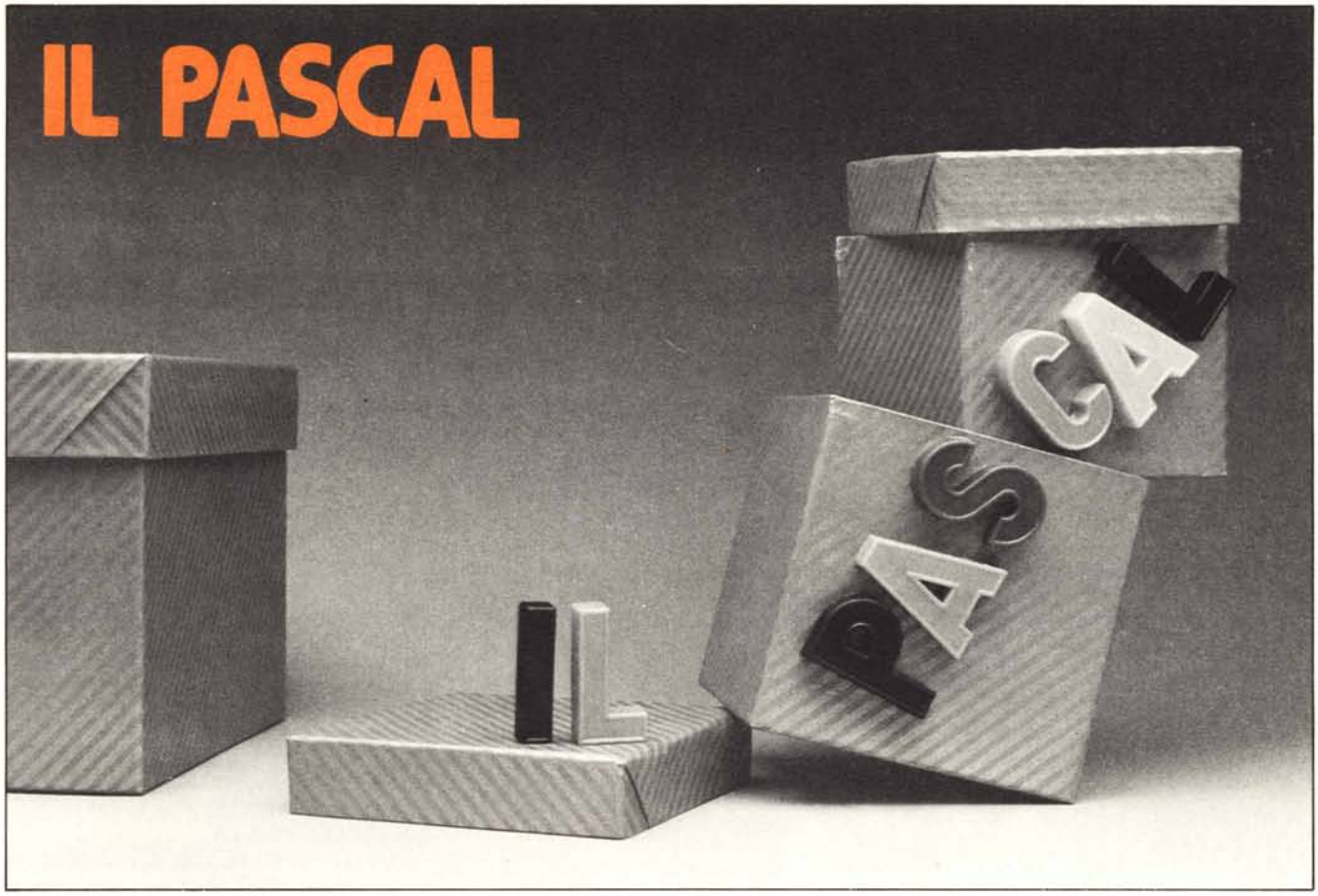
IL PASCAL



Parte terza
I tipi strutturati
record e *pointer*

Questa parte della breve e sommaria esposizione del PASCAL che stiamo portando avanti ormai da tre mesi si occupa di due soli elementi del linguaggio: i tipi strutturati record *e* pointer. *Riteniamo che la loro importanza sia tale da meritare un capitolo a parte e di conseguenza un'esposizione più dettagliata.*

Ho dovuto recentemente trasportare in cantina un certo quantitativo di bottiglie vuote (anche gli articolisti possono essere felicemente sposati) delle dimensioni più diverse, dalla piccola Peroni ai due litri di vino; e mi sono trovato in un grosso guaio con le casse di acqua minerale che mi ero procurato: le bottiglie più grosse non entravano, quelle più piccole scappavano fuori, insomma è stato un disastro. Ho rimediato con le normali e volgari buste di plastica del supermercato, ed ho pensato che avevo sbagliato struttura: un *array* non può contenere dati di tipo diverso senza dover fare i salti mortali, ci vuole invece una struttura più flessibile.

