scritti fra parentesi tonde sono detti *parametri attuali*.

Il corpo della funzione deve essere delimitato dalle parole chiave *begin* e *end*. La funzione deve restituire, al termine dell'esecuzione, un valore attraverso un certo meccanismo, che in Pascal corrisponde con l'assegnare alla funzione il corrispondente valore. Inoltre l'assegnamento deve rappresentare l'ultima istruzione del blocco:

```
nome_funzione := valore_da_restituire;
```

Vediamo come esempio la funzione che calcola il logaritmo con base rappresentata da un qualunque numero intero non presente fra le funzioni matematiche implementate in Pascal:

```
program logaritmo;
uses crt;
var xx:real;
    bb:integer;

function log(x:real;b:integer):real;
begin
    if x=0 then
        writeln('Errore...')
    else
        log:= ln(x)/ln(b);
end;
```

```

        end;

begin
writeln('scrivi il numero: ');
readln(xx);
writeln('scrivi la base: ');
readln(bb);
writeln(' Il logaritmo di ',xx:4:2,' in base ',bb,' è: ',log(xx,bb):4:4);
readln;
end.

```

La chiamata di funzione avviene nel blocco principale all'interno dell'istruzione di stampa, come un qualsiasi comando, e i valori introdotti da tastiera del numero di cui voglio effettuare il calcolo del logaritmo e la base vengono passati come parametri alla funzione (parametri attuali).

Nell'esempio non è stato necessario utilizzare altre variabili oltre i parametri formali della funzione. Spesso però è necessario utilizzare delle variabili locali per la funzione, che vengono usate solo al suo interno. Una variabile locale nasce all'atto della chiamata della funzione e muore al termine dell'esecuzione della stessa funzione.

Vediamo a questo punto due esempi di programmi che utilizzano funzioni. Il primo programma calcola il fattoriale di un numero intero:

```

program Calcolo_fattoriale;
var n:integer;

function fat(k:integer):longint;
var i,t: integer;
begin
t:=1;
if k <> 0 then
for i:=k downto 1 do
    t:=t*i;
fat:= t;
end;

begin
write('Scrivi un numero intero: ');
readln(n);
writeln('Il fattoriale di ',n,' è: ',fat(n));
readln;
end.

```

Il secondo esempio usa più funzioni ed effettua il calcolo dell'area di alcune figure geometriche, le funzioni sono molto semplici e sono usate come comando per il calcolo di una determinata area nel blocco principale. Alle funzioni vengono passati i parametri letti da tastiera.

```

program Calcolo_aree;
uses CRT;
var scelta: char;
    base, altez, raggio: real;

```

```

function AreaRett(b,h:real):real;
begin
AreaRett:=b*h;
end;

function AreaTrian(b,h:real):real;
begin
AreaTrian:= b*h/2;
end;

function AreaCerch(r:real):real;
begin
AreaCerch:= PI*r*r;
end;

begin
clrscr;
repeat
write('Scegli una figura: --c (CERCHIO), r (RETT), t(TRIAN), e(USCIRE)--> ');
readln(scelta);
case scelta of
't':begin
write('scrivi la base: ');
readln(base);
write('scrivi l'altezza: ');
readln(altez);
writeln('L'area del Triangolo è: ',AreaTrian(base,altez):4:2);
end;
'r':begin
write('scrivi la base: ');
readln(base);
write('scrivi l'altezza: ');
readln(altez);
writeln('L'area del Rettangolo è: ',AreaRett(base,altez):4:2);
end;
'c':begin
write('Scrivi il raggio: ');
readln(raggio);
writeln('L'area del Cerchio è: ',AreaCerch(raggio):4:2);
end;
end;
until scelta = 'e';
end.

```

Le procedure

Il Pascal distingue fra sottoprogrammi che ritornano un risultato e sottoprogrammi che non ritornano alcun valore. L'insieme di istruzioni che non deve necessariamente fornire in uscita un valore viene indicato con il termine procedura.

La struttura di una procedura è:

procedure nome_procedura(lista parametri):

const ...; (* definizione di eventuali costanti locali *)

type ...; (* definizione dei tipi locali *)

var ...; (* dichiarazione delle variabili locali *)

< eventuali funzioni o procedure interne >

begin

istruzioni della funzione;

end;

Per fare un esempio, l'algoritmo che scambia il contenuto di due variabili, già visto in precedenza non può essere scritto sotto forma di funzione ma sarà necessariamente sviluppato sotto forma di procedura:

```
procedure scambia(var a, b:integer);
```

```
var temp:integer;
```

```
begin
```

```
temp := a;
```

```
a:= b;
```

```
b:= temp;
```

```
end;
```

Alla procedura vengono passati due parametri il cui contenuto deve essere scambiato. Nel nostro caso, questi parametri vengono dichiarati come variabili attraverso la parola chiave *var* che precede la lista di variabili, ovvero vengono passati per *riferimento* (cioè per indirizzo). Si fa questo per permettere alla procedura di modificare il loro contenuto. Quando i parametri vengono passati per valore, cioè senza dichiararli come variabili, il valore non viene alterato dal blocco di istruzioni. Quindi bisogna passare i parametri per riferimento ogni qualvolta la procedura deve effettuare delle modifiche ai valori contenuti, in caso contrario si passano per valore.

Nel nostro esempio la variabile *temp* va dichiarata locale alla procedura e non è visibile al di fuori di essa.

Vediamo un esempio di programma che utilizza delle procedure e che esegue lo scambio delle posizioni di un array prendendo il primo elemento dell'array e scambiandolo con l'ultimo, il secondo con il penultimo e così via.

```
program scambia_posiz;  
uses CRT;  
const dimmax=100;  
type vettore= array[1..dimmax] of integer;  
var i,nelem:integer;  
    vet:vettore;
```

```

procedure lettura(var v:vettore; var n:integer);
begin
  write('numero di elementi: ');
  readln(n);
  for i:=1 to n do
  begin
    write('v[' ,i, ']: ');
    readln(v[i]);
  end;
end;

procedure scambio(var v:vettore; n:integer);
var k,j,temp,h:integer;
begin
  j:=n;
  h:= n DIV 2;
  for i:=1 to h do
  begin
    temp:=v[i];
    v[i]:=v[j];
    v[j]:=temp;
    j:=j-1;
  end;
end;

begin
  clrscr;
  nelem:=0;
  lettura(vet,nelem);
  scambio(vet,nelem);
  for i:=1 to nelem do
  writeln('v[' ,i, ']: ',vet[i]);
  readln
end.

```

Ci deve essere corrispondenza tra parametri effettivi e parametri formali: cioè il numero di parametri effettivi deve essere uguale al numero di parametri formali; ciascun parametro effettivo deve corrispondere al parametro formale che occupa la stessa posizione e deve essere dello stesso tipo.

Visibilità delle variabili

Una variabile dichiarata locale ad una procedura esiste solo durante l'esecuzione della procedura (essa è creata in entrata alla procedura e distrutta in uscita). Ne segue che:

- Una variabile locale ad una procedura non può essere usata dal programma che ha chiamato la procedura.
- Non c'è nessuna relazione tra i valori delle variabili create da successive esecuzioni della stessa procedura. Il valore di una variabile locale è sempre non definito quando si entra in una procedura.

- La memoria necessaria per le variabili locali di una procedura viene acquisita solo quando si entra nella procedura; diventa riutilizzabile al termine della procedura.

Program M;

...

procedure P;

...

procedure A;

...

end; {A}

end; {P}

procedure Q;

...

procedure B;

...

end; {B}

procedure C;

...

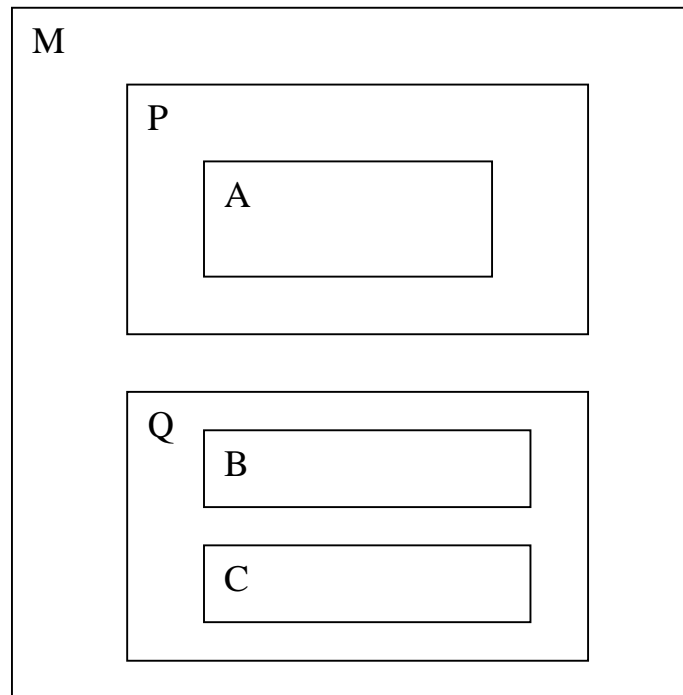
end; {C}

end; {Q}

begin

<blocco principale>

end. {M}



Identificatori definiti nel blocco di:	Sono accessibili in:
M	M, P, A, Q, B, C
P	P, A
A	A
Q	Q, B, C
B	B
C	C

Vediamo come può essere realizzato l'ordinamento di un vettore di interi già visto in precedenza utilizzando le procedure. L'algoritmo di ordinamento è quello denominato Bubble Sort:

```
program bubble_sort;
uses CRT;
const n=10;
type vettore = array [1..n] of integer;

var v:vettore;

procedure letturaDati(var a:vettore);
var i:integer;
begin
  for i:=1 to n do
    begin
      write('scrivi a[' ,i, ']: ');
      readln(a[i]);
    end;
end;

procedure scambia(var h,k:integer);
var temp:integer;
begin
  temp:= h;
  h:= k;
  k:= temp;
end;

procedure ordinaVet(var a:vettore);
var i,j:integer;
begin
  for i:= 1 to n-1 do
    for j:= n downto i+1 do
      if a[j] < a[j-1] then
        scambia(a[j-1], a[j]);
    end;
end;

procedure stampaVet(a:vettore);
var i:integer;
begin
  for i:=1 to n do
    write(a[i], ' ');
end;

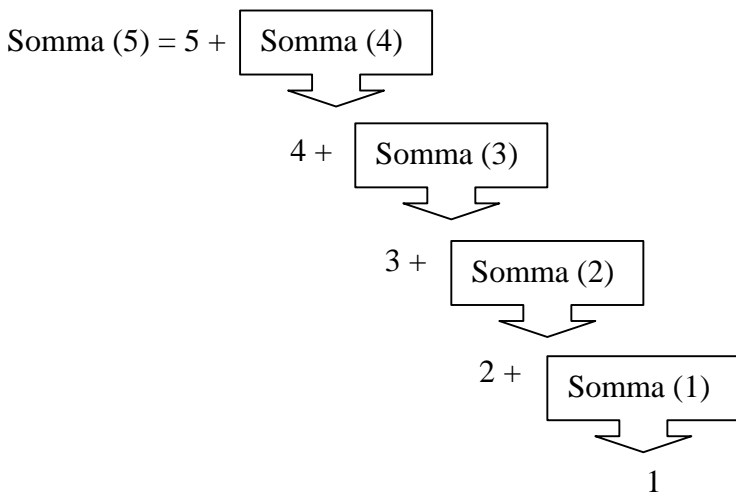
(* Programma principale *)
begin
  clrscr;
  letturaDati(v);
  ordinaVet(v);
  writeln('Il vettore ordinato è:');
  stampaVet(v);
  readln;
end.
```

La ricorsione

Una procedura o una funzione può invocare anche se stessa: si ha in questo caso una procedura o funzione ricorsiva. La ricorsione è una tecnica di programmazione che si impiega per risolvere quei problemi che possono essere suddivisi in una serie di sottoproblemi fra loro equivalenti e più semplici da trattare. Vediamo come primo semplice esempio la somma dei primi n numeri naturali.

Se $n = 4$ in tal caso abbiamo $somma = 4 + 3 + 2 + 1 + 0$.

Il problema è caratterizzato dalla presenza di un *caso base* che si ha quando n vale 0 o 1 ed in questo caso una funzione chiamata somma restituirà il valore di n e di un *caso generale* nel quale la funzione restituisce il valore di n sommato alla funzione *somma di $n - 1$* .



La funzione è espressa in forma ricorsiva: se il numero dato rientra nel caso base la funzione restituisce il numero stesso, altrimenti se n è maggiore di 1 si esegue una chiamata ricorsiva alla funzione stessa, passando come parametro attuale il numero dato diminuito di una unità. Quindi questa operazione viene eseguita automaticamente finché il parametro passato non è uguale ad 1.

La funzione assume la forma seguente:

```
function somma(N: integer): integer;  
begin  
    if N <= 1 then somma := N;    {caso base}  
    else  
        somma := N + somma( N-1) {chiamata ricorsiva}  
end;
```

Ogni funzione ricorsiva ha una struttura di questo tipo: si deve individuare il caso base che deve essere controllato per primo; dopo si deve individuare il caso generale, il quale deve essere eseguito ogni qualvolta non sia verificato il caso base.

