di Sergio Polini

Valori e indirizzi

Variabili globali e variabili locali, parametri-valore e parametri-variabile, stack e heap: sono tutti concetti di estrema importanza. Un discorso teorico e fine a se stesso rischierebbe di risultare ridondante rispetto a quanto potete trovare sul manuale, su qualsiasi libro dedicato al Pascal, su altri numeri della rivista; vi propongo quindi un approccio diverso: ci soffermeremo su quegli argomenti per illustrare l'uso degli «inline statement» e della dichiarazione «external», riprendendo da dove ci eravamo lasciati la volta scorsa. Avremo anche modo di descrivere brevemente la gestione della memoria operata dal Turbo Pascal, preparando così il terreno per il tema del mese prossimo, l'allocazione dinamica della memoria. Una discussione sull'uso di routine in linguaggio macchina rischia però di allargarsi a macchia d'olio. Dobbiamo ovviamente imporci un limite; parleremo quindi solo di procedure e di funzioni che ritornano un intero, dal momento che per le funzioni sarebbe troppo lungo considerare tanti casi quanti sono i possibili tipi del loro risultato. Non è un gran danno, sia perché si può quasi sempre usare una procedura con un parametro variabile invece di una funzione, sia perché in fondo non faremo altro che evitare di ripetere quanto è illustrato a pagina 224 del manuale

La volta scorsa abbiamo visto alcuni esempi di uso delle risorse offerte dall'hardware di un PC, dal BIOS e dal DOS. Abbiamo anche detto che non sempre è opportuno usare le procedure Intr e MsDos per chiamare gli interrupt del DOS. Quando si chiama un interrupt di norma vengono salvati nello stack i registri CS, IP e i flag; alla fine vengono ripristinati tutti e tre togliendoli dallo stack con una istruzione IRET. Vi sono però pochi interrupt che si comportano diversamente: lasciano infatti i flag nello stack perché settano il «carry flag» se si è verificato un errore, al fine di consentire al programma che era stato interrotto di accorgersi che non tutto è andato come previsto (se i flag venissero ripristinati il valore del carry sarebbe quello precedente la chiamata dell'interrupt). Le procedure Intr e MsDos si aspettano un comportamento «normale» e non consentono quindi di leggere il carry. Per ovviare a questo problema basta scriversi una breve routine in Assembler e trasformarla in un «inline statement» o in una procedura «external»; vedremo tra breve come procedere, dopo aver esaminato da vicino come il compilatore traduce in linguaggio macchina un programma Pascal.

Segment e Offset

Normalmente un file COM occupa solo 64K e i registri CS, DS e SS hanno tutti lo stesso valore. Il Turbo Pascal (e i programmi compilati con questo) possiedono però una routine di inzializzazione che assegna a DS l'indirizzo di un «data segment» immediatamente successivo all'area occupata dal codice eseguibile e a SS quello di un'area per lo stack collocata alla fine della RAM. In questo modo non solo si dispone di 64K per il codice PIÙ 64K per i dati, ma soprattutto si può usare per lo heap (e quindi per le variabili dinamiche) tutta la memoria compresa tra il data segment e lo stack.

Nel code segment ci sono naturalmente le istruzioni, nel data segment le variabili globali. Quando un programma viene compilato, per ogni variabile globale viene riservato uno spazio nel data segment; successivamente, ogni volta che il compilatore trova nel sorgente il nome di una variabile globale, lo traduce in un numero di due byte che rappresenta l'offset della variabile nel data segment, cioè l'indirizzo di quello spazio espresso come distanza dall'origine del data segment. Questo indirizzo non cambia mai, lo spazio riservato per le variabili globali «esiste sempre», anche se poi il programma non ne fa alcun uso. Più o meno lo stesso accade per le costanti tipizzate: la differenza è che l'indirizzo si riferisce al code segment.

Le variabili locali e i parametri hanno abitudini diverse: nascono quando viene chiamata la procedura in cui sono definiti, muoiono quando questa termina; risiedono infatti, quando sono «in vita», nella parte della memoria dinamica riservata allo stack. Si parla di memoria dinamica perché lo stack funziona un po' «a fisarmonica», nel senso che si allarga quando servono nuove variabili locali e/o nuovi parametri, si restringe quando possono essere scartati.