I linguaggi ad alto livello più in uso (FORTRAN e BASIC) forniscono al programmatore soltanto strutture molto rigide rispetto al tipo di dato che possono contenere, e anche i tipi strutturati del PASCAL visti nella scorsa puntata, seppur rivoluzionari nella loro forma, hanno questo difetto di contenuto. Una variabile di tipo *set* definita su un insieme di caratteri potrà sì subire tutte le operazioni sugli insieme che vogliamo, ma non potrà mai e poi mai manipolare sia caratteri che numeri interi, per non parlare di strutture più complesse.

Il problema di mescolare nella stessa struttura dati diversi fra loro è stato affrontato fin dai tempi del FORTRAN (gli assemblatori in questo senso non danno problemi) e risolto con modalità farraginose, ad esempio facendo di ogni variabile una matrice a due colonne di cui una conteneva il dato e l'altra un parametro che specificava se il dato era da interpretarsi come un numero reale, o due numeri interi, o quattro caratteri ASCII, e così via.

L'unico dei linguaggi classici che permetta strutture di tipo misto è il COBOL, ma anche in questo caso vengono poste certe limitazioni che impediscono una strutturazione avanzata.

Il PASCAL trae invece buona parte della sua potenza dal fatto di poter strutturare in una sola unità - e quindi indirizzare con una sola variabile - dati di tipo differente, anche in vari livelli gerarchici, fino a costruire architetture di dati molto complesse: fra i tipi che compongono la struttura "mista" vi possono essere anche delle strutture miste, che possono a loro volta contenere altre strutture miste, e così via.

Il tipo strutturato capace di compiere questa notevole operazione prende il nome di *record*, e la sua struttura generale è la seguente:

```
record   nome 1, nome 2,... : tipo 1
         nome 3, nome 4,... : tipo 2
         nome N-1, nome N,... : tipo N
end;
```

Fra *record* e *end* troviamo una serie di variabili raggruppate secondo un ordine a scelta di chi costruisce il record: ad ognuna di queste variabili possono poi essere assegnati dei valori secondo modalità che vedremo, ma si potrà sempre indirizzare il record nel suo complesso.

Vediamo subito un piccolo esempio pratico:

```
type COMPLEX = record REALE, IMAG : real
                    end;
```

Questa struttura definisce un numero complesso secondo la classica dizione matematica, ossia come *coppia ordinata di numeri reali;* il *record* è infatti composto di due variabili reali REALE e IMAG, che contengono rispettivamente la parte reale e la parte immaginaria del numero.

Se ora dichiariamo una variabile come complessa:
var ZETA : COMPLEX;
ogni volta che faremo riferimento a ZETA faremo riferimento alla *coppia* di variabili reali REALE e IMAG associate alla variabile complessa ZETA.

Ma gli elementi del record sono anche accessibili separatamente, per poterli assegnare e modificare. Ciò si ottiene specificando il particolare elemento della variabile di tipo record cui si vuole accedere, nel seguente modo:
ZETA . REALE : = 1;
ZETA . IMAG : = 2;

Abbiamo così assegnato alla variabile ZETA il valore (1 + 2i). Gli elementi di tipo *non strutturato* così puntati possono essere usati in ogni tipo di calcolo, come nel seguente esempio:
var ZETA, WU, SUM : COMPLEX;
ZETA . REALE : = 1;
ZETA . IMAG : = 2;
WU . REALE : = 3;
WU . IMAG : = 4;
SUM. REALE : = ZETA . REALE + WU . REALE;
SUM. IMAG : = ZETA . IMAG + WU . IMAG;

Questo piccolo programma assegna alla variabile complessa SUM il valore pari alla *somma* (secondo le regole dei numeri complessi) delle variabili complesse ZETA e WU.