Scrivendo un programma in modo ricorsivo dobbiamo sempre fare in modo che la funzione termini prima o poi, ovvero si deve sempre stabilire il *caso base*, altrimenti l'istruzione di chiamata ricorsiva potrebbe essere eseguita infinite volte, e di conseguenza il programma potrebbe non terminare mai.

Un secondo esempio di programma ricorsivo è il calcolo del fattoriale di un numero intero positivo. Questo calcolo si presta bene ad essere realizzato da un algoritmo ricorsivo in quanto per fattoriale di un numero intero n si intende il prodotto di tutti gli interi fino ad n sapendo che il fattoriale di 0 è definito uguale ad 1 (questo rappresenta il caso base). L'algoritmo è quindi analogo al precedente, cambia solo l'operazione: in questo caso è il prodotto mentre nell'esempio precedente era la somma. Vediamo la realizzazione in Pascal dell'algoritmo:

```
program fattoriale_ricorsivo;

var n:integer;

function fat(i:integer):longint; { funzione ricorsiva }
begin
    if i= 0 then fat:=1
    else
        fat:=fat(i-1)*i;
end;

begin
write('scrivi il numero : ');
readln(n);
writeln('Il fattoriale di n è: ',fat(n));
readln;
end.
```

La realizzazione ricorsiva del fattoriale è stata realizzata seguendo fedelmente la definizione naturale del concetto di fattoriale. Questa è una delle caratteristiche principali della ricorsione, cioè viene tradotto in algoritmo il modo naturale di pensare umano. Quindi una procedura ricorsiva è spesso più semplice da capire e da verificare. Le procedure ricorsive tendono a dividere il problema in problemi più semplici e questo spesso aumenta l'efficienza dell'algoritmo. L'uso della ricorsione non determina una maggiore velocità di esecuzione del programma, anzi, spesso si ottiene un peggioramento di velocità a causa dalla continua creazione ed eliminazione delle variabili locali allocate nell'area di memoria detta *stack* ad ogni chiamata di funzione.

Vediamo ora un algoritmo ricorsivo di ordinamento molto usato che sfrutta la caratteristica fondamentale della ricorsione della suddivisione di un problema in sottoproblemi più semplici ma simili. Si tratta dell'ordinamento quicksort.

Ordinamento quicksort

L'algoritmo si basa sulla divisione di un vettore da ordinare in due vettori più piccoli ed sull'ordinamento di questi ultimi dividendoli a loro volta in vettori sempre più piccoli. Si va avanti con la ricorsione fino a quando non si ottiene un vettore formato da un solo elemento automaticamente ordinato.

Rimane il problema di come dividere il vettore in due vettori più piccoli e di come rimetterli insieme una volta ordinati. Un'idea semplice è quella di scegliere un elemento a caso, che chiamiamo pivot, e di confrontare tutti gli elementi con questo. Mettiamo tutti gli elementi minori in un vettore e quelli maggiori in un altro. Si può, però, ricorrere ad altre soluzioni. Vediamo una realizzazione di ordinamento di un vettore di interi, dopo analizzeremo l'algoritmo adoperato:

```
Program quick_sort;  
const N = 10;  
type lista = array[1..N] of integer;  
  
var vet : lista;  
    i:integer;  
  
procedure scambia( var a, b:integer);  
var temp: integer;  
begin  
    temp:=a;  
    a:= b;  
    b:= temp;  
end;  
  
procedure quick(sin, des: integer; var v:lista);  
var i,j,media:integer;  
begin  
media:= (v[sin]+v[des])div 2;  
i:=sin;  
j:=des;  
  
repeat  
while v[i] < media do i:=i+1;  
while v[j] > media do j:=j-1;  
if i<=j then begin  
scambia(v[i],v[j]);  
i:=i+1;  
j:=j-1;  
end  
until i>j;  
  
if sin <j then quick(sin,j,v);  
if i<des then quick(i,des,v);  
end;          { Fine procedura quick }  
  
begin  
for i:= 1 to N do begin  
    write ('scrivi il ',i,'° numero: ');  
    readln(vet[i]);  
end;
```

```

quick(1, N, vet);
writeln('Il vettore ordinato è:');
for i:= 1 to N do writeln(vet[i]);
readln;
end.

```

In questo caso si è scelto come elemento pivot il valore medio fra il primo e l'ultimo elemento del vettore che viene passato come parametro alla procedura *quick*, cioè viene passato l'indirizzo del vettore (passaggio per riferimento) e due valori che rappresentano le posizioni estreme del vettore da ordinare. In questo modo non viene usata memoria aggiuntiva per costruire il vettore ordinato, ma questo viene ricavato all'interno del vettore originale.

L'algoritmo implementato dalla procedura ricorsiva *quick* è il calcolo del pivot come media fra gli elementi corrispondenti alle due posizioni estreme che indicano la parte di vettore da ordinare e che sono passate per valore (sin, des); lo scambio degli elementi in modo da posizionarli in due vettori: in uno compariranno gli elementi di valore minore del pivot e nell'altro quelli di valore maggiore.

L'algoritmo rieseguirà le stesse operazioni (chiamata ricorsiva) sui nuovi sottovettori; vengono passate sempre le posizioni limite del nuovo vettore da ordinare e così via fino a quando il sottovettore sarà formato da due soli elementi che vengono posizionati uno nella posizione di sinistra (quello di valore inferiore al pivot) e l'altro nella posizione di destra e non ci sarà più una chiamata ricorsiva in quanto il nuovo sottovettore è formato da un solo elemento.

Vediamo in pratica l'ordinamento di un vettore di dieci elementi letti da tastiera:

$v = [13, 1, 5, 72, 35, 91, 27, 63, 9, 36]$

La tabella seguente mostra gli elementi del vettore nella rispettiva posizione dopo ogni chiamata della procedura ricorsiva *quick*:

Chiamate ricorsive	Pivot	1	2	3	4	5	6	7	8	9	10
quick(1,10,v)	24	13	1	5	9	35	91	27	63	72	36
quick(1,4,v)	11	9	1	5	13	35	91	27	63	72	36
quick(1,3,v)	7	5	1	9	13	35	91	27	63	72	36
quick(1,2,v)	3	1	5	9	13	35	91	27	63	72	36
quick(5,10,v)	35	1	5	9	13	27	91	35	63	72	36
quick(6,10,v)	63	1	5	9	13	27	36	35	63	72	91
quick(6,7,v)	35	1	5	9	13	27	35	36	63	72	91
quick(9,10,v)	81	1	5	9	13	27	35	36	63	72	91

Vediamo come può essere cambiato il programma precedente per poter ordinare un vettore di stringhe:

```
Program quick_sort;
const N = 10;
type lista = array[1..N] of string;

var vet : lista;
    i:integer;

procedure scambia( var a, b:string);
var temp: string;
begin
    temp:=a;
    a:= b;
    b:= temp;
end;

procedure quick(sin, des: integer; var v:lista);
var i,j,media:integer;
    pivot:string;
begin
    media:= (sin+des)div 2;
    pivot:=v[media];
    i:=sin;
    j:=des;

    repeat
        while v[i] < pivot do i:=i+1;
        while v[j] > pivot do j:=j-1;
        if i<=j then begin
            scambia(v[i],v[j]);
            i:=i+1;
            j:=j-1;
        end
    until i>j;

    if sin <j then quick(sin,j,v);
    if i<des then quick(i,des,v);
end;
{ Fine procedura quick }

begin
for i:= 1 to N do begin
    write ('scrivi il ',i,'° nome: ');
    readln(vet[i]);
end;

quick(1, N, vet);
writeln('Il vettore ordinato è:');
for i:= 1 to N do writeln(vet[i]);
readln;
end.
```

Funzioni e procedure per il trattamento di stringhe

Il Turbo Pascal mette a disposizione come già visto il tipo *string*, non disponibile nel Pascal Standard dove le stringhe vengono trattate come array di caratteri (ad esempio la definizione di una stringa di 20 caratteri risulta: *type stringa = packed array [1..20] of char*).

Sono inoltre disponibili una serie di funzioni e procedure che operano sui dati di tipo stringa.

Funzioni:

Length (str)

Con str di tipo string. Ritorna un intero che rappresenta la lunghezza della stringa str.

Esempio:

```
program Prova_Length;
var nome : string;
    num_carat:integer;
begin
    write('Scrivi il nome: ');
    readln(nome);
    num_carat:=Length(nome);
    writeln('Il nome è di ',num_carat,' caratteri');
    readln;
end.
```

Concat (str1, str2, ...)

Unisce le stringhe str1, str2, ... Non possono essere superati i 255 caratteri (dimensione massima di una stringa in Turbo Pascal). La stessa operazione eseguita da Concat si ottiene con l'operatore "+".

Esempio: stringa := str1+ str2;)

Esempio:

```
Program prova_Concat;
var nome, cognome, spazio, stringa: string;
begin
    write('Scrivi il nome: ');
    readln(nome);
    write('Scrivi il cognome: ');
    readln(cognome);
    spazio:=' ';
    stringa:=concat(cognome,spazio,nome);
    (* stringa contiene il cognome, poi uno spazio ed il nome *)
    writeln(stringa);
    readln;
end.
```

Copy (str, i, num)

Con *str* di tipo string ed *i, num* di tipo integer. Restituisce una sottostringa formata da *num* caratteri prelevati da *str* a partire dall'*i*-esimo carattere.

Esempio:

```
Program prova_Copy;
var stringa, nome:string;
begin
    nome:='Bianca Neri';
    stringa:=Copy(nome,8,4);
    writeln(stringa);
    (* Stampa sul video solo il Cognome 'Neri' *)
    readln;
end.
```

Pos (sot_str, str)

Con *sot_str, str* di tipo string. Se la sottostringa *sot_str* non viene trovata all'interno della stringa *str*, restituisce zero, altrimenti un numero intero che indica la posizione del primo carattere della sottostringa all'interno della stringa.

Esempio:

```
Program prova_Pos;
var nome:string;
    posiz:integer;
begin
    nome:='Bianca Neri';
    posiz:=Pos('Neri',nome);
    writeln('La posizione è: ',posiz);
    (* Stampa sul video la posizione di 'Neri', cioè 8 *)
    readln;
end.
```

Procedure:

Str (num, str_num)

Con *num* di tipo integer o real; *str_num* di tipo string. Il dato *num* di tipo intero o reale, viene convertito nella stringa *str_num*.

Esempio:

```
Program prova_Str;
var tel:longint;
    pref,n_tel,stringa:string;
begin
    tel:=222222;
    pref:='030/';
    Str(tel,stringa);
    n_tel:=Concat(pref,stringa);
    (* n_tel contiene la stringa '030/222222' *)
end.
```

```

        writeln('Il telefono è: ',n_tel);
        readln
end.

```

Si può fissare la larghezza della stringa *str_num* (a seconda del valore di *largh*) e il numero di decimali da convertire (valore di *dec*) come con la procedura **write**:

Str (num: largh: dec, str_num)

Con *num* di tipo integer o real; *str_num*: string; *largh,dec* di tipo integer.

Esempio:

```

Program prova_Str;
var tensione:real;
    stringa1,stringa2:string;
begin
    tensione:=22.2222;
    stringa1:='Vc = ';
    Str(tensione :5: 2, stringa2);
    stringa1:=Concat(stringa1,stringa2);
    writeln('La tensione è ',stringa1);
    readln
end.

```

Val (str_n, num, cod)

Con *str_n* di tipo string; *num*: integer o real; *cod* : integer. La stringa *str_n*, se contiene informazioni numeriche adatte, viene convertita nel numero *num*. La procedura ritorna in *cod* il valore zero se la conversione ha operato correttamente.

Esempio:

```

Program prova_Val;
var tel:longint;
    cod_p,cod_t,pref:integer;
    st_pref,st_tel,stringa:string;
begin
    stringa:='030/222222';
    st_pref:=Copy(stringa,1,3);
    st_tel:=Copy(stringa,5,6);
    Val(st_pref,pref,cod_p);
    Val(st_tel,tel,cod_t);
    (*Verifica l'esito della conversione e stampa il risultato *)
    if cod_p = 0 then
        writeln('Il prefisso è: ',pref);
    if cod_t = 0 then
        writeln('Il telefono è: ',tel);
    readln
end.

```

Questa procedura viene usata in particolar modo quando vengono acquisiti dati numerici sotto forma di stringhe e sui quali occorre effettuare delle operazioni.

Delete (str, i, num)

Con *str*: string; *i*, *num*: integer. Cancella dalla stringa *str*, *num* caratteri a partire dal carattere *i*-esimo.

```
Program prova_Delete;
var nome:string;
begin
  nome:='Bianca Neri';
  Delete(nome,1,7);
  writeln(nome);
  (* Stampa sul video solo il Cognome 'Neri' *)
  readln;
end.
```

Insert (str1, str2, i)

Con *str1*, *str2*: string; *i*: integer. Il parametro *str2* è passato per riferimento, gli altri per valore. La stringa *str1* viene inserita in *str2* a partire dalla posizione *i*. Se *str2*, dopo l'inserimento, supera i 255 caratteri, essa verrà troncata.