Il code segment, il data segment e lo stack occupano al massimo 64K ognuno; ogni indirizzo a loro relativo può quindi essere espresso con un numero di due byte (da 0 a 65535). Lo heap però può occupare anche qualche centinaio di Kbyte; per l'indirizzo di una variabile dinamica due byte non bastano più, e l'architettura dell'8086/8088 costringe ad usarne quattro: due per un «segment» (generica area di 64K) e due per un «offset» (distanza dall'origine di quest'area).

Variabili, indirizzi e parametri

Consideriamo una normale assegnazione, ad esempio «a:=b». Se «a» e «b» sono ambedue variabili posso anche scrivere «b:=a», ma la simmetria è meno completa di quanto sembrerebbe. In tutti e due i casi rendo il valore di una variabile uguale al valore di un'altra, ma il compilatore opera una sottile distinzione tra quello che compare a sinistra e quello che compare a destra dell'operatore di assegnazione: il VALORE della variabile di destra viene posto nell'INDIRIZZO di quella di sinistra. Per

```
program Nulla;
var
  Glb11, Glb12, Glb13: integer;
function Pippo(Vlr: integer; var Vrbl: integer): integer;

                                { cccc:2D9F PUSH BP          }
                                {           MOV  BP,SP       }
var                             {           PUSH BP          }
  Lcl: integer;                 {           SUB  SP,2        }
begin
  Lcl := Vlr;                   {           MOV  AX,[BP+8]   }
                                {           MOV  [BP-4],AX   }
  Glb11 := Vlr;                 {           MOV  [0260],AX   }
  Vrbl := Lcl;                  {           LES  DI,[BP+4]   }
                                {           MOV  AX,[BP-4]   }
                                {           MOV  ES:[DI],AX  }
  Pippo := 1                    {           MOV  AX,1
                                {           MOV  [BP+0A],AX  }
                                {           MOV  AX,[BP+0A]  }

end;                            {           MOV  SP,BP       }
                                {           POP  BP          }
                                {           RET  8           }
begin
  Glb13 := Pippo(9,Glb12)       { cccc:2DD5 SUB  SP,2        }
                                {           MOV  AX,9        }
                                {           PUSH AX          }
                                {           MOV  DI,0262     }
                                {           PUSH DS          }
                                {           PUSH DI          }
                                { cccc:2DE0 CALL 2D9F        }
                                { cccc:2DE3 MOV  [0264],AX   }
end.                            {           CALL 0C89        }
```

Figura 1 - Esempio di traduzione operata dal compilatore dal sorgente in Assembler (cccc = code segment; i numeri sono in esadecimale). Rispetto al codice effettivamente prodotto dal Turbo Pascal è stata operata qualche semplificazione.

apprezzare l'importanza della distinzione basta pensare che anche una espressione come «2+2» ha un valore, ma certo non ha un indirizzo. In altri termini, SI USA il valore di una variabile o di un parametro o di una espressione, SI MODIFICA una variabile attraverso il suo indirizzo.

Sappiamo che i parametri compresi nella dichiarazione di una procedura sono detti parametri formali, quelli effettivamente passati alla procedura quando viene chiamata, sono detti parametri effettivi. Sappiamo anche che ad una procedura possiamo passare sia parametri-valore che parametri-variabile. I primi rappresentano solo dei VALORI, i secondi sono invece INDIRIZZI di variabili. Nel primo caso viene posto nello stack il valore del parametro effettivo (e infatti questo può anche essere rappresentato da una espressione), nel secondo l'indirizzo della variabile che vogliamo passare alla procedura. È questo il motivo per cui la variabile passata come parametro-valore non può essere modificata: la procedura ne riceve infatti solo il valore e lo può usare come meglio crede, ma non può modificare il parametro effettivo perché non ne conosce l'indirizzo.

Vediamo ora più da vicino come avviene il passaggio dei parametri. Ogni volta che viene chiamata una procedura vengono posti nello stack i valori dei parametri-valore, gli indirizzi dei parametri-variabile e l'indirizzo nel code segment dell'istruzione immediatamente successiva a quella di chiamata; in questo modo, quando la procedura termina, il programma può proseguire semplicemente ricavando dallo stack l'indirizzo dell'istruzione da eseguire.

Per accedere a quanto contenuto nello stack ci si può servire dai registri BP o SP; in quest'ultimo è sempre contenuto l'indirizzo dell'ultimo dato inserito nello stack. Lo stack cresce «verso il basso» e quindi, se vi metto nell'ordine i due integer «a» e «b», l'indirizzo di «b» sarà SP, quello di «a» sarà SP+2 (un integer occupa due byte).