L'esempio del numero complesso non è però significativo, perché il record da noi definito si limita a raggruppare variabili dello stesso tipo, e questo può farlo anche una normale *array*: semmai è comodo poter chiamare le variabili con il *nome* di reale e immaginaria anziché con dei numeri, ma anche questo scoglio può essere superato dimensionando l'*array* con un tipo *scalare* appositamente definito.

Riscriviamo a riprova l'ultimo esempio senza usare la struttura *record*:
type COMPLEL = (reale, imag);
COMPLEX = *array* COMPLEL *of* reale;
var ZETA, WU, SUM : COMPLEX;
ZETA [reale] : = 1
(etc)
SUM [reale] : = ZETA [reale] + WU [reale] ;
SUM [imag] : = ZETA [imag] + WU [imag] ;

Come si vede il risultato è identico, ed abbiamo guadagnato in compattezza.

Dove invece la struttura in *record* mostra tutta la sua potenza è quando gli elementi del suo interno sono di tipo diverso, come ad esempio:
type CONVASC = *record* CAR : char;
ASC : integer
end;
var ALFA : CONVASC;

Secondo quanto visto poco sopra, la variabile ALFA è definita come *record* formato da una parte di tipo carattere e una di tipo intero. Se ora selezioniamo un unico elemento di ALFA avremo in un caso una variabile di tipo *char*, e nell'altro una variabile di tipo intero:
ALFA . CAR : = 'A' ;
ALFA . ASC : = 65 ;

Con la prima operazione abbiamo lavorato sui caratteri e con la seconda sui numeri interi, facendo sempre riferimento alla stessa variabile ALFA: nessuna *array* può realizzare quest'operazione.

Si ricordi poi che il *record* è un tipo come tutti gli altri, e può essere usato in tutte le definizioni di tipo come "mattone" per costruire strutture più complesse. Il prossimo esempio mostrerà come si può usare una *array di records* per eseguire con sveltezza la conversione tra i caratteri ASCII e il loro valore numerico:
var ASCINT : *array* 0..127 *of* CONVASC;
I : integer;
for I : = 0 *to* 127 *do*
begin ASCINT [I] . ASC = I;
ASCINT [I] . CAR = chr (I)

end

Con questo programma abbiamo costruito il vettore rappresentato in fig. 1). Ogni elemento di questo vettore è composto da una parte di tipo carattere e una parte intera, e la struttura contiene i 128 caratteri ASCII ed i loro codici numerici. Per ottenere - senza usare l'apposita funzione ord(ch) - il carattere ASCII corrispondente ad un numero intero basterà, puntarsi all'elemento giusto del vettore.

Come si è detto, un record può inserirsi in qualsiasi struttura più complessa: un'*array* nell'esempio appena visto, ma anche *un altro record*. In questo modo si ottengono *records nidificati*.

Fig.1

CAR	ASC	
CAR	ASC	0
CAR	ASC	1
CAR	ASC	2
		.
		.
		.
		.
		.
		.
		.
CAR	ASC	127

Vediamo il seguente esempio:
type DATA = *record* giorno : 1..31;
mese : (jan, feb, mar, apr, may, jun, jul, sep, oct, nov, dec);
anno : integer

end;

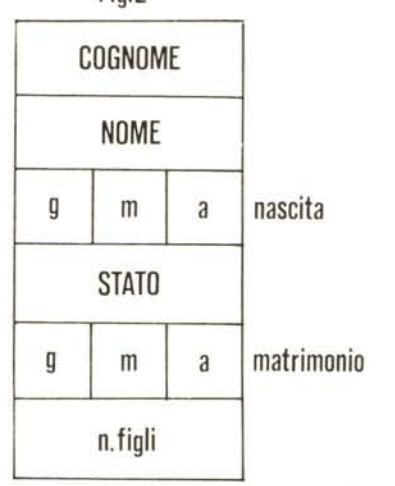
Questo *record* è diviso nei tre campi (giorno, mese e anno) che formano una data. Come si vede, il campo "giorno" è di tipo *subrange*, cosicché il compilatore segnerà un errore ogni volta che si tenterà di assegnare un giorno maggiore di 31; mentre il campo "mese" è addirittura di tipo scalare, per assegnare ai mesi direttamente il loro nome; infine il campo "anno" è un numero intero.

Inseriamo ora questa struttura in un altro record più grande, ad esempio in una collezione di dati anagrafici riguardanti una persona:
type STRINGA = *array* 1..16 *of* char;
type PERSONA = *record* COGNOME, NOME: STRINGA;
NASCITA : DATA;
STATO : (cel, nub, con, div, ved);
MATRIM : DATA;
N FIGLI : integer

end;

La struttura di questo record è rappresentata in fig. 2).