Esempio:

```
program prova_Insert;
var stringa1,stringa2:string;
begin
  stringa1:='Gigi è andato a scuola';
  stringa2:='non ';
  writeln('Prima di Insert:-> ',stringa1);
  Insert(stringa2,stringa1,6);
  writeln('Dopo Insert:-> ',stringa1);
  readln;
end.
```

Procedure di uscita e di terminazione

exit

La procedura *exit* permette di uscire dal sottoprogramma in uso. Nel caso che la procedura *exit* venga utilizzata nel programma principale, il programma si arresta.

continue

E' una procedura che viene sfruttata per ritornare all'inizio del ciclo. Uno degli usi più frequenti è quello di verificare l'input dell'utente.

break

Lo scopo della procedura *break* è quello di terminare l'esecuzione di un ciclo. Quando il programma esegue *break*, esce dal ciclo e prosegue con la successiva istruzione del programma.

Esempio:

```
var
  S: String;
begin
  while True do
    begin
      ReadLn(S);
      if S = '' then Break;
      WriteLn(S);
    end;
  (* Legge delle stringhe e le visualizza sullo schermo, termina quando viene
  fornita una stringa vuota*)
end.
```

In Pascal è possibile effettuare l'istruzione di salto attraverso l'istruzione **goto** (salta a) caratteristica di alcuni linguaggi. Ma in questo linguaggio è assolutamente inutile l'utilizzo di tale istruzione in quanto vi sono strutture di controllo estremamente efficienti e flessibili; questa istruzione oltre a rendere meno efficiente l'esecuzione del programma, lo rende meno leggibile ed inoltre non consente una programmazione strutturata che è una delle caratteristiche innovative del linguaggio Pascal.

Il tipo file

Si tratta della struttura dati più conosciuta da ogni utente di computer. Compito fondamentale di un sistema di elaborazione è quello di raggruppare logicamente e gestire le informazioni. Solitamente le informazioni vengono raggruppate in archivi; il file non è altro che un termine informatico per indicare l'archivio. Le informazioni vengono inserite nel file che viene poi memorizzato nella memoria di massa del calcolatore (hard disk). I file dell'hard disk vengono gestiti da una componente del sistema operativo che prende il nome di file management system, il quale semplifica il lavoro dell'utente raggruppando i file in directory (o cartelle) che possono contenere anche altre directory in una struttura ad albero, occupandosi dell'intera gestione del sistema. L'utente si limita a indicare dove registrare o reperire determinati dati.

Un file è, quindi, una struttura costituita da elementi dello stesso tipo, che può contenere svariati tipi di dati. Un semplice tipo di file è rappresentato dal file di testo che viene creato con un programma di scrittura e contiene una sequenza di caratteri (che possono essere codificati secondo diversi codici: ASCII, ANSI, ecc.). I file generati dall'Edit del DOS sono costituiti da caratteri ASCII. Esistono anche file più complessi come file di record in cui ogni singolo dato è costituito da un record suddiviso in campi.

Il file può essere organizzato o come una sequenza di dati o casualmente. L'accesso è anche di due tipi:

- ☞ *Accesso sequenziale*: gli elementi del file vengono esaminati sequenzialmente a partire dal primo fino a raggiungere l'informazione desiderata.
- ☞ *Accesso diretto*: consente di individuare direttamente l'elemento desiderato. Sui file organizzati in maniera casuale è consentito solo questo tipo di accesso. Questo tipo d'accesso non è consentito su tutti i tipi di dispositivi: sui nastri magnetici, ad esempio, occorre scorrere tutti gli elementi.

In Pascal il file rappresenta un tipo di dati: è una collezione di elementi tutti dello stesso tipo, sia esso semplice o strutturato. La sua lunghezza non è prestabilita e contiene un carattere speciale (*end-of-file*) che indica la fine del file. Il file non può però essere un contenitore di file. Al file deve essere associato un file del sistema operativo dove vengono registrate o prelevate le informazioni (queste risiedono quindi sulla memoria di massa).

Nei vecchi compilatori bisognava specificare dopo il nome del programma le parole input ed output che rappresentano il file di ingresso, da dove vengono lette in modo standard le informazioni e quello dove vengono scritte (tastiera e schermo):

program esempio (input, output);

Il file è, dunque, un tipo di dati e deve essere dichiarato all'inizio di ogni programma dove viene utilizzato. Ad esempio un file contenente numeri reali deve essere dichiarato in questo modo:

```
☞ type file_di_esempio = file of real;
```

dove `file_di_esempio` rappresenta il nome del file.

Se volessimo costruire un'agenda telefonica che memorizza i dati su memoria di massa potremmo dichiarare una struttura contenente record con il campo nome ed il campo numero telefonico:

```
☞ type persona = record
    nome: string[30];
    tel: string[20];
end;
agenda = file of persona;
```

I file di tipo testo, contenenti caratteri sono già predefiniti come tipo *text* e quindi posso dichiarare una variabile di tipo file di testo nel modo seguente:

```
☞ var f1 : text;
```

`f1` è un file che contiene sequenze di caratteri.

In Pascal il file è sequenziale in quanto la lettura e la scrittura dei dati avviene selezionando un elemento dopo l'altro in successione. Ad ogni file viene associata automaticamente una variabile buffer di dimensione uguale al singolo elemento che compone il file, nella quale viene memorizzato di volta in volta il valore di ogni elemento letto dal file. Questa variabile è identificata dal nome del file seguito dal carattere “^”; esempio:

```
☞ agenda^
```

Vediamo le operazioni che possono essere effettuate sui file (con *f* indichiamo il nome di un file Pascal):

Assign (f, 'c:\agenda.dat') Assegna al nome del file fisico con cui verrà registrato su disco (nel nostro caso 'c:\agenda.dat') il nome del file Pascal (*f*). Se non si indica il percorso il file viene cercato o creato nella directory corrente. Il secondo parametro può essere anche una variabile di tipo stringa che contiene il nome del file fisico.

Eof (f) questa funzione restituisce il valore *True* se il puntatore è sul carattere di terminazione del file (`end_of_file`), *False* altrimenti.

Rewrite (f) crea ed apre il file *f* registrandolo sul disco con il nome assegnato dall'istruzione `assign`. Se il file esiste, il vecchio contenuto viene perso (riscrittura del file).

Reset (f) apre un file già esistente che può essere letto o modificato.

Append (f) apre solo un file di tipo testo (text) per aggiungere nuovi dati a partire dalla fine del file (indicata dal carattere ASCII 26, `ctrl+Z`).

Close (f) chiude il file precedentemente aperto per la lettura o la scrittura. Nel caso di scrittura questo comando svuota il buffer registrando i dati contenuti in esso e non ancora salvati su disco.

Write (f, elemento) scrive il valore della variabile *elemento* nel file *f*. Naturalmente il file deve essere un contenitore di dati dello stesso tipo di *elemento*. Il puntatore del file viene incrementato di una posizione.

Read (f, elemento) legge un dato dal file *f* e lo memorizza nella variabile *elemento*, incrementando il puntatore del file di una posizione.

Filesize (f) restituisce il numero di dati presenti nel file.

Seek(f, numero) posiziona il puntatore sul dato del file *f* la cui posizione è espressa da *numero* di tipo intero.

Seek(f, filesize(f)) posiziona il puntatore alla fine del file *f*. Questa funzione serve per aggiungere in coda nuovi dati ad un file già esistente.

Vediamo adesso un esempio di lettura di un file di testo (comando `type` del DOS):

```
program lettura_file_testo;
uses crt;

type f=text;

var fp:f;
    nome:string;
    carat:char;

begin
```

```

clrscr;
writeln('file da leggere');
readln(nome);
assign(fp,nome);
reset(fp);
while not eof(fp) do
begin
read(fp,carat);
write(carat);
end;
close(fp);
end.

```

Esempio di un programma Agenda che consente la lettura o la modifica di un file esistente:

```

program scrivi_leggi_file;
uses crt;
type persona=record
  nome:string[15];
  tel:string[12];
end;
fp=file of persona;

var f:fp;
    pr:persona;
    ch,risp:char;
    v: array[1..30] of persona;
    i,h,k:integer;
begin
clrscr;
assign(f, 'agend.dat');
repeat
clrscr;
gotoxy(10,1);
writeln('----- AGENDA -----');writeln;
writeln('Premi <l> per leggere ed <s> per scrivere nel file <e> per uscire dal
        programma');
gotoxy(4,6);
write('Scelta --> ');
readln(ch);
case ch of
's': begin
  clrscr;
  write('Vuoi inserire una persona? (s / n ): ');
  readln(risp);
  i:= 0;
  if risp = 's' then
  repeat
  write('Scrivi il nome: ');
  readln(v[i].nome);
  write('Scrivi il numero di telefono: ');
  readln(v[i].tel);
  write('Vuoi ancora inserire una persona? (s / n ): ');
  readln(risp);
  i:=i+1;
  until risp = 'n';
  write('Vuoi sovrascrivere il file? (s / n): ');
  readln(risp);
  if risp = 's' then
  begin
    rewrite(f);
    for h:= 0 to i-1 do

```

```

        write(f,v[h]);
        close(f);
    end
else
    begin
        reset(f);
        seek(f,filesize(f));
        for h:= 0 to i-1 do
            write(f,v[h]);
        close(f);
        end;
    end;
end;
'l': begin
    clrscr;
    writeln('----- Agenda -----'); writeln;
    reset(f);
    while not eof(f) do
        begin
            read(f,pr);
            writeln(pr.nome);
            writeln(pr.tel);
        end;
        writeln('Premi Invio per tornare al menu ');
        readln;
        close(f);
    end;
end;
until ch= 'e';
end.

```

Questo programma molto semplice nella struttura consente la lettura del file esistente, l'aggiornamento o la completa riscrittura del file (apertura con l'istruzione *rewrite*) a seconda del carattere letto da tastiera.

I file del Sistema Operativo presentano degli attributi (ad esempio file di sola lettura o file nascosto). Questi attributi sono definiti in un modo che dipende dal particolare sistema. Sono specificati in una variabile di tipo word il cui valore dipende dal particolare sistema operativo. In DOS si ha:

ReadOnly	\$01	(file di sola lettura)
Hidden	\$02	(file nascosto)
SysFile	\$04	(file di sistema)
VolumeID	\$08	
Directory	\$10	
Archive	\$20	(file archivio)
AnyFile	\$3F	

Nella *unit dos* vi sono delle procedure per leggere o settare gli attributi di un dato file:

GetFAttr (var F; var Attr: word); per leggere l'attributo
SetFAttr (var F; Attr: word); per settare un attributo

Vediamo un esempio:

```

program esempio_GetFAttr;
uses Dos;

```

```

var
  F: file;
  Attr: Word;
begin
  { Legge il file come parametro dalla linea di comando }
  Assign(F, ParamStr(1));
  GetFAttr(F, Attr);
  Writeln(ParamStr(1));
  if DosError <> 0 then
    Writeln('DOS error code = ', DosError)
  else
    begin
      Write('Attributo = ', Attr);
      { Determina l'attributo utilizzando le costanti }
      if Attr and ReadOnly <> 0 then      { In Windows: faReadOnly }
        Writeln('File di sola lettura');
      if Attr and Hidden <> 0 then        { In Windows: faHidden }
        Writeln('File nascosto');
      if Attr and SysFile <> 0 then       { In Windows: faSysFile }
        Writeln('File di sistema');
      if Attr and VolumeID <> 0 then      { In Windows: faVolumeID }
        Writeln('Volume ID');
      if Attr and Directory <> 0 then     { In Windows: faDirectory }
        Writeln(' Nome di Directory ');
      if Attr and Archive <> 0 then       { In Windows: faArchive }
        Writeln('File archivio');
    end; { else }
end.

```

Direttive

Per la gestione degli errori di I/O (ingresso/uscita) è necessario indicare al compilatore come comportarsi in corrispondenza di istruzioni di I/O (come, ad esempio, la lettura di un file). Questa indicazione viene detta direttiva. La direttiva per il controllo automatico degli errori di I/O è **{SI+}** oppure **{SI-}**.

Con **{SI+}** se si verifica un errore di I/O l'esecuzione del programma viene terminata immediatamente con una segnalazione di errore a runtime.

Con **{SI-}** si elimina la terminazione automatica e la gestione degli errori di I/O è interamente affidata al programmatore.

La funzione **IOResult** ritorna il codice di errore dell'ultima istruzione di I/O eseguita. Se il codice ritornato è 0, allora non si è verificato alcun errore.

Nel caso di apertura di un file in lettura può essere conveniente disabilitare il controllo automatico degli errori e confrontare il valore ritornato da **IOResult**.