Una procedura, come vedremo subito, può decrementare SP per creare nello stack lo spazio per le sue variabili locali; per prima cosa, quindi, si salva SP in BP. Più esattamente, dato che BP può già contenere questo tipo di informazione a beneficio di un'altra procedura che abbia chiamato quella che stiamo considerando, le prime istruzioni di qualsiasi funzione o procedura sono:

PUSH BP
MOV BP, SP

ovvero: salvo il BP di chi mi ha chiamato nello stack, quindi mi copio SP in BP. Risultato: il primo parametro nello stack sarà all'indirizzo BP+4, in quanto BP è uguale a SP, SP «punta» ora al BP appena salvato (due byte), sopra questo c'è l'indirizzo dell'istruzione cui la procedura deve tornare (altri due byte). È quindi sbagliato quanto si legge a pagina 222 del manuale, dove si dice che il primo parametro si trova in BP−1 (è questo in verità un errore un po' curioso, che ricorre solo in alcune edizioni del manuale).

Nel manuale c'è anche un'altra inesattezza, in quanto si dice che l'area per le variabili locali viene allocata subito sotto il BP appena salvato sullo stack; se ne potrebbe dedurre che in BP−1 si trova i primo byte delle variabili locali. In realtà in BP−1 (e BP−2) c'è... un altro BP! Dopo quelle due istruzioni, infatti, c'è un secondo PUSH BP (il Turbo Pascal usa BP non solo per accedere allo stack, ma anche per gestire gli indici degli array; in queste occasioni può modificarlo, e quindi se ne tiene una copia di riserva nello stack).

Un esempio

Consideriamo una funzione Pippo tanto semplice quanto inutile (figura 1). Il

Indirizzo assoluto	Indirizzo risp. a BP	Contenuto	
ssss:0FFE	[BP+0A]	????	spazio per il risultato della funzione
ssss:0FFC	[BP+8]	0009	valore per il parametro Vlr
ssss:0FFA	[BP+6]	dddd	indirizzo della variabile Glb12: segmento (DS)
ssss:0FF8	[BP+4]	0262	e offset
ssss:0FF6	[BP+2]	2DE3	indirizzo dell'istruzione da eseguire dopo il RET
ssss:0FF4	[BP+0]	1000	valore precedente di BP
ssss:0FF2	[BP-2]	0FF4	nuovo valore di BP (= SP dopo il primo PUSH BP)
ssss:0FF0	[BP-4]	????	spazio per la variabile Lcl

Figura 2 - Situazione dello stack durante l'esecuzione del programma della figura 1 subito dopo SUB SP,2 (dddd = data segment, ssss = stack segment, ???? = non definito). Per comodità il contenuto dello stack viene rappresentato in word invece che in singoli byte.

programma comincia con l'istruzione in 2DD5; per prima cosa viene decrementato SP per riservare nello stack lo spazio per il risultato della funzione, quindi vengono messi nello stack i parametri effettivi: «9» per il parametro Vlr, e l'indirizzo di Glbl2, segment e offset, per Vrbl.

Quando viene eseguita l'istruzione CALL 2D9F viene posto nello stack anche l'indirizzo dell'istruzione successiva, 2DE3.

Inizia quindi la funzione, con le tre istruzioni che abbiamo visto sopra. Segue un SUB SP,2 il cui scopo è quello di «far nascere» la variabile Lcl: la prossima cosa che verrà messa nello stack non andrà subito sotto il BP appena salvato, ma due byte più giù; rimane così libero lo spazio destinato ad ospitare la nostra variabile locale.

Nella figura 2 è rappresentata la situazione dello stack a questo punto. Per eseguire l'istruzione «Lcl:=Vlr» viene preso il valore che si trova nella locazione 0FFC dello stack e viene messo nell'indirizzo di Lcl, cioè nella locazione 0FF0 dello stack; il tutto avviene mediante il registro BP, che vale 0FF4. Lo stesso valore viene posto poi nell'indirizzo della variabile Glbl1, rappresentato dal suo offset nel data segment (0260).