Fig.2



In questa architettura sono raggruppati alcuni dati riferentisi ad una persona; innanzitutto il cognome e il nome, che sono variabili di tipo STRINGA. Questo ultimo è stato precedentemente definito come array di caratteri a dimensione 16; si prevede dunque che nomi e cognomi delle persone descritte da questa struttura non saranno più lunghi di 16 caratteri.

Segue la data di nascita, dichiarata tramite il record DATA: sarà pertanto suddivisa in giorno, mese e anno come abbiamo visto sopra.

Lo stato civile è una variabile di tipo *scalare*, e distingue le persone in celibi, nubili, coniugati, divorziati e vedovi. Per gli ultimi tre tipi di stato civile sono significativi i successivi campi, ossia la data del matrimonio e il numero dei figli: il primo è nuovamente un record DATA, mentre il secondo è un numero intero.

Come al solito, se vogliamo *definire* una variabile di tipo PERSONA ed assegnarvi dei valori, dovremo agire in questo modo:

```
var PER : PERSONA;
PER . COGNOME : = 'Hasenmajer';
PER . NOME : 'PIETRO';
PER . NASCITA : = .....
```

Qui ci troviamo impicciati, poiché PER . NASCITA è una variabile di tipo record, e non può esservi assegnato direttamente un valore. Ma niente paura: le regole per l'assegnamento dei valori ai campi dei record continuano a valere, e possiamo procedere in questo modo:

```
PER . NASCITA . GIORNO : = 7;
PER . NASCITA . MESE : = jan;
.....
```

e così via: ad ogni livello ci portiamo appresso la traccia di come ci siamo arrivati.

Analogamente, per la data di matrimonio:

```
PER . MATRIM. GIORNO : = 3;
.....
```

In questo modo si evita ogni ambiguità.

È tuttavia chiaro che l'assegnamento dei valori ad un record diventa pesante se ogni volta occorre ripetere la catena delle variabili, specie quando il record stesso è complesso e nidificato. Per sveltire gli assegnamenti il PASCAL prevede una particolare istruzione chiamata *with*, che permette di sottintendere il nome della variabile o di un campo a livello superiore.

Si può quindi assegnare il record dell'ultimo esempio in questo modo:

```
with PER do
  COGNOME : = 'Hasenmajer';
  NOME : 'PIETRO';
  NASCITA . GIORNO : = 7;
.....
end;
```

L'istruzione *with* può essere usata a qualsiasi livello nella struttura *record*; è perciò valido scrivere:

```
with PER. MATRIM do
GIORNO : = 3
MESE : = jun
ANNO : = 1978
end;
```

oppure si possono nidificare i *with*:

```
with PER do
```

Fig.3


```
COGNOME : = '.....';
NOME : = '.....';
with NASCITA do
    GIORNO : = .....;
    MESE : = .....;
    ANNO : = .....;
end;
STATO : .....
end;
```

L'istruzione *with* è dunque utilissima per sveltire e compattare l'assegnamento dei campi di una struttura record a uno o più livelli.

Riprendiamo ora la struttura generale del *record* PERSONA: abbiamo detto che i campi relativi al matrimonio non sono significativi per le persone celibi e nubili: è quindi un peccato sprecare spazio di memoria per il gusto della compatibilità: d'altronde per queste pesone (celibi e nubili) possono essere importanti altri dati, ad esempio una variabile logica che specifichi se vivono da soli o a carico della famiglia, cosa che invece non interessa gli sposati.

Il PASCAL prevede anche questo: un *record* può essere formato da una parte *fissa* e da una parte *variabile* che dipende da uno dei parametri per mezzo di un'istruzione *case*.

Si può quindi modificare la definizione del record PERSONA in questo modo:

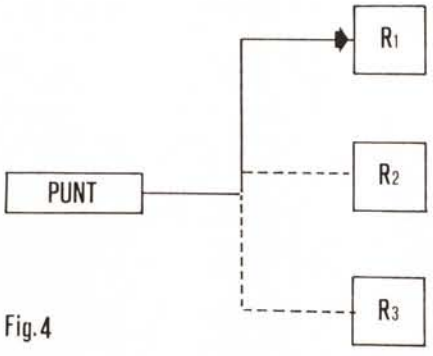


Fig.4

```
type PERSONA : = record COGNOME,
NOME : STRINGA;
NASCITA : DATA;
case STATO : (cel, nub, con, div, ved,) of
cel, nub : INDIP : booleam;
con, div, ved : MATRIM : DATA;
N FIGLI : integer

end

end
```

Questa scrittura diversifica l'ultima parte del *record*, a seconda che la persona a cui si fa riferimento abbia o no contratto un matrimonio. A questo punto la fig. 2) mostra il record PERSONA per una persona sposata, mentre il record PERSONA per gli scapoli e le nubili è rappresentato in fig. 3).