Vediamo l'esempio del programma `copia_file` (realizza il comando `copy` del DOS):

```

program copia_file;
var sorg, dest: file of byte;
    elemento: byte;
begin
if ParamCount<>2 then
  begin

```

```

writeln('Specificare il file sorgente e destinazione');
writeln('Scrivi:');
writeln('copia <file sorgente> <file destinazione>');
exit;
end;
assign(sorg, ParamStr(1));
assign(dest, ParamStr(2));
{$I-}
reset(sorg);
if IOResult = 0 then
begin
begin
{$I+}
rewrite(dest);
while not eof(sorg) do
begin
read(sorg,elemento);
write(dest,elemento);
end;
close(sorg);
close(dest);
end
else
writeln('Impossibile aprire il file sorgente');
end.

```

Se il file sorgente non esiste il programma visualizza il messaggio di errore e termina. Il controllo di errore viene poi riabilitato per consentire la terminazione automatica in caso di errori di scrittura o lettura all'interno del ciclo *while*. I file sono stati definiti come contenitori di byte in quanto il programma deve poter copiare qualunque tipo di file. Il nome del file sorgente e quello di destinazione vengono passati come parametri. Sono state utilizzate due funzioni predefinite:

ParamCount ritorna un intero che indica il numero dei parametri

ParamStr(numero_posizione) ritorna una stringa che corrisponde al parametro individuato dal numero intero contenuto in *numero_posizione*.

Tramite ParamCount si verifica che i parametri passati siano due (file sorgente e destinazione), mentre la funzione ParamStr ritorna la stringa corrispondente al primo o al secondo parametro a seconda del valore passato alla funzione come parametro attuale (1 o 2). Il programma reso eseguibile si lancia dal Sistema Operativo come il comando copy:

copia file_sorgente file_destinazione.

Le Unit

Il Pascal permette di definire unità compilabili separatamente allo scopo di realizzare una programmazione modulare che rende più flessibile la scrittura di programmi complessi. Quindi il progetto viene diviso in più moduli. Questo tipo di programmazione insieme ad un'implementazione ad oggetti è divenuta uno standard (sistema di sviluppo Delphi della Borland). Mentre il programma principale è delimitato dalle parole chiave *program* e *end.*, le varie unità, contenenti funzioni e procedure che vengono utilizzate nel programma principale, sono delimitate dalle parole chiave *unit* e *end.* È possibile utilizzare in un'unità comandi implementati in altri moduli attraverso la parola chiave *uses*.

Nel programma principale, nella sezione delimitata da *uses*, occorre specificare tutti i moduli utilizzati. La unit si divide in due parti:

l'**interfaccia** che contiene le dichiarazioni globali a tutta la unit (costanti e tipi di dati) e le definizioni di procedure e funzioni da esportare;

l'**implementazione**, che contiene i corpi delle procedure e funzioni sopra dichiarate.

Vediamo la struttura di una unità:

unit <nome_unità>;

interface

```
const < dichiarazione delle costanti >  
type < dichiarazione dei tipi di dati >  
function f1 ( parametri: tipo): tipo;  
procedure p1 ( parametri: tipo);
```

implementation

```
function f1 ( parametri: tipo): tipo;  
    var < dichiarazione variabili locali >;  
    begin  
        < corpo della funzione >;  
    end;
```

```
procedure p1 ( parametri: tipo);  
    var < dichiarazione variabili locali >;  
    begin  
        < corpo della procedura >;  
    end;
```

end.

Il codice delle unit è salvato in file con estensione .PAS e compilato con “destinazione” settata su “disk”. Il file compilato, che rappresenta un modulo del programma principale ha estensione .TPU.

Il Turbo Pascal presenta alcuni moduli standard già implementati e compilati, ad esempio:

CRT per la gestione del video in modalità carattere

Graph per la gestione del video in modalità grafica

Queste unità saranno analizzate in seguito. Vediamo adesso un esempio di programma che utilizza una unità appositamente costruita. Il programma esegue operazioni sui numeri complessi utilizzando funzioni di una *unit* chiamata *complex*:

```
unit complex;
interface
  type
    C = record
      Re, Im: real
    end;

  function Sign(a: real): integer;
  function parteRe(a: C): real;
  function parteIm(a: C): real;
  function Arg(a: C): real;
  function AbsC(a: C): real;
  (* Le funzioni ritornano la parte reale, la parte immaginaria,
   l'argomento e il modulo di un numero complesso *)
  procedure setR(m, n: real; var z: C);
  procedure setPR(ro, fi: real; var z: C);
  (* Costruttori dei numeri complessi a partire dalla rappresentazione
   cartesiana o polare *)
  procedure SumC(a, b : C; var z: C); (*somma tra due numeri complessi *)
  procedure DiffC(a, b : C; var z: C); (*sottrazione tra due numeri complessi*)
  procedure MultC(a, b : C; var z: C); (*moltiplicazione tra due numeri
   complessi *)
  procedure Conj(a: C; var z: C); (*effettua il complesso coniugato *)
  procedure DivC(a, b: C; var z: C); (*divisione tra due numeri complessi *)

implementation
  function Sign(a: real): integer;
  begin
    if a > 0 then
      Sign:=1
    else if a < 0 then
      Sign:= -1
    else
      Sign:=0
    end;
  end;

  function parteRe(a: C): real;
  begin
    parteRe:= a.Re
  end;

  function parteIm(a: C): real;
  begin
    parteIm:= a.Im
  end;

  function Arg(a: C): real;
  begin
```

```

        if a.Re = 0 then
            Arg:= Sign(a.Im)*PI/2
        else if a.Im = 0 then
            Arg:= (1-Sign(a.Re))*PI/2
        else if a.Re >0 then
            Arg:= arctan(a.Im/a.Re)
        else
            Arg:= arctan(a.Im/a.Re)+ Sign(a.Im)*PI
        end;

function AbsC(a:C):real;
begin
    AbsC := sqrt(sqr(parteRe(a))+sqr(parteIm(a)))
end;

procedure setR(m,n:real; var z:C);
begin
    z.Re:=m;
    z.Im:=n;
end;

procedure setPR(ro, fi:real; var z:C);
begin
    setR(ro*cos(fi),ro*sin(fi),z)
end;

procedure SumC(a,b:C; var z:C);
begin
    setR(parteRe(a)+parteRe(b), parteIm(a)+parteIm(b), z)
end;

procedure DiffC (a, b:C;var z:C);
begin
    setR(parteRe(a)-parteRe(b), parteIm(a)-parteIm(b), z)
end;

procedure MultC (a, b: C; var z:C);
begin
    setR(parteRe(a)*parteRe(b)-parteIm(a)*parteIm(b),
        parteRe(a)*parteIm(b)+parteIm(a)*parteRe(b), z)
end;

procedure Conj(a:C; var z:C);
begin
    setR(parteRe(a), -parteIm(a), z)
end;

procedure DivC(a, b: C; var z:C);
begin
    setPR(AbsC(a)/AbsC(b), Arg(a)- Arg(b), z)
end;

end.

```

Nel programma principale vi è la scelta dell'operazione da eseguire, la lettura dei numeri complessi e la stampa del risultato. Per eseguire le operazioni sui numeri complessi si utilizzano le procedure implementate nella unit complex. Queste possono essere utilizzate da altri programmi che operano con numeri complessi. Anche il tipo di dato *numero complesso* è stato definito nella unit.

```

program CalcolaComplex;
uses Complex, crt;
var r,c1,c2:C;
    rr,ff,pf:real;
    ch:char;

procedure readC(var s: C);
var x,y:real;
begin
    write('Scrivi la parte Reale: ');
    readln(x);
    write('Scrivi la parte Immaginaria: ');
    readln(y);
    SetR(x,y,s);
end;

procedure writeC(s:C);
const t = 0.0000001;
begin
    if Abs(parteIm(s))< t then
        writeln(parteRe(s):8:4)
    else if parteIm(s) > 0 then
        writeln(parteRe(s):8:4, ' +i* ',parteIm(s):8:4)
    else writeln(parteRe(s):8:4, ' -i* ',-parteIm(s):8:4);
end;

procedure lettura (var z1,z2:C);
begin
    writeln('Scrivi z1: ');
    readC(z1);
    writeC(z1);
    writeln('Scrivi z2: ');
    readC(z2);
    writeC(z2);
end;

procedure stampa(z: C);
begin
    writeln;
    writeC(z);
    writeln;writeln;
    writeln('Premere Invio per tornare al menu principale');
    readln;
end;

begin
    clrscr;
    gotoxy(5,10);
    writeln('ooooooo          OPERAZIONI CON I NUMERI COMPLESSI          ooooooo');
    gotoxy(5,20);
    write('          Premi Invio per proseguire...');
    readln;
    repeat
        clrscr;
        writeln('Quale operazione vuoi effettuare?');
        writeln('Premi + per eseguire la somma fra due numeri complessi');
        writeln('Premi - per eseguire la differenza fra due numeri complessi');
        writeln('Premi * per eseguire il prodotto fra due numeri complessi');
        writeln('Premi / per eseguire la divisione fra due numeri complessi');
        writeln('Premi j per avere il complesso coniugato di un numero');
        writeln('Premi r per passare alla rappresentazione cartesiana');
        writeln('Premi p per passare alla rappresentazione polare');

```

```

writeln('Premi   e   per uscire dal programma');
write('Scelta -> ');
readln(ch);
case ch of
'+': begin
    writeln('SOMMA di z1 con z2 ');
    lettura(c1,c2);
    SumC(c1,c2,r);
    writeln('Il risultato della somma è: ');
    stampa(r);
    end;
'-': begin
    writeln('SOTTRAZIONE tra z1 e z2 ');
    lettura(c1,c2);
    DiffC(c1,c2,r);
    writeln;
    writeln('Il risultato della differenza è: ');
    stampa(r);
    end;
'*': begin
    writeln('PRODOTTO tra z1 e z2 ');
    lettura(c1, c2);
    MultC(c1,c2,r);
    writeln('Il risultato del prodotto è: ');
    stampa(r);
    end;
'/': begin
    writeln('DIVISIONE di z1 con z2 ');
    lettura(c1,c2);
    writeln;
    writeln('Il risultato della divisione è:');
    DivC(c1,c2,r);
    stampa(r);
    end;
'j': begin
    writeln('COMPLESSO CONIUGATO di z');
    writeln('scrivi z ');
    readC(c1);
    Conj(c1,r);
    writeln;
    writeln('Il complesso coniugato z* è: ');
    stampa(r);
    end;
'r': begin
    writeln('    P -> R');
    write('scrivi ro: ');
    readln(rr);
    write('scrivi fi (angolo espresso in °): ');
    readln(ff);
    pf:= (ff*PI)/180;
    SetPR(rr,pf,r);
    writeln('Il numero complesso in coordinate cartesiane è: ');
    stampa(r);
    end;
'p': begin
    writeln('    R -> P');
    writeln; writeln('scrivi il numero complesso z: ');
    readC(c1);
    writeln; writeln('Il numero complesso in coordinate polari è: ');
    pf:= Arg(c1);
    ff:=(pf*180)/PI;
    writeln('z = ',AbsC(c1):8:4,' /__',ff:8:4,'°');
    writeln; writeln('Premi Invio per tornare al menu principale');

```

```
        readln;  
        end;  
end;  
until (ch = 'e')  
end.
```

L'istruzione gotoxy definita nell'unità CRT è utilizzata per spostare il cursore sullo schermo a seconda delle coordinate che vengono passate come parametri.

È possibile includere un file contenente le procedure che si intende utilizzare in un dato programma ricorrendo a una direttiva. La sintassi è:

{*\$I Nomefile*}

Nomefile deve contenere il percorso se il file è in una directory diversa da quella nella quale si sta operando e ha come estensione standard .PAS.

Il file da includere viene inserito nel testo compilato subito dopo la direttiva. La direttiva va inserita prima del begin (ad esempio dopo la dichiarazione delle variabili).

La unit CRT

Le procedure e le funzioni implementate nel modulo standard CRT vengono utilizzate per la gestione del video in modalità testo non incluse nel Pascal standard. Lo schermo è visto come un insieme di righe, ciascuna formata da un determinato numero di caratteri. Il numero di righe, e il numero di caratteri per ciascuna di esse, è legato alla particolare modalità video presente nel computer.

L'angolo in alto a sinistra corrisponde alle coordinate 1, 1 (prima riga, prima colonna).

Funzioni e procedure del modulo Crt:

AssignCrt; associa un file di testo alla finestra video.

ClrEol; cancella tutti i caratteri a partire dalla posizione del cursore fino alla fine della linea.

ClrScr; cancella lo schermo e ritorna il cursore nell'angolo superiore sinistro.

Delay (n: word); attende per il numero di millisecondi specificato.

DelLine; cancella la linea contenente il cursore

GotoXY (x, y: byte); sposta il cursore sulle coordinate indicate (riga x e colonna y)

HighVideo; seleziona la modalità dei caratteri *alta luminosità*; i successivi caratteri sono scritti ad alta luminosità.

InsLine; inserisce una linea vuota nella posizione del cursore

KeyPressed; funzione che determina se è stato premuto un tasto sulla tastiera. Ritorna un valore booleano (true o false)

Esempio:

```
uses Crt;  
begin  
    repeat  
        Write('Ciao ');  
    until KeyPressed;  
end.
```

LowVideo; seleziona la modalità dei caratteri *bassa luminosità*; i successivi caratteri sono scritti in bassa luminosità.