Si tratta poi di modificare il valore della variabile Glbl2, passata alla funzione come parametro Vrbl, attraverso i quattro byte del suo indirizzo. L'istruzione LES DI,[BP+4] prepara il terreno, prendendo i quattro byte che partono da SS:BP+4 e convertendoli in un indirizzo ES:DI (ovvero: segmento in ES, offset in DI). Infine viene messo nello stack il risultato della funzione. Questo avviene sempre anche se, nel caso di funzioni integer, il risultato viene in realtà passato attraverso il registro AX; il risultato viene parcheggiato nello stack nel caso che l'esecuzione della funzione non termini subito dopo «Pippo:=1», ma quel che conta è che alla fine sia in AX.

Svolti i suoi compiti la funzione restituisce il controllo al programma principale. Viene ripristinato il valore originario di SP (0FF2, e così «scompare» la variabile Lcl), viene recuperato dallo stack quello di BP, si torna al punto in cui l'esecuzione era stata interrotta con un RET 8.

Quest'ultima istruzione fa sì che venga preso dallo stack l'indirizzo cui si deve tornare (2DE3) e venga subito dopo aggiunto un 8 a SP in modo da togliere dallo stack i parametri (e così «scompaiono» pure questi).

In realtà i parametri occupano solo 6 byte; il RET 8 toglie dallo stack anche il risultato che vi era stato «parcheggiato» e che ora non serve più. Possiamo infatti vedere che l'istruzione in 2DE3 mette nell'indirizzo di Glbl3 il risultato della funzione prendendolo dal registro AX.

In pratica

Vediamo come procedere per realizzare una procedura che usi gli interrupt 25H e 26H del DOS, quelli che consentono rispettivamente la lettura e la scrittura assoluta dei settori di un disco, ampiamente utilizzati dalle Norton Utilities, dai PC Tools, ecc. Si potrebbe pensare di fare più o meno così:

```
Reg.AL:=NumeroDrive; {A=0, ecc.}
Reg.CX:=NumeroSettoriDaLeggere;
Reg.DX:=PrimoSettoreDaLeggere;
Reg.DS:=Seg(Buffer);
Reg.BX:=Ofs(Buffer);
Intr($25,Reg);
```

Se tutto va bene... va tutto bene. Se però si verifica un errore viene settato il carry flag ma, per i motivi illustrati in apertura, non c'è modo di andare a vedere se questo è settato o no. Decidiamo quindi di scrivere in Assembler la nostra procedura, che chiameremo ABSDISK e alla quale passeremo quattro parametri-valore (Drive, NumSettori, PrimoSettore e Operazione; quest'ultimo sarà 0 per la lettura e 1 per la scrittura) e due parametri-variabile: Buffer e Esito.

Figura 3 - Il file ABSDISK.LST prodotto dall'Assembler sulla base del sorgente contenuto in ABSDISK.ASM. Poiché lo spazio è tiranno, e poiché un file LST riproduce il suo file ASM, non vengono presentati i due file. Si è invece aggiunto un bordo che delimita ABSDISK.ASM.

```
IBM Personal Computer MACRO Assembler   Version 2.00    Page    1-1
ABSDISK.ASM                                              01-07-88

                        TITLE   ABSDISK.ASM
                        PAGE    ,132
                ;-----------------------------------------------------------------
                ; TABELLA DEGLI OFFSET DEI PARAMETRI RISPETTO A BP.
                ; I parametri vengono posti nello stack cominciando dal primo a
                ; sinistra e quindi l'ultimo e' il piu' vicino, in BP+4
                ;-----------------------------------------------------------------
= 0004          Esito           EQU     4       ; parametro var, occupa 4 byte
= 0008          Buffer          EQU     8       ; parametro var, occupa 4 byte
= 000C          Operazione      EQU     12      ; integer, occupa 2 byte
= 000E          PrimoSettore    EQU     14      ; integer, occupa 2 byte
= 0010          NumSettori      EQU     16      ; integer, occupa 2 byte
= 0012          Drive           EQU     18
                ;-----------------------------------------------------------------
0000            CODE    SEGMENT
                        ASSUME  CS:CODE
0000            ABSDISK PROC    NEAR
0000  8A 46 12           MOV     AL,[BP+Drive]
0003  8B 4E 10           MOV     CX,[BP+NumSettori]
0006  8B 56 0E           MOV     DX,[BP+PrimoSettore]
0009  1E                 PUSH    DS                      ; Modificabile da LDS
000A  55                 PUSH    BP                      ; Modificato dall'INT
000B  C5 5E 08           LDS     BX,[BP+Buffer]
000E  80 7E 0C 01        CMP     BYTE PTR [BP+Operazione],1 ; Scrittura?
0012  74 04              JE      scrittura               ; Se si' salta
0014  CD 25              INT     25H                     ; Se no leggi
0016  EB 02              JMP     SHORT prosegui
0018                    scrittura:
0018  CD 26              INT     26H
001A                    prosegui:
001A  72 02              JC      errore                  ; E' settato il carry?
001C  2B C0              SUB     AX,AX                   ; Se tutto bene AX:=0
001E                    errore:
001E  9D                 POPF                            ; Via i flag dallo stack
001F  5D                 POP     BP
0020  C5 5E 04           LDS     BX,[BP+Esito]
0023  89 07              MOV     [BX],AX
0025  1F                 POP     DS
0026            ABSDISK ENDP
0026            CODE    ENDS
                        END
```