Si noti che la parte variabile deve sempre trovarsi *in fondo* al *record*: questa è una delle poche regole ferree del PASCAL. Tuttavia poiché un *record* può contenere atri *record*, potremo benissimo trovarci delle parti variabili in mezzo ad una struttura in quanto terminanti un *record* di livello inferiore.

La struttura *record* si rivela dunque la più complessa e completa architettura di dati che il linguaggio possa fornire, ed è ideale per costruire ad esempio degli archivi contenenti i dati anagrafici di tutti gli abitanti di una città.

In questo caso ci troviamo di fronte ad un nuovo problema: quello della *staticità* delle strutture di dati esaminate finora. Che siano usati o no, agli elementi descritti da queste strutture viene riservato, all'atto della dichiarazione, una certa area di memoria che rimane fissa e immutabile nei secoli: se dichiariamo una variabile come *array* [1..10000] *of* integer, verranno riservate subito diecimila parole di memoria per questa variabile, anche se ci vorranno poi degli anni perché tutte e diecimila vengano usate. Se l'*array*, invece di essere composta da numeri interi, fosse composta da *records* di tipo PERSONA, il problema si moltiplicherebbe di conseguenza.

La gestione dei dati anagrafici degli abitanti di città come Milano o Roma richiederebbe di dimensionare la struttura di dati (ad esempio un'*array* di *records*) sul numero di abitanti previsto, e bisognerebbe anzi *sovradimensionare* il vettore per mettersi al riparo da spiacevoli sorprese: per Roma occorrerebbero più di quattro milioni di elementi, e non so se la situazione sarebbe molto sostenibile anche per il calcolatore più grande in commercio. Vi è poi un'altra difficoltà: l'accesso a questi dati, sia in termini di ricerca, sia in termini di gestione una volta trovato il nome giusto. Si pensi che il trasferimento di un intero

record dalla struttura di archivio ad una variabile di appoggio richiede un numero di operazioni pari al grado di suddivisione del record stesso: e l'istruzione *with* sveltisce soltanto la scrittura software, ma non accelera per nulla il tempo di elaborazione.

Quanto alla ricerca del dato giusto in mezzo a qualche milione di *record* tutti uguali, abbiamo esposto in altra sede le grandi difficoltà in cui si incorre con le strutture *array* e *file* non appena inizia a crescere il numero dei dati in esse contenuti: anche la macchina più veloce può andare in "tilt" per qualche ora se non è fornita di un intelligente algoritmo di ricerca.

Tutto ciò ha portato a un certo qual "ritorno all'antico" (come avviene in molti altri campi della vita), rispolverando un vecchio concetto che è il punto di forza dei linguaggi assemblatori: *l'indirizzamento indiretto*.

type nome = ↑ tipo;

ove "tipo" è un tipo qualsiasi anche non ancora dichiarato. Solitamente è una struttura *record*, poiché soltanto essa può contenere, oltre ai dati, il puntatore al prossimo elemento.

Per fare un esempio, aggiungiamo al *record* PERSONA un puntatore, per poter costruire una struttura a lista:

```
type PERSPT = ↑ PERSONA;
PERSONA = record COGNOME ecc...
NEXT : PERSPT;
case ecc...
end

end;
var LISTA, PTR : PERSPT;
```

Abbiamo inserito nella *parte fissa* del record PERSONA una variabile di tipo puntatore, ed abbiamo anche dichiarato altri due puntatori come variabili di appoggio (LISTA e PTR).

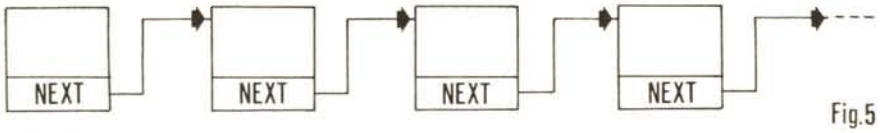


Fig.5

Supponiamo di avere un certo numero di strutture *record* e di volervi accedere tramite una variabile di appoggio: questa variabile sia una sola parola di memoria che contiene l'*indirizzo* del *record* cercato. A questo punto, per cambiare *record*, basta cambiare soltano il valore di questa variabile (che si chiama *puntatore*), e subito è accessibile un altro dato comunque strutturato, senza doverne trasferire tutti i campi da un posto all'altro della memoria.