NormVideo; ripristina per il testo, il valore originale letto all'inizio del programma

NoSound; spegne l'audio attivato dalla procedura Sound.

ReadKey; è una funzione che legge un carattere dalla tastiera senza effettuarne l'eco sul video. Ritorna il carattere letto, che verrà assegnato ad una variabile di tipo char.

Esempio:

```
uses Crt;
```

```

var c: char;
begin
  writeln('Premi un tasto');
  c:= ReadKey;
  writeln( 'Hai premuto', c, 'con codice ASCII = ', Ord(c));
end.

```

Sound (Hz: word); attiva l'audio; *Hz* specifica la frequenza del suono emesso in herz. Il suono continua fino a quando l'audio non è disattivato dalla procedura *NoSound*.

TextBackground (colore: byte); seleziona il colore dello sfondo; per colore può essere usato il nome o la costante numerica: **Black** (nero) = 0; **Blue** = 1; **Green** (verde) = 2; **Cyan** (Ciano) = 3; **Red** (rosso) = 4; **Magenta** = 5; **Brown** (marrone) = 6; **LightGray** (Grigio chiaro) = 7.

TextColor (colore: byte); seleziona il colore del testo; il valore di colore può essere il nome o la costante numerica: **Black** (nero) = 0; **Blue** = 1; **Green** (verde) = 2; **Cyan** (Ciano) = 3; **Red** (rosso) = 4; **Magenta** = 5; **Brown** (marrone) = 6; **LightGray** (Grigio chiaro) = 7; **DarkGray** (grigio scuro) = 8; **LightBlue** = 9; **LightGreen** = 10; **LightCyan** = 11; **LightRed** = 12; **LightMagenta** = 13; **Yellow** (giallo) = 14; **White** (bianco) = 15; aggiungendo la costante **Blink** = 128 i caratteri scritti dopo la chiamata di *TextColor* sono lampeggianti. Es.: *TextColor* (colore + Blink).

TextMode (modo); attiva il modo testo specificato; modo è una variabile di tipo intero, che può assumere dei valori interi stabiliti ed assegnati a delle costanti definite all'interno del modulo: **BW40** = 0 corrisponde a 40 caratteri per 25 righe monocromatiche, **C40** = 1 corrisponde a 40 caratteri per 25 righe a colori, **BW80** = 2 corrisponde a 80 caratteri per 25 righe monocromatiche, **C80** = 3 corrisponde a 80 caratteri per 25 righe a colori, **Font8x8** = 256 corrisponde a 80 caratteri per 43 righe con adattatore EGA e a 80 caratteri per 50 righe con adattatore VGA.

Questi valori si possono combinare, ad esempio la seguente chiamata alla procedura *TextMode*:

TextMode(C80 + Font8x8) attiva il modo 43 righe e 80 colonne con una EGA o 50 righe e 80 colonne con una VGA a colori.

La chiamata **TextMode>LastMode)** con *LastMode* variabile di tipo integer, ripristina il modo iniziale in quanto all'avvio del programma in *LastMode* viene memorizzato il modo attivo in quel momento.

WhereX; funzione che ritorna la coordinata x corrente del cursore, il valore ritornato è di tipo byte.

WhereY; funzione che ritorna la coordinata y corrente del cursore, il valore ritornato è di tipo byte.

Window (x1,y1,x2,y2); definisce una finestra (zona rettangolare del video) di tipo testo sullo schermo; x1, y1, x2, y2 sono di tipo byte. x1 e y1 sono le coordinate dell'angolo in alto a sinistra e x2 e y2 le coordinate dell'angolo in basso a destra della finestra. Se vengono aperte più finestre, una sola finestra per volta può essere quella attiva.

Esempio:

```
program prova_finestra;
uses crt;

begin
  window(5,3,35,20);
  clrscr;
  delay(1000);
  repeat
    write('$ ');
    delay(10);
  until keypressed;
end.
```

Vediamo un programma che disegna una cornice con una scritta interna lampeggiante:

```
program prova_cornice;
uses crt;

procedure Cornice(x,y,x1,y1,c:integer); (*disegno di cornici *)
var a,b:integer;

begin
  a:=x; b:=y;
  For x:= x to x1 do
  begin gotoxy(x,y); textcolor(c); write(char(196));
  gotoxy(x,y1); write(char(196)); end;
  For y:=y to y1-1 do
  begin gotoxy(a,y+1); write(char(179));
  gotoxy(x1,y+1); write(char(179)); end;
  gotoxy(a,b); write(char(218)); gotoxy(x1,b); write(char(191));
  gotoxy(a,y1); write(char(192)); gotoxy(x1,y1); write(char(217));
  end;

begin
  clrscr;
  cornice(6,6,39,21,4);
  gotoxy(13,10);
  TextColor(4+Blink);
  writeln('Ciao Mondo ');
  readln;
end.
```

La unit graph

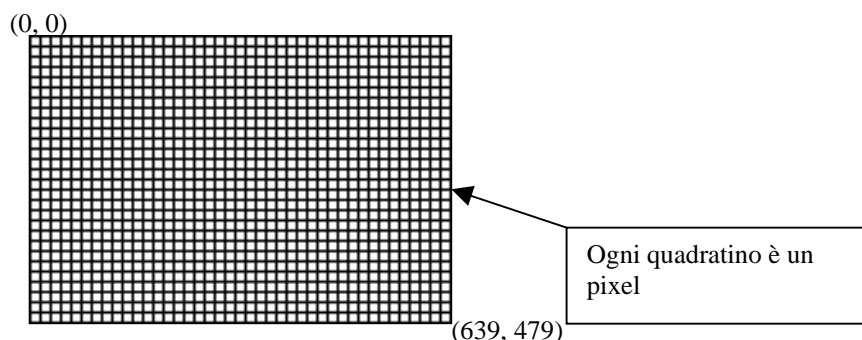
Il Pascal standard non contiene istruzioni per la rappresentazione grafica sullo schermo. Nel Turbo Pascal 7.0 le funzioni grafiche sono state raggruppate nel modulo *graph*. È inoltre presente un modulo chiamato *graph3* contenente le funzioni grafiche relative alla versione 3.0 del Turbo Pascal. In ambiente grafico, il video è visto come un insieme di punti detti *pixel*, che possono essere illuminati o meno per formare immagini grafiche. Il numero di pixel dipende dal tipo di monitor e dal tipo di scheda grafica presente all'interno del computer.

Le principali schede grafiche supportate dal Pascal sono:

- ⇒ **CGA** risoluzione 320 pixel orizzontali per 200 pixel verticali con 3 colori a disposizione più uno per lo sfondo.
- ⇒ **CGA** risoluzione 640 pixel orizzontali per 200 pixel verticali con 1 colore a disposizione più uno per lo sfondo.
- ⇒ **EGA** risoluzione 640 pixel orizzontali per 350 pixel verticali con 16 colori a disposizione.
- ⇒ **VGA** risoluzione 640 pixel orizzontali per 480 pixel verticali con 16 colori a disposizione.

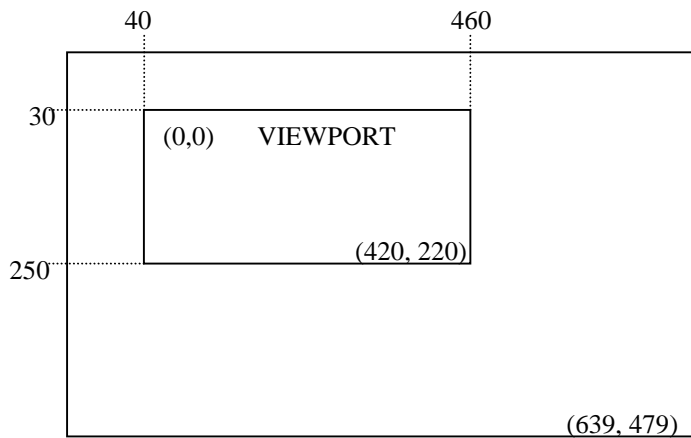
Per poter utilizzare le funzioni grafiche è necessario che insieme al programma siano presenti particolari file, detti driver, con estensione .BGI, che realizzano l'interfaccia con la scheda grafica. Il driver adatto deve essere presente sul disco nella directory specificata nel programma che si manda in esecuzione.

Il video può essere paragonato ad un foglio di carta millimetrata. L'origine del foglio è situata nell'angolo in alto a sinistra e tale punto ha coordinate (0, 0). La coordinata dell'angolo in basso a destra dipende dalla modalità grafica. Ad esempio con una scheda VGA si ha:



In modalità grafica esiste un cursore, definito puntatore corrente (CP) non visibile, che contraddistingue un punto all'interno dell'area attiva di disegno. Da questo punto inizierà la rappresentazione grafica se la posizione non viene variata dalle istruzioni grafiche.

L'intero schermo può essere suddiviso in riquadri rettangolari più piccoli chiamati *viewport*; e tutti gli elementi grafici, comprese le coordinate sono riferiti alla viewport attiva.



Principali procedure e funzioni del modulo graph:

Arc (x, y: integer; AngIniz, AngFin, Raggio: word); procedura che disegna un arco in senso antiorario; x, y sono le coordinate del punto centrale dell'arco; gli angoli sono espressi in gradi. Esempio:

```

program provaGraf;
uses Graph;

var
  Gd, Gm: Integer;
  Radius: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, 'c:\tp\bgi ');
  if GraphResult <> grOk then
    Halt(1);
  for Radius := 1 to 5 do
    Arc(100, 100, 0, 90, Radius * 10);
  Readln;
  CloseGraph;
end.

```

Bar (x1,y1,x2,y2: integer); traccia un rettangolo riempito con un motivo e colorato. Usa il motivo e il colore definiti da **SetFillPattern** e **SetFillStyle**.

Circle (x, y: integer; Raggio: word); traccia un cerchio del colore settato con **SetColor** usando (x, y) come centro.

ClearViewPort; cancella la ViewPort corrente e sposta il cursore nel punto (0, 0). Esempio:

```

program prova_ClearViewPort;
uses Graph;

```

```

var Gd, Gm: Integer;
begin
Gd := Detect;
InitGraph(Gd, Gm, 'c:\tp\bgi ');
if GraphResult <> grOk then
Halt(1);
Rectangle(19, 19, GetMaxX - 19, GetMaxY - 19);
SetViewPort(20, 20, GetMaxX - 20, GetMaxY - 20, ClipOn);
OutTextXY(5, 10, '<ENTER> per cancellare la viewport:');
Readln;
ClearViewPort;
OutTextXY(0, 0, '<ENTER> per uscire:');
Readln; CloseGraph;
end.

```

Closegraph; chiude il modo grafico; questa procedura consente di ritornare allo schermo precedente, dal quale si è passati in modalità grafica.

DetectGraph (var driver, mode: integer); la procedura rileva automaticamente il tipo di driver e il modo grafico corrispondente alla scheda grafica. Se non viene rilevato un hardware grafico in driver viene scritto il valore grNotDetected (costante predefinita), valore ritornato anche dalla funzione GraphResult.

DrawPoly (Numpunti: word; var PuntiPolig); traccia le linee di un poligono nel colore settato.

Esempio:

```

program poligono;
uses Graph;
const
Triangle: array[1..4] of PointType = ((X: 50; Y: 100), (X: 100; Y:100),
(X: 150; Y: 150), (X: 50; Y: 100));
var Gd, Gm: Integer;
begin
Gd := Detect;
InitGraph(Gd, Gm, 'c:\tp\bgi ');
if GraphResult <> grOk then
    Halt(1);
    DrawPoly(SizeOf(Triangle) div SizeOf(PointType), Triangle);{ 4 }
    Readln;
CloseGraph;
end.

```

Ellipse (x, y: integer; AngIniz, AngFin: word; xRaggio, yRaggio: word); traccia un arco ellittico in senso antiorario; (x, y) sono le coordinate del centro; gli angoli sono espressi in gradi; xRaggio è l'asse orizzontale e yRaggio l'asse verticale dell'ellisse.

FillEllipse; traccia un ellisse riempito con un motivo definito.

GetBkColor; funzione che ritorna il colore dello sfondo.

GetColor; ritorna il colore del disegno.

GetGraphMode; ritorna il modo grafico corrente.

GetImage (x1, y1, x2, y2: integer; var Bitmap); salva un'immagine bitmap di una regione specificata in un buffer; x1, y1, x2, y2 definiscono la regione rettangolare dello schermo.

GetMaxX; ritorna il massimo numero di pixel orizzontali del modo grafico corrente.

GetMaxY; ritorna il massimo numero di pixel verticali del modo grafico corrente.

GetPixel (x, y: integer):word; ritorna il colore del *pixel* del punto (x, y).

GetX; ritorna la coordinata x (integer) del puntatore corrente.

GetY; ritorna la coordinata y (integer) del puntatore corrente.

GraphResult; ritorna un codice d'errore dell'ultima operazione grafica o il valore grOk (costante predefinita di valore 0) se non c'è stato errore.

ImageSize; ritorna il numero di byte necessari per caricare una regione rettangolare dello schermo.