«Esito» servirà a trasmettere alla routine chiamante 0 se tutto è andato bene, o il codice d'errore del DOS in caso contraro (vi rimando ai vari «Technical Reference», Duncan o Norton per una esposizione dettagliata dei codici di errore). Buffer è invece l'indirizzo (in quanto parametro-variabile) dell'area di memoria in cui vogliamo immettere i dati letti da disco o da cui vogliamo trarre quello che vi vogliamo scrivere. Il Turbo Pascal consente di dichiararlo senza specificare il tipo: cosa molto comoda perché può servirci ora un «array[0..511] of byte» ora un «array[1..1024] of char»; non specificare vuol dire guadagnare in generalità e flessibilità.

Il riquadro interno alla figura n. 3 contiene il sorgente in Assembler per la nostra procedura. La cosa più delicata è rappresentata dall'esatto riferimento ai parametri nello stack; per questo conviene farsi prima un po' di conti e prepararsi una tabella di EQU. Per il resto, dopo tutto quello che abbiamo detto finora non dovreste avere difficoltà a comprendere i vari passaggi. Notate comunque che si esamina lo stato del carry flag dopo l'interrupt e poi si tolgono dallo stack i flag che questo vi aveva lasciato.

Preparato il file ABSDISK.ASM si assembla con:

MASM ABSDÍSK,,ABSDISK;

In questo modo si ottiene anche un file ABSDIKS.LST (figura 3) dal quale poi, con un po' di pazienza, si copiano i codici esadecimali nella procedura ABSDISK (figura 4). I più pigri possono trovare in MC-Link un programma IN_LINE.PAS che, preso un file ASM e il relativo file BIN prodotto da Assembler Linker e EXE2BIN, genera automaticamente l'inline statement.

Nel nostro esempio usiamo solo parametri. Se la procedura dovesse leggere o modificare variabili globali non passate come parametri dovremmo aggiungere un passaggio; si tratta di mettere tra gli EQU qualcosa come

NomeVarGlob EQU 1111H

o, al posto di 1111H, un qualsiasi numero maggiore di 0FFH (255) e minore di 0FFFFH (65535), per essere sicuri che l'Assembler lo intenda come numero di due byte. Quando poi ci scriveremo il nostro inline statement sostituiremo ogni «1111» con «NomeVarGlob»: ci penserà il Turbo Pascal a sostituire «NomeVarGlob» con il numero di due byte corrispondente al suo offset nel data segment.

Se invece vogliamo fare di ABSDISK.ASM il sorgente di una procedura «external», dobbiamo aggiungere all'inizio le istruzioni PUSH BP e MOV BP,SP e alla fine MOV SP,BP e POP BP seguite da RET n. Quest'ultimo n deve essere il numero di byte occupati dai parametri della procedura (16 nel nostro caso; se ABSDISK fosse una funzione intera dovremmo aggiungere i byte occupati dal risultato). Si procede poi con:

MASM ABSDISK;
LINK ABSDISK;
EXE2BIN ABSDISK.EXE ABSDISK.BIN

ignorando messaggi d'errore come «No Stack Segment». La procedura ABSDISK si riduce così alla sola intestazione (nome e elenco parametri) seguita da «external 'ABSDISK.BIN». Non vi sono codici esadecimali da copiare a mano, in quanto le istruzioni contenute in ABSDISK.BIN vengono inserite nel programma automaticamente durante la compilazione. La conseguenza è però che non è possibile intervenire su questo automatismo per aggiungere il passaggio che abbiamo visto necessario per le variabili globali, e quindi si può accedere a queste solo se passate come parametri. È questo il significato di quella frase del manuale che dice che in una procedura external «non vi devono essere riferimenti al segmento dati» (p. 210). MC

```
program AbsDisk;
var
  i, Errore: integer;
  Buf: array[1..512] of char;
const
  LETTURA   = 0;
  SCRITTURA = 1;
procedure AbsDisk(Drive, NumSettori, PrimoSettore, Operazione: integer;
                  var Buffer; var Esito: integer);
begin
  inline(
    $8A/$46/$12/       { MOV  AL,[BP+Drive]                }
    $8B/$4E/$10/       { MOV  CX,[BP+NumSettori]           }
    $8B/$56/$0E/       { MOV  DX,[BP+PrimoSettore]         }
    $1E/               { PUSH DS                           }
    $55/               { PUSH BP                           }
    $C5/$5E/$08/       { LDS  BX,[BP+Buffer]               }
    $80/$7E/$0C/$01/   { CMP  BYTE PTR [BP+Operazione],1   }
    $74/$04/           { JE   scrittura                    }
    $CD/$25/           { INT  25H; lettura                 }
    $EB/$02/           { JMP  SHORT prosegui               }
    { scrittura: }
    $CD/$26/           { INT  26H                          }
    { prosegui: }
    $72/$02/           { JC   errore                       }
    $2B/$C0/           { SUB  AX,AX                        }
    {errore:}
    $9D/               { POPF                              }
    $5D/               { POP  BP                           }
    $C5/$5E/$04/       { LDS  BX,[BP+Esito]                }
    $89/$07/           { MOV  [BX],AX                      }
    $1F)               { POP  DS                           }
end;
begin { legge il boot record del disco in A: }
  AbsDisk(0,1,0,LETTURA,Buf,Errore);
  if Errore = 0 then
    for i := 1 to 512 do
      if Buf[i] in [' '..'~'] then write(Buf[i])
end.
```

Figura 4 - La procedura ABSDISK come prodotta copiando i codici esadecimali di ABSDISK.LST in un «inline statement».

Scrive 400 Mb, si rimuove come un floppy si usa come un Winchester è un disco ottico

Optotech

I Drive Ottici Optotech scrivono i dati su una cartuccia removibile da 5,25". Pratica e facile da usare quanto un Floppy, ogni cartuccia ha una capacità di più di 400 Mbytes (200 per facciata). Grazie al software di corredo i Drive Optotech si usano come un qualsiasi Winchester e permettono di archiviare economicamente una massa di informazioni illimitata. Si installano in pochi minuti con estrema facilità.

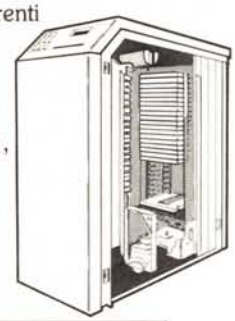
Il controller, disponibile per tutti i principali sistemi, può guidare fino a 4 Drive.
I dati immessi sono leggibili in qualsiasi momento ma non più cancellabili, garantendo un'assoluta sicurezza di archiviazione.
Le unità sono disponibili in versioni pronte all'uso per IBM XT/AT, Olivetti, Microvax, Macintosh. I dati possono essere interscambiati tra differenti sistemi.
Optotech è disponibile anche in versione OEM.

Caratteristiche tecniche

Optical Disk Drive
- Capacità formattata 202,4 Mbytes per facciata.
- Tecnologia di lettura durante la scrittura, con controllo in tempo reale della corretta registrazione.

Optofile: una workstation da 26,4 Gigabytes!

Optofile è il primo auto-caricatore di dischi ottici da 5,25": può contenere fino a 66 dischi e consente l'accesso automatico a tutti i file in essi registrati. È compatibile con tutti i principali sistemi attraverso l'interfaccia SCSI; include da uno a quattro Drive, adattandosi così a differenti esigenze di velocità di accesso.



Versioni:
- rack (fino a 66 dischi), capacità 26,4 Gigabytes;
- tower e tavolo (fino a 32 dischi) capacità 12,8 Gigabytes;
- box speciale Apple.

Per maggiori informazioni sui prodotti distribuiti dalla Contradata, telefonate allo 039/737015 o scrivere a Contradata s.r.l., via Monte Bianco 4, 20052 Monza (MI), telex 352830 CONTRA I - fax 039-735276 G3.

contradata

PER COMPUTER CHE NON HANNO TEMPO DA PERDERE