La fig. 4) mostra il funzionamento di un puntatore: agendo su una sola parola di memoria possiamo "vedere" un gran numero di dati anche molto complessi.

Resta il problema di come trovare l'indirizzo del dato cercato: questo scoglio può essere superato includendo delle variabili di tipo puntatore nella *struttura record stessa*; una di queste variabili può essere caricata con l'indirizzo del prossimo record, e si può così gestire facilmente il puntatore di appoggio copiandovi questo dato per accedere al prossimo elemento.

La struttura descritta sopra si chiama *lista*, ed è rappresentata in fig. 5). Oltre all'uso dei puntatori, il PASCAL permette anche una gestione dinamica della memoria, nel senso che, data una variabile di tipo puntatore, un'apposita funzione può generare un dato nuovo e depositare nel puntatore il suo indirizzo. Questo permette di allocare la memoria nel corso del programma soltanto quando vi è bisogno, senza crearsi problemi di sovradimensionamento delle strutture.

Ma vediamo nei particolari la scrittura PASCAL delle variabili di tipo puntatore e il loro impiego nei programmi.

La struttura generale del tipo "*pointer*", o puntatore, è la seguente:

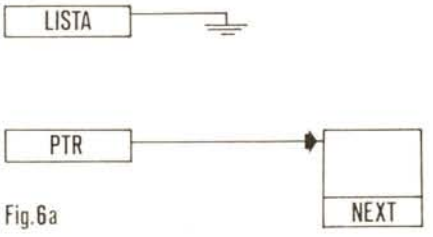


Fig.6a

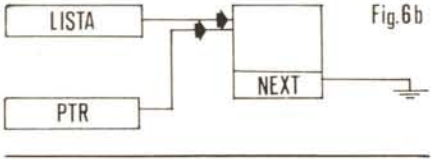


Fig.6b

Come per accedere ai vari campi del record si usa il *punto*, per accedere al dato indirizzato da un puntatore si usa la *freccia*; così LISTA è una variabile di tipo puntatore, mentre LISTA ↑ è una variabile di tipo record; infine LISTA ↑. NOME è una stringa di 16 caratteri, e LISTA ↑. NEXT è di nuovo un puntatore (e allora si può ripetere il ciclo scrivendo LISTA ↑. NEXT ↑ che è di nuovo un record, e così via).

Con questi soli elementi in mano, senza cioè aver dichiarato alcun effettivo record, possiamo costruire i dati per tutta la popolazione del mondo (sempre compatibilmente con la memoria dell'elaboratore a nostra disposizione). Infatti il PASCAL, come abbiamo detto, permette di creare *dinamicamente* le strutture mediante una apposita funzione, che prende il nome di *new*.

Inizializziamo dunque la nostra struttura:

```
LISTA : = 0;
new (PER);
```

Questa situazione è rappresentata in fig. 6a): la variabile LISTA contiene un indirizzo nullo, che rappresenterà da ora in poi la fine della lista; invece il puntatore PTR è legato ad un record creato tramite la funzione *new*.

Si può ora riempire l'elemento appena creato con i dati relativi ad una persona:

```
with PTR ↑ do
  COGNOME : = .....;
  .....

end;
```

(Notare come si acceda al record sempre con la scrittura "puntatore-freccia").

Per appendere ora il record alla lista (che finora è vuota), si esegue qualche operazione di trasferimento fra i puntatori;

```
PTR ↑ . NEXT : = LISTA
LISTA : = PTR;
```

Con la prima operazione abbiamo "legato" il nuovo record alla cima della lista, con la seconda abbiamo spostato il puntatore LISTA al nuovo elemento: la situazione è riprodotta in fig. 6b).

Possiamo ora liberare il puntatore PTR creando magari un nuovo dato:

```
new (PTR);
```

e ripetere il ciclo di riempimento ed allocazione nella lista.

Una volta che la lista è stata formata, è possibile svolgervi delle ricerche servendosi sempre di una variabile di appoggio PTR.