InitGraph (var Driver, Mode: integer; Percorso: string); pone il sistema in modalità grafica; *Driver* deve contenere il tipo di driver della scheda grafica utilizzata, *Mode* la modalità grafica supportata dal Turbo Pascal. *Percorso* deve contenere l'indicazione (path) della directory dove si trovano, sul disco, i driver grafici.

Line (x1, y1, x2, y2: integer); traccia una linea dal punto (x1, y1) al punto (x2, y2).

LineRel (Dx, Dy: integer); traccia una linea dal puntatore corrente ad una distanza espressa da *Dx* e *Dy*.

LineTo (x, y: integer); traccia una linea dal puntatore corrente al punto (x, y).

MoveRel (Dx, Dy: integer); sposta il puntatore corrente a una distanza relativa alla posizione corrente ed espressa da *Dx* e *Dy*.

MoveTo (x, y: integer); sposta il puntatore corrente nel punto (x, y).

OutText (testo: string); visualizza una stringa a partire dalla posizione attuale del puntatore corrente. *OutText('Testo')*; visualizza una stringa costante.

OutTextXY (x, y: integer; testo: string); visualizza una stringa a partire dalle coordinate (x, y).

PutPixel (x, y: integer; Colore: word); accende un pixel del colore specificato nel punto di coordinate (x, y).

Rectangle (x1, y1, x2, y2: integer); disegna un rettangolo con spigolo superiore in (x1, y1) e inferiore in (x2, y2).

SetBkColor (Colore: word); imposta il colore dello sfondo.

SetColor (Colore: word); imposta il colore del disegno.

SetGraphMode (Modo: integer); imposta un determinato sistema grafico e cancella lo schermo.

SetLineStyle (Stile, Pattern, Thickness: word); imposta lo stile e la larghezza della linea.

SetPalette (ColoreNum: word; Colore: shortint); cambia una tavolozza (palette) di colori come specificato da *ColoreNum* e *Colore*.

SetRGBPalette (ColoreNum, RedValue, GreenValue, BlueValue: integer); modifica la tavolozza per i driver grafici IBM 8514 e VGA. *ColoreNum* definisce un indice del colore, mentre gli altri tre parametri indicano le componenti dei tre colori fondamentali (rosso, verde e blu) che formano il colore che sarà caricato nella tavolozza. Per l'IBM 8514 *ColoreNum* può assumere un valore compreso fra 0 e 255 mentre per la VGA tra 0 e 15.

SetTextStyle (Font, Direzione, CharSize: word); definisce lo stile del testo in modalità grafica. I valori da assegnare ai parametri sono:

<i>Costante</i>	<i>Valore</i>	<i>Significato</i>
DefaultFont	0	8x8 bit mapped font
TriplexFont	1	Stroked font
SmallFont	2	Stroked font
SansSerifFont	3	Stroked font
GothicFont	4	Stroked font
HorizDir	0	Orient left to right
VertDir	1	Orient bottom to top
UserCharSize	0	User-defined character size

SetViewPort (x1, y1, x2, y2: integer; clip: boolean); definisce la ViewPort corrente (finestra in modalità grafica). Il parametro *clip* determina se gli elementi grafici saranno visualizzati nei limiti della viewport corrente; in questo caso il parametro deve avere il valore *ClipOn* o *true*.

TextHight (Stringa: string): word; ritorna l'altezza di una stringa, in pixel.

TextWidth (Stringa: string): word; ritorna la larghezza di una stringa, in pixel.

Vediamo i tipi di driver e le modalità di impostazione con i valori interi corrispondenti:

Detect	0/Requests autodetection
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMono	5
IBM8514	6
HercMono	7
ATT400	8
VGA	9
PC3270	10

<i>Costante</i>	<i>Valore</i>	<i>Significato</i>	<i>Costante</i>	<i>Valore</i>	<i>Significato</i>
CGAC0	0	320 x 200	EGALo	0	640 x 200
CGAC	1	320 x 200	EGAHi	1	640 x 350
CGAC2	2	320 x 200			
CGAC3	3	320 x 200	EGA64Lo	0	640 x 200
CGAHi	4	640 x 200	EGA64Hi	1	640 x 350
MCGAC0	0	320 x 200	ATT400C0	0	320 x 200
MCGAC1	1	320 x 200	ATT400C1	1	320 x 200
MCGAC2	2	320 x 200	ATT400C2	2	320 x 200
MCGAC3	3	320 x 200	ATT400C3	3	320 x 200
MCGAMed	4	640 x 200	ATT400Med	4	640 x 200
MCGAHi	5	640 x 480	ATT400Hi	5	640 x 400
EGAMonoHi	3	640 x 350	IBM8514Lo	0	640 x 480
HercMonoHi	0	720 x 348	IBM8514Hi	1	1024 x 768
VGALo	0	640 x 200	PC3270Hi	0	720 x 350
VGAMed	1	640 x 350	VGAMed	2	640 x 480

Nel modulo `graph` non è stata implementata una procedura o una funzione per la lettura di stringhe da tastiera con eco sullo schermo grafico (l'equivalente dell'istruzione `read` con il video in modalità testo). Possiamo, allora, costruire una *unit* che implementi la funzione chiamata ***InText*** per leggere delle stringhe. La unit viene chiamata *graph2*:

Unit `Graph2`;

Interface

```
uses graph, crt;
function InText: string;
```

Implementation

function `InText`: string;

```
var c:char;
    Col, x, d:integer;
    stringa: string;

begin
    Col:= GetColor;
    x:=GetX;
    stringa:='';
    c:=ReadKey;
    while c<> #13 do
    begin
        if (Ord(c)>32) and (Ord(c)<126) then
        begin
            OutText(c);
            stringa:= Concat(stringa,c);
        end;
        if (c=#8) And (GetX>x) then
        begin
            d:=TextWidth(copy(stringa,(Length(stringa)),1));
            MoveRel(-d,0);
            SetColor(GetBkColor);
            OutText(Copy(Stringa,(Length(stringa)),1));
        end;
    end;
```

```

        SetColor(Col);
        MoveRel(-d,0);
        stringa:=Copy(Stringa,0,(Length(stringa)-1));
    end;
    c:= ReadKey;
end;
Intext:=stringa;
end;

```

end.

Vediamo un semplice programma che utilizza la unit graph2:

```

program prova_InText;
uses Graph, Graph2;

var
    Gd, Gm: Integer;
    ttt:string;
begin
    Gd := Detect;
    InitGraph(Gd, Gm, 'c:\tp\bgi ');
    if GraphResult <> grOk then
        Halt(1);
    OutText('Scrivi una stringa: ');
    ttt:=InText;
    OutTextXY(0,100,'La stringa letta è: ');
    OutTextXY(0,200,ttt);
    OutTextXY(0,300,'Premi Enter per uscire');
    Readln;
    CloseGraph;
end.

```

Vediamo un esempio di programma che presenta alcuni concetti di grafica visti: il programma traccia una griglia e poi un grafico fra quelli selezionati. Tale esempio ci consente di vedere anche l'applicazione di numerose procedure e funzioni del modulo graph:

```

{ ***** }
{ Programma GRAFICI: Visualizza i grafici di alcune funzioni }
{ ***** }
PROGRAM grafici;
USES
    graph,crt,printer;
TYPE
    insieme_dati=ARRAY[0..427] OF real;
VAR
    scelta          : Char;
    A,B,C,base     : Real;
    nom_funct       : String;
    strtmp          : String;
    valori          : insieme_dati;
    n_punti         : Integer;
    i               : Integer;
    x               : Real;
    scala           : Real;
    vxmin,vxmax    : Real;
    xbordo,ybordo  : Integer;

FUNCTION inizializza_grafica:boolean;
var gd,gm:integer;

```

```

BEGIN
  detectgraph(gd, gm);
  initgraph(gd, gm, 'c:\tp\bgi');
  if graphresult=0 then
  BEGIN
    settextstyle(2, horizdir, 0);
    xbordo:=trunc((getmaxx+1)/6);
    ybordo:=trunc((getmaxy+1)/6);
    inizializza_grafica:=true
  END
  else
    inizializza_grafica:=false
  END;

PROCEDURE griglia(xmin,xmax:real; nome_var_x,unita_misura:string);
var i:byte;
    sc,dx:real;
    maxx,maxy:integer;
    s:string;
BEGIN
  setcolor(3);
  maxx:=getmaxx;
  maxy:=getmaxy;
  sc:=(maxx-2*xbordo)/10;
  dx:=(xmax-xmin)/10;
  setlinestyle(dottedln,0,normwidth);
  {imposta la linea tratteggiata}
  for i:=0 to 10 do
  BEGIN
    line(xbordo+trunc(i*sc),ybordo,xbordo+trunc(i*sc),maxy-ybordo);
    str(xmin+dx*i:8:2,s);
    outtextxy(xbordo+trunc(i*sc)-30,maxy-ybordo+10,s);
  END;
  outtextxy(maxx div 2,maxy-ybordo+22,nome_var_x+' '+unita_misura);
  sc:=(maxy-2*ybordo)/10;
  for i:=0 to 10 do
    line(xbordo,ybordo+trunc(i*sc),maxx-xbordo,ybordo+trunc(i*sc));
  setlinestyle(solidln,0,normwidth);
  END;

PROCEDURE minmax(y:insieme_dati; da,a:integer;var ymin,ymax:real);
var i:integer;
BEGIN
  ymax:=y[da];
  ymin:=y[da];
  for i:=da+1 to a do
  BEGIN
    if y[i]>ymax then
      ymax:=y[i];
    if y[i]<ymin then
      ymin:=y[i]
  END;
  if ymax=ymin then
  BEGIN
    ymax:=1.1*ymax;
    ymin:=0.9*ymin;
    if ymax=0 then
    BEGIN
      ymax:=1;
      ymin:=-1
    END;
  END;
  END;

```

```

PROCEDURE asse_y(y:insieme_dati; da,a:integer;
                 nome_var_y,unita_misura:string; colore,dove:byte);
var i:byte;
    sc,dy,ymax,ymin:real;
    maxx,maxy:integer;
    s:string;
BEGIN
    setcolor(colore);
    maxx:=getmaxx;
    maxy:=getmaxy;
    minmax(y,da,a,ymin,ymax);
    sc:=(maxy-2*ybordo)/10;
    dy:=(ymax-ymin)/10;
    for i:=0 to 10 do
    BEGIN
        str(ymin+dy*i:8:2,s);
        if dove=0 then
            outtextxy(5,maxy-ybordo-trunc(i*sc)-5,s)
        else
            outtextxy(maxx-xbordo+35,maxy-ybordo-trunc(i*sc)-5,s)
        END;
        if dove=0 then
            outtextxy(xbordo-10,ybordo-20,nome_var_y+' '+unita_misura)
        else
            outtextxy(maxx-xbordo-10,ybordo-20,nome_var_y+' '+unita_misura)
        END;
    END;
END;

```

```

PROCEDURE grafico_xy(y:insieme_dati; da,a:integer; colore:byte);
var ymin,ymax,sx,sy:real;
    i,maxx,maxy:integer;
BEGIN
    setcolor(colore);
    maxx:=getmaxx;
    maxy:=getmaxy;
    minmax(y,da,a,ymin,ymax);
    sy:=(maxy-2*ybordo)/(ymax-ymin);
    sx:=(maxx-2*xbordo)/(a-da);
    moveto(xbordo,maxy-ybordo-trunc((y[da]-ymin)*sy));
    for i:=da+1 to a do
        lineto(xbordo+trunc(sx*(i-da)),maxy-ybordo-trunc((y[i]-ymin)*sy))
    END;
END;

```

```

PROCEDURE Maschera;
BEGIN
    textbackground(1);
    textcolor(15);
    clrscr;
    textbackground(1);
    gotoxy(20,3); writeln('* programma GRAFICI * ');
    gotoxy(1,1); write(char(201));
    FOR i:=2 TO 79 DO write(char(205));
    write(char(187));
    FOR i:=2 TO 23 DO
        BEGIN
            gotoxy(1,i);
            write(char(186));
            gotoxy(80,i);
            write(char(186))
        END;
    gotoxy(1,24); write(char(200));
    FOR i:=2 TO 79 DO write(char(205));
    write(char(188));
END;