Per *accedere* alla lista basta posizionare questa variabile al suo inizio:

```
PTR = LISTA;
```

e per *spostarsi* lungo di essa basta eseguire questa istruzione;

```
PTR : = PTR ↑ NEXT;
```

Se per caso arriviamo in fondo alla lista prima di aver trovato l'elemento cercato, ce ne accorgiamo perché PTR assume il valore zero (fine della lista).

Provate — come esercizio — a scrivere le istruzioni per inserire un nuovo elemento a metà della lista, ad esempio dopo un certo elemento puntato da una variabile di appoggio, e viceversa per estrarre un elemento dalla lista (nota: per quest'ultima operazione occorrono due puntatori). Nella prossima puntata daremo le soluzioni di questi problemini.

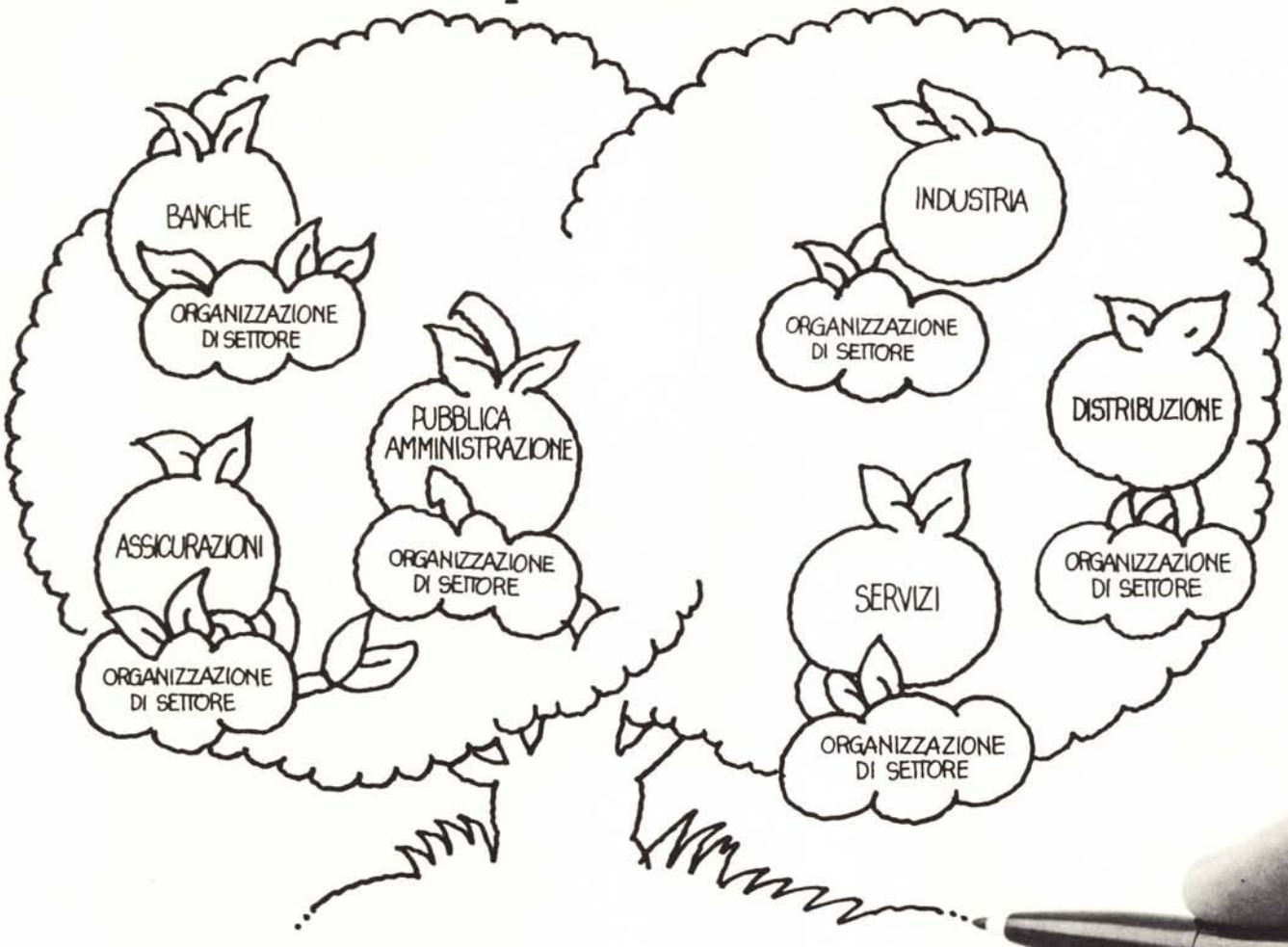
Conclusione

Con i tipi record e puntatore si conclude la rassegna delle strutture di dati del PASCAL. Ci son volute tre puntate, e questo è indicativo di quanto sia estesa e articolata la gestione di dati in questo linguaggio. Dal prossimo numero inizieremo a vedere le istruzioni, che, seppure in numero minore dei tipi di variabili, offrono ampie possibilità di controllo dei cicli e soprattutto di strutturazione a blocchi: ricordiamoci sempre che questo è l'orientamento fondamentale del PASCAL.

Pietro Hasenmajer

"I problemi non sono uguali per tutti."

Ma non per tutti è cosí ovvio.



ATA-Univas

La Honeywell è l'unica azienda di informatica che si è data una struttura di marketing in grado di affrontare i problemi specifici di ogni specifico segmento di mercato e di risolvere cosí le precise esigenze di ogni cliente.

Le imprese industriali, ad esempio.

Produzione, magazzino, gestione degli ordini, gestione finanziaria sono alcuni fra i principali problemi che le imprese industriali affrontano.

Proprio in queste aree, la metodologia informatica si è rivelata lo strumento piú economico ed efficiente. La HISI, con la molteplicità delle sue esperienze, è in grado di proporre non solo le apparecchiature piú idonee ma anche uomini, idee, e soluzioni applicative piú avanzate, perfettamente aderenti alle specifiche necessità.

Honeywell

Honeywell Information Systems Italia

La conoscenza a monte della soluzione.

sistemi informatici *innovativi*