```

```

gotoxy(1,5); write(char(204));
FOR i:=2 TO 79 DO write(char(205));
write(char(185));
END;
{ La procedura crea una cornice in modalit  testo}

PROCEDURE stampa_grafica;
var maxx,maxy:integer;
    yy,y,x,yb:word;
    bitmap,bkcolor:byte;
begin
    maxx:=getmaxx;
    maxy:=getmaxy;
    bkcolor:=getbkcolor;
    write(lst,#27,#64);
    write(lst,#27,#67,#0,#11);
    write(lst,#27,#51,#24);
    for y:=0 to (maxy div 8) do
    begin
        yb:=y*8;
        write(lst,#10,#13,#27,#42,#4,chr((maxx+1) mod 256),chr((maxx+1) div 256));
        for x:=0 to maxx do
        begin
            bitmap:=0;
            for yy:=0 to 7 do
            begin
                if getpixel(x,yb+yy)<>bkcolor then
                    bitmap:=bitmap+(128 shr yy)
            end;
            write(lst,chr(bitmap))
        end
    end;
    writeln(lst)
end;
{ Stampa l'immagine presente sullo schermo su una stampante ad aghi }
PROCEDURE Seleziona_Scelta;
BEGIN
    gotoxy(5,7); writeln(' 1) Esponenziale:  $Y = B^x$ ');
    gotoxy(5,9); writeln(' 2) Seno :  $Y = \text{SIN}( aX )$ ');
    gotoxy(5,11); writeln(' 3) Coseno :  $Y = \text{COS}( aX )$ ');
    gotoxy(5,13); writeln(' 4) Parabola :  $Y = ax^2 + bx + c$ ');
    gotoxy(5,15); writeln(' 5) Retta :  $Y = aX + b$ ');
    gotoxy(5,17); writeln(' 0) Uscita dal programma');
    REPEAT
        gotoxy(30,20); write('< seleziona la funzione desiderata > _ ');
        gotoxy(69,20); readln(scelta);
    UNTIL (scelta<'6') AND (Scelta>='0');
END;

{ programma principale}
BEGIN
    Maschera;
    Seleziona_Scelta;
    Maschera;
    vxmin:=-10;
    vxmax:=+10;
    n_punti:=427;
    scala:=(vxmax-vxmin)/n_punti;
    CASE scelta OF
        '1': BEGIN
            gotoxy(20,3); writeln('*** INIZIO PROGRAMMA ESPONENZIALE ***');
            gotoxy(8,7); write ('scrivi la base : ');
            readln(B);

```

```

IF inizializza_grafica= false THEN exit;
str(B:2:2,strtmp);
nom_funct:='Esponenziale con base: '+strtmp;
griglia(0,vxmax,nom_funct,'');
FOR i:=1 TO N_punti DO
    BEGIN
        x:=vxmin+(i-1)*scala;
        valori[i]:=exp(x*ln(B));
    END;
asse_y(valori,1,n_punti,'',' ',3,0);
grafico_xy(valori,1,n_punti,14);
END;
'2': BEGIN
    gotoxy(20,3); writeln(' *** INIZIO PROGRAMMA SIN(aX) *** ');
    gotoxy(8,7); write ('scrivi il coefficiente A di X : ');
    readln(A);
    IF inizializza_grafica= false THEN exit;
    str(A:2:2,strtmp);
    nom_funct:='Sin('+strtmp+'*X)';
    griglia(-pi,+pi,nom_funct,'');
    FOR i:=1 TO n_punti DO
        BEGIN
            x:=(vxmin+(i-1)*scala)/pi;
            valori[i]:=sin(A*x);
        END;
    asse_y(valori,1,n_punti,'',' ',3,0);
    grafico_xy(valori,1,n_punti,14);
END;
'3': BEGIN
    gotoxy(20,3); writeln(' *** INIZIO PROGRAMMA COS(aX) *** ');
    gotoxy(8,7); write ('scrivi il coefficiente A di X : ');
    readln(A);
    IF inizializza_grafica= false THEN exit;
    str(A:2:2,strtmp);
    nom_funct:='Cos('+strtmp+'*X)';
    griglia(-pi,+pi,nom_funct,'');
    FOR i:=1 TO N_punti DO
        BEGIN
            x:=(vxmin+(i-1)*scala)/pi;
            valori[i]:=cos(A*x);
        END;
    asse_y(valori,1,n_punti,'',' ',3,0);
    grafico_xy(valori,1,n_punti,14);
END;
'4': BEGIN
    gotoxy(20,3); writeln(' *** INIZIO PROGRAMMA PARABOLA *** ');
    gotoxy(8,7); write ('scrivi il coefficiente A : ');
    readln(A);
    gotoxy(8,9); write ('scrivi il coefficiente B : ');
    readln (B);
    gotoxy(8,11); write ('scrivi il coefficiente C : ');
    readln (C);
    IF inizializza_grafica= false THEN exit;
    str(A:2:2,strtmp);
    nom_funct:='Parabola con A='+strtmp+' B=';
    str(B:2:2,strtmp);
    nom_funct:=nom_funct+strtmp+' C=';
    str(C:2:2,strtmp);
    nom_funct:=nom_funct+strtmp;
    griglia(vxmin,vxmax,nom_funct,'');
    FOR i:=1 TO n_punti DO
        BEGIN
            x:=vxmin+(i-1)*scala;

```

```

        valori[i]:=A*x*x+B*x+C;
    END;
    asse_y(valori,1,n_punti,' ',' ',3,0);
    grafico_xy(valori,1,n_punti,14);
END;
'5': BEGIN
    gotoxy(20,3); writeln(' *** PROGRAMMA Y = AX + B *** ');
    gotoxy(8,7); write ('scrivi il coefficiente A : ');
    readln(A);
    gotoxy(8,9); write ('scrivi il coefficiente B : ');
    readln (B);
    IF inizializza_grafica= false THEN exit;
    str(A:2:2,strtmp);
    nom_funct:='Retta '+strtmp+'*X+';
    str(B:2:2,strtmp);
    nom_funct:=nom_funct+strtmp;
    griglia(vxmin,vxmax,nom_funct,' ');
    FOR i:=1 TO n_punti DO
        BEGIN
            x:=vxmin+(i-1)*scala;
            valori[i]:=A*x+B;
        END;
    asse_y(valori,1,n_punti,' ',' ',3,0);
    grafico_xy(valori,1,n_punti,14);
END;
'0': BEGIN
    Clrscr;
    gotoxy (35,15); write ('Programma Terminato');
    exit;
END;
END;
outtextxy(20,1,'Premere lo <spazio> per stampare');
readln(scelta);
IF (scelta=' ') THEN
    BEGIN
        { Stampa_Grafica; }
        outtextxy(20,20,'Grafico stampato');
        readln;
    END;
closegraph;
END.

```

Il programma consente la stampa dei grafici ma solo su stampante ad aghi; la scrittura attraverso l'istruzione write avviene in un file chiamato *lst* che rappresenta la stampante. In questo file vengono scritti dei caratteri ASCII che consentono l'impostazione della stampante e i pixel dello schermo.

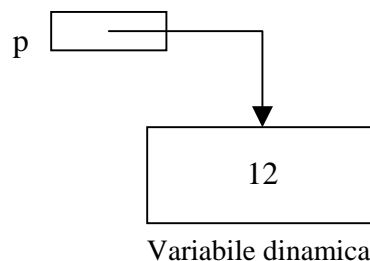
I puntatori

È un tipo di dato del Pascal. Il puntatore è una variabile che contiene come valore l'indirizzo di un'altra variabile. Ogni dato assegnato a una variabile è registrato nella memoria del calcolatore; occuperà un certo numero di byte contigui a partire da una posizione ben definita. Ogni locazione di memoria è identificata da un indirizzo. L'indirizzo di una variabile coincide con l'indirizzo del primo byte che essa occupa. I puntatori vengono utilizzati nei programmi per accedere alla memoria, ovvero per fare riferimento alle variabili utilizzando direttamente i loro indirizzi. In Pascal, ogni puntatore deve essere dichiarato come tipo all'inizio del programma. Il tipo del puntatore è uguale al tipo cui appartiene il dato al quale il puntatore si riferisce. Se il puntatore è riferito ad una variabile di tipo record, anche il puntatore sarà di tipo record.

```
☞      Type punt = ^elemento;
        elemento = record
        campo1: string;
        campo2: string;
        end;
```

Il simbolo “ \uparrow ” è usato in Pascal per indicare un puntatore; questo simbolo, assente dalla tastiera, viene sostituito dal carattere “ \wedge ”, significa “punta a ...” e serve a definire il tipo di puntatore. Quindi al tipo *punt* dichiarato nell'esempio precedente possono appartenere delle variabili, che saranno puntatori ad oggetti di tipo elemento. Il simbolo **nil** predefinito rappresenta un puntatore vuoto, qualunque sia il tipo. I puntatori permettono l'utilizzo di strutture dati dinamiche che possono essere allocate in memoria o deallocate dinamicamente per mezzo delle procedure **new(puntatore)** e **dispose(puntatore)**. Esempio:

```
type      punt = ^integer;
var      p: punt;
begin
    new(p);
    p $\wedge$  :=12;
end.
```



Le variabili create dinamicamente sono allocate in una speciale area di memoria, detta **heap**, la cui gestione è totalmente a carico del programmatore. Quindi quando una variabile non viene più utilizzata deve essere deallocata con la procedura *dispose (puntatore)* per liberare memoria.

Le operazioni permesse con i puntatori sono:

- Accesso alla locazione di memoria alla quale il puntatore fa riferimento;
- L'assegnamento ad una variabile puntatore del valore di un'altra variabile dello stesso tipo o del valore *nil*;
- Il confronto;
- Allocazione e deallocazione dinamica della struttura alla quale fa riferimento il puntatore.

Vediamo alcuni semplici esempi di uso dei puntatori:

```

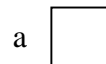
type pelem = ^elemento;
  elemento = record
    num: integer;
    punt: pelem;
  end;

```

```

var a, b: pelem;
begin
  { creazione di un elemento }

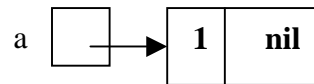
```



```

new (a);
a^.num := 1;
a^.punt := nil;

```



```

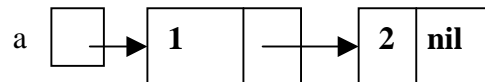
{ inserzione di un elemento in seconda posizione }

```

```

new (b);
b^.num := 2;
b^.punt := nil;
a^.punt := b;

```



```

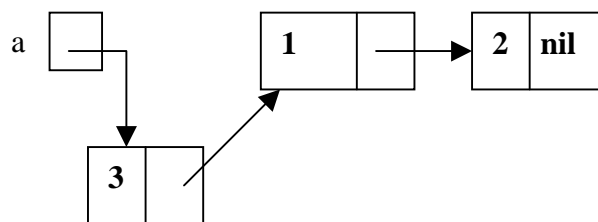
{ inserzione in testa di un elemento }

```

```

new (b);
b^.num := 3;
b^.punt := a;
a := b;

```



```

{ eliminazione di un elemento }

```

```

b := a^.punt^.punt;
{ in b viene salvato il puntatore all'elemento contenente l'intero 2 }
dispose (a^.punt);

```

```

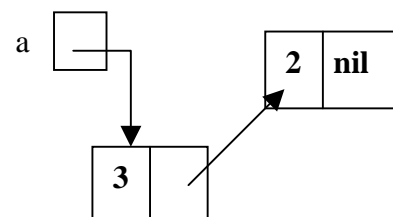
{ deallocazione della memoria contenente
l'elemento con il campo di valore 1 }

```

```

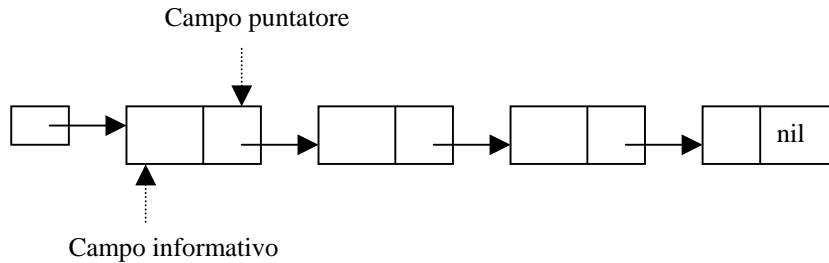
a^.punt := b;
end.

```



L'utilizzo di variabili di tipo puntatore permette la costruzione di strutture dati dinamiche complesse come le liste, le liste doppie e gli alberi.

Vediamo la rappresentazione grafica di una lista semplice:



Una *lista*, come un array, è una collezione di elementi dello stesso tipo record, nei quali uno dei campi è un puntatore al successivo elemento. Negli altri campi vengono memorizzati i dati. L'importanza delle liste, e quindi il loro impiego al posto degli array, è dovuta al fatto che esse sono strutture dati dinamiche, dunque di dimensione variabile.

Nel caso di un array, invece, quando viene dichiarato, viene anche stabilita la lunghezza che dovrà rimanere invariata per tutto il programma, essendo l'array una struttura dati statica.

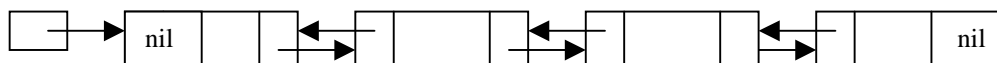
La dimensione di una lista aumenta quando viene inserito un nuovo elemento e diminuisce quando un elemento viene rimosso.

Un'altra differenza tra *array* e *lista* è rappresentata dal tipo di accesso. Per fare riferimento ad esempio al decimo elemento di un array è sufficiente usare l'indice 10 scrivendo $v[10]$ considerando v come nome del vettore. Invece se gli stessi dati fossero memorizzati in una lista sarebbe necessario scorrere in modo sequenziale tutti gli elementi che precedono l'elemento cercato.