ATARI 800

Il più completo personal computer.
Grafica, colore e sintesi musicale rivoluzionarie comprese nell'unità base. 128 variazioni di colore (16 colori in 8 livelli di luminosità).

Apple III

Dal grande successo dell'Apple II il nuovo potente personal per professionisti e manager esigenti.
Analisi finanziarie, budgets, previsioni e simulazioni, preparazione testi e calcolo.

ONYX C 8000

Decisamente non è un personal... È parente del personal soltanto nel prezzo.
Memoria RAM da 64 a 1024 Kb. fino a 16 posti di lavoro - memoria di massa su dischi Winchester espandibile da 10 a 320 Mb - Unità a nastro magnetico da 12 Mb per le copie di sicurezza. Sistemi operativi Multitask MOASIS ed UNIX. Collegabili in rete locale.

ZENITH Z89

Un raffinato e potente personal computer ma anche efficiente elaboratore gestionale per la piccola azienda.
Basato sullo Z 80 con 64 Kb, floppy da 5" e 8". Sistemi operativi HDOS, CP/M e PASCAL UCSD.

L'ONYX è stato progettato appositamente per la gestione razionale della media azienda.

Nei propri centri di vendita in Torino e Milano la SOFTEC mette a disposizione dei clienti:

- sale per dimostrazione e prova sistemi;
- completa assistenza tecnica;
- seminari e corsi di istruzione;
- programmi standard gestionali, professionali ed hobbystici;
- magazzino parti di ricambio e accessori.

servizio assistenza rivenditori ATARI - ZENITH - ONYX

10124 TORINO
C.so San Maurizio, 79
Tel. (011) 8396444 (5 l.)

20155 MILANO
Via G. Govone, 56
Tel. (02) 3490231 - 3490367

10015 IVREA
Via delle Miniere, 4
Tel. (0125) 43673

Importante!!!
La SOFTEC cambia la sede di Milano.
NEI NUOVI UFFICI saranno a disposizione dei clienti e rivenditori, grandi sale per la dimostrazione, vendita e assistenza di: **APPLE II, ATARI - ZENITH - ONYX**
Nuovo indirizzo dal 20-5-1981:
MILANO Viale Majno, 10 Tel: 702320 708916 783627

informatica
SOFTEC

Agente ADVEICO per il Piemonte, Lombardia e Liguria

Desidero ricevere maggiori informazioni sui seguenti sistemi: ☐ ONYX
☐ Apple II ☐ Apple III ☐ ATARI ☐ ZENITH
Riservato ai rivenditori ☐ **ATARI** ☐ **ZENITH** ☐ **ONYX**
Ci interessa rivendere:
nome .....
indirizzo .....
Telefono .....
Città .....