Su una lista si possono effettuare le seguenti operazioni fondamentali:

- Inserire un nuovo elemento in una qualunque posizione della lista.
- Estrarre un elemento da una qualunque posizione.
- Stabilire se la lista contiene degli elementi oppure è vuota.

Esistono tipi di lista più complesse, formate da elementi che contengono più di un campo puntatore. Ad esempio la lista doppia ha un campo puntatore all'elemento successivo e uno all'elemento precedente:



Una lista in cui l'ultimo elemento inserito sarà il primo ad essere eliminato realizza una struttura dati detta pila (LIFO), mentre se l'ultimo elemento inserito sarà l'ultimo ad essere eliminato realizza una struttura dati detta coda (FIFO). Queste strutture dati sono semplici e non consentono l'inserzione di un elemento in posizioni intermedie.

Vediamo un programma di creazione e gestione di una lista semplice con le seguenti operazioni:

- **Creazione** procedura che si occupa di creare un nuovo elemento da inserire nella lista, che inizialmente è vuota
- **Inserzione** inserimento di un elemento creato nella lista. Esistono vari tipi di inserzione:
 - ☞ *Inserzione in testa* L'elemento creato viene aggiunto in testa alla lista, diventando quindi il primo elemento
 - ☞ *Inserzione in coda* L'elemento creato viene posto come ultimo nella catena.
 - ☞ *Inserzione in mezzo* Inserimento del dato in una posizione ben precisa, fra due elementi esistenti.
- **Ricerca** procedura che consente di ricercare un particolare elemento che contiene una precisa informazione
- **Rimozione** una procedura che consente di cancellare un dato elemento dalla lista.

Nel programma seguente viene considerata una lista semplice di record formati da due campi informazione e un campo puntatore all'elemento successivo. L'inserimento avviene seguendo l'ordine crescente della stringa del campo nome.

```

program lista;
uses crt;
type punt = ^elemento;
   elemento = record
     nome:string[30];
     tel:string[20];
     next: punt;
   end;
var persona: elemento;
    ch:char;
    nm:string;
    pt:punt;

procedure crea_nuovo_el (var nuovo:punt; inf:elemento);
var temp: punt;
begin
  new(temp);
  temp^.nome:= inf.nome;
  temp^.tel:= inf.tel;
  temp^.next:= nil;
  nuovo:= temp
end;
```

```

procedure ricerca_elemento (var testa: punt; n:string);
var temp: punt;
    posiz: integer;
begin
    if testa = nil then
        writeln('La lista è vuota')
    else
        begin
            temp:= testa;
            posiz:=1;
            while (temp^.nome <> n) and (temp^.next <> nil) do
                begin
                    temp:= temp^.next;
                    posiz:=posiz + 1;
                end;
            if (temp^.nome = n) then
                writeln(n, ' è presente nella posizione # ',posiz,' con
                    telefono:',temp^.tel)
            else
                writeln(n, ' non è presente nella lista');
            end
        end;
end;

procedure inserisci (var testa: punt; inf:elemento);
var temp, nuovo, prec:punt;
begin
    crea_nuovo_el (nuovo,inf);
    if testa = nil then testa:= nuovo
    else
        begin
            temp:= testa;
            if (temp^.nome > inf.nome) and (temp^.next = nil) then
                begin
                    nuovo^.next:= temp;
                    testa:=nuovo;
                end
            else
                begin
                    while (temp^.nome < inf.nome) and (temp^.next <> nil) do
                        begin
                            prec:=temp;
                            temp:= temp^.next;
                        end;
                    if (temp^.next = nil) and (temp^.nome < inf.nome) then
                        begin
                            nuovo^.next := temp^.next;
                            temp^.next:= nuovo;
                        end
                    else
                        begin
                            nuovo^.next:=temp;
                            prec^.next:= nuovo;
                        end
                    end
                end;
        end;
end;

procedure cancella_el ( var testa: punt; n:string);
var prec, temp: punt;
begin
    if testa = nil then writeln('La lista è vuota!')
    else
        if testa^.nome = n then

```

```

begin
    temp:= testa;
    testa:= testa^.next;
    dispose(temp); { eliminazione del dato dalla memoria }
end
else
begin
    temp:= testa;
    prec:= nil;
    while (temp^.next <> nil) and (temp^.nome <> n) do
    begin
        prec := temp;
        temp:=temp^.next;
    end;
    if temp^.nome = n then
    begin
        prec^.next := temp^.next;
        dispose(temp);
        writeln('E' stato eliminato il record di ',n);
    end
    else
        writeln(n, ' non è presente nella lista');
    end
end;

begin
clrscr;
writeln('----- Lista di record -----');
writeln('"""""""" Operazioni consentite """"""""');
writeln('r -> Ricerca per nome');
writeln('i -> Inserisci');
writeln('c -> Cancella');
writeln('e -> Esci');
    repeat
gotoxy(0,8);
write('Scelta ---> ');
readln(ch);
case ch of
'r':begin
    write('Scrivi il nome: ');
    readln(nm);
    ricerca_elemento(pt,nm);
    end;
'i':begin
    writeln('***** Inserimento *****');
    write('Scrivi il nome: ');
    readln(persona.nome);
    write('Scrivi il numero di telefono: ');
    readln(persona.tel);
    inserisci(pt, persona);
    end;
'c':begin
    writeln('***** Cancellazione *****');
    write('Scrivi il nome: ');
    readln(nm);
    cancella_el (pt, nm);
    end
end;
until ch = 'e';
end.

```

Vediamo ora un programma che realizza un'agenda utilizzando una struttura dati dinamica di tipo coda. I dati vengono letti da un file e poi salvati dopo eventuali modifiche o inserimenti nello stesso file chiamato agend.dat.

```
program coda_agenda;
uses crt;

type punt=^scheda;
  scheda= record
    nome:string[15];
    tel:string[12];
    next:punt
  end;
  gente=record
    nome:string[15];
    tel:string[12]
  end;

  f=file of gente;

var ptesta,pcoda:punt;
  persona:gente;
  opz:integer;
  fp:f;

procedure inizio(var ptesta,pcoda:punt;var fp:f);
var pers:gente;
  pt:punt;
  j:integer;

begin
  assign(fp, 'agend.dat');
  reset(fp);
  if eof(fp) then writeln('Agenda vuota');
  pt:=NIL;
  while not eof(fp) do
  begin
    new(pt);
    read(fp,pers);
    pt^.nome:=pers.nome;
    pt^.tel:=pers.tel;
    pt^.next:=NIL;
    if ptesta=NIL then
      begin
        ptesta:=pt;
        pcoda:=pt;
      end
    else
      begin
        ptesta^.next:=pt;
        ptesta:=pt;
      end;
  end;
end;
end;

procedure salva(pcoda:punt;var fp:f);
var pers:gente;
  pt:punt;

begin
  pt:=pcoda;
```

```

rewrite(fp);
while pt<>NIL do
begin
pers.nome:=pt^.nome;
pers.tel:=pt^.tel;
write(fp,pers);
pt:=pt^.next;
end;
end;

procedure insert(var ptesta,pcoda:punt;pers:gente);
var pt:punt;

begin
new(pt);
pt^.nome:=pers.nome;
pt^.tel:=pers.tel;
pt^.next:=NIL;
if ptesta = NIL then
begin
ptesta:=pt;
pcoda:=pt;
end
else
begin
ptesta^.next:=pt;
ptesta:=pt;
end
end;

procedure extract(var ptesta,pcoda:punt;var pers:gente);
var pt:punt;

begin
if pcoda = NIL then
writeln('attenzione coda vuota')
else
begin
pers.nome:=pcoda^.nome;
pers.tel:=pcoda^.tel;
pt:= pcoda;
pcoda:= pcoda^.next;
if pcoda=NIL then ptesta:=NIL;
end
end;

procedure legkip(var ps:gente);
var nome:string[15];
tel:string[12];

begin
write('nome da inserire: ');
readln(nome);
write ('telefono: ');
readln(tel);
ps.nome:=nome;
ps.tel:=tel;
end;

procedure stampa(pers:gente);
begin
writeln('persona eliminata: ');
writeln('nome: ',pers.nome);

```

```

        writeln('tel: ',pers.tel)
end;

procedure stampaCoda(pcosa:punt);
var pt:punt;
    nm:string[15];
    tl:string[12];
    j:integer;
begin
    j:=1;
    pt:=pcoda;
    while pt<>NIL do
        begin
            nm:=pt^.nome;
            tl:=pt^.tel;
            writeln('Elemento # ',j,' della coda: ');
            writeln(pt^.nome);
            writeln(pt^.tel);
            writeln(' ooooooooooooo ');
            j:=j+1;
            pt:=pt^.next;
        end;
    end;

begin
clrscr;
ptesta:=NIL;
pcoda:=NIL;
writeln(' =====');
writeln(' ----- GESTIONE DI UNA CODA DI CLIENTI -----');
writeln(' =====');
writeln(' ');
inizio(ptesta,pcoda,fp);
repeat
    writeln('scegli una opzione: ');
    writeln('1 - inserimento nella coda');
    writeln('2 - estrazione dalla coda');
    writeln('3 - stampa coda');
    writeln('4 - fine');
    readln(opz);
    case opz of
        1:begin
            legkip(persona);
            insert(ptesta,pcoda,persona)
        end;
        2:begin
            extract(ptesta,pcoda,persona);
            stampa(persona);
            writeln(' <-----> '); writeln
        end;
        3:begin
            stampaCoda(pcosa);
            writeln
        end;
        end;
until opz= 4;
salva(pcosa,fp);
close(fp);
end.

```

Operazioni di I/O

Il Pascal rende possibile la realizzazione di programmi che possono inviare dati verso periferiche esterne (output) o ricevere da esse dei dati in ingresso (input).

Viene denominato col termine *porta* un canale attraverso cui il calcolatore si collega con il mondo esterno. Si parla di porta di input per la lettura dei dati e di porta di output per l'uscita dei dati. Le porte sono caratterizzate da un indirizzo solitamente espresso in notazione esadecimale. Per accedere alle porte in Pascal sono stati implementati due array predefiniti: **Port** e **PortW**, unidimensionali, i cui elementi corrispondono ai dati letti o da inviare in uscita (di tipo byte per Port e di tipo word per PortW), e il cui indice è rappresentato dall'indirizzo della porta.



Operazione di INPUT:

dato := Port [indirizzo]; il dato ricevuto viene assegnato alla variabile *dato*



Operazione di OUTPUT:

Port [indirizzo] := dato; il dato presente nella variabile *dato* viene inviato in uscita

Per esprimere l'indirizzo in esadecimale si utilizza il carattere "\$". Ad esempio:

```
dato := $0A;           {assegna alla variabile dato il valore 0AH  
                      (esadecimale)}
```

```
Port [$378] := dato   {invia alla porta parallela il dato preparato}
```

378 è l'indirizzo della porta parallela dove viene collegata la stampante per i dati in uscita.

È possibile dunque interfacciare il calcolatore con dispositivi hardware e comandarli direttamente con un programma scritto in Pascal. Si può, ad esempio, comandare dei relè, accendere dei LED o pilotare motori passo-passo attraverso la porta parallela; acquisire dati da strumentazione elettronica attraverso la porta seriale; interfacciarsi con apposite schede inserite nel computer.

Il seguente programma di esempio genera un segnale ad onda quadra sulla linea D0 della porta parallela; la durata del periodo può essere cambiata utilizzando i tasti '+' e '-'. L'onda quadra viene anche visualizzata in modo approssimativo sullo schermo:

```
program onda_quadra;  
uses crt;  
var temp: word;  
    ch: char;  
    h, i: integer;  
  
begin  
  clrscr;  
  writeln( 'Premere [+] per aumentare il periodo');  
  writeln( 'Premere [-] per diminuire il periodo');  
  writeln( 'Premere [ESC] per uscire');  
  temp:=501;
```

```
repeat
h:=(temp div 10)+1;
port[$378]:=0;
for i:=1 to h do
write('_');
delay(temp);
port[$378]:=1;
for i:=1 to h do
write('-');
delay(temp);
if keypressed then ch:=readkey;
if (ch= '+') and (temp < 2000) then
temp:= temp+10;
if (ch= '-') and (temp >10) then
temp:= temp-10;
until ch = #27;  {ripete finchè il tasto premuto è quello di ESC}
end.
```

INDICE

Caratteristiche del linguaggio Pascal	2
Tipi di dati	3
I tipi predefiniti	4
Procedure di immissione e visualizzazione dati	6
Strutture di controllo	7
Tipi di dati strutturati	13
Tipo Array	13
Tipo Record	15
Tipo set	19
Programmazione: il problema dello scambio	21
Algoritmi di ordinamento	22
Semplice algoritmo di ordinamento	22
Bubble Sort	25
Funzioni e procedure	27
Le procedure	30
Visibilità delle variabili	32
La ricorsione	35
Ordinamento quicksort	37
Funzioni e procedure per il trattamento di stringhe	40
Procedure di uscita e di terminazione	44
Il tipo file	45
Le Unit	52
La unit CRT	58
La unit graph	61
I puntatori	73
Operazioni di I/O	82