di Sergio Polini

Le strutture di dati della OOP

Nelle prime due puntate di quest'anno abbiamo cominciato a vedere la «sintassi» della OOP; la semplicità del Turbo Pascal ha consentito di ridurre il tutto all'esame di poche nuove parole riservate (object, constructor, destructor, virtual e private), che abbiamo esaminato quasi tutte mediante semplici escursioni ... nel regno animale. Ci rimane ancora da vedere cosa sono e a cosa servono i destructor, nonché altri aspetti della sintassi estesa delle procedure New e Dispose. Anche per questo chiederemo aiuto a qualche quadrupede; subito dopo, però, inizieremo un più serio discorso sulla programmazione orientata all'oggetto come «programmazione per classi», e quindi sulle gerarchie di classi tipiche del nuovo stile di programmazione

```
unit Animali;

interface

type
  PAnimale = ^TAnimale;
  TAnimale = object
    constructor Init(LaSpecie, IlVerso: string);
(*$IFDEF STATIC*)
    destructor Done;
(*$ELSE*)
    destructor Done; virtual;
(*$ENDIF*)
    procedure ChiSei; virtual;
  private
    Specie: ^string;
    Verso: ^string;
  end;

implementation

constructor TAnimale.Init(LaSpecie, IlVerso: string);
begin
  GetMem(Specie, Length(LaSpecie)+1);
  Specie^ := LaSpecie;
  GetMem(Verso, Length(IlVerso)+1);
  Verso^ := IlVerso;
end;

destructor TAnimale.Done;
begin
  FreeMem(Specie, Length(Specie^)+1);
  FreeMem(Verso, Length(Verso^)+1);
end;

procedure TAnimale.ChiSei;
begin
  Writeln('Sono un ', Specie^, ' e il mio verso e'': ', Verso^);
end;

end.
```

Figura 1 - La unit ANIMALI, in cui viene dichiarata la classe TAnimale. *Se compilata normalmente, il* ***destructor*** *sarà virtuale; sarà invece statico se si compila dopo aver definito «STATIC» (con «Options/Compiler/Conditional defines» nell'ambiente integrato, o con l'opzione «/dSTATIC» se si usa il compilatore separato, TPC.EXE).*

La volta scorsa abbiamo visto che, a partire dal Turbo Pascal 5.5, è possibile chiamare la procedura *New* con un secondo parametro opzionale, mediante il quale «passare» il **constructor** della classe cui appartiene l'oggetto il cui indirizzo vogliamo ci venga ritornato nel primo parametro. Si dispone di analoghe estensioni per la procedura *Dispose* (alla quale va ovviamente passato il **destructor**). *New* può poi essere usata come funzione oltre che come procedura; in altri termini, il puntatore in cui vogliamo l'indirizzo dell'oggetto allocato può essere sia il primo parametro della procedura *New*, o può essere ciò cui si assegna il valore ritornato da una funzione *New* il cui primo parametro sia il tipo del puntatore (ne trovate esempi nel programma in figura 4).

Si tratta di accorgimenti che hanno lo scopo di rendere più semplice la scrittura di codice relativo alla «costruzione» e poi alla «distruzione» di oggetti allocati dinamicamente, in considerazione della frequenza con cui può capitare di avere a che fare con tali oggetti e con i relativi puntatori. A ben vedere, tuttavia, le estensioni di *New* possono essere attribuite soprattutto ad una esigenza di «simmetria» con la nuova *Dispose;* a differenza di quanto accade con *New*, infatti, la sintassi estesa di *Dispose* non è opzionale, ma spesso obbligatoria.

Le tabelle dei metodi virtuali

Ripetiamo che in una classe in cui siano dichiarati metodi virtuali deve essere previsto anche almeno un **constructor**, e che questo va chiamato prima di qualsiasi altro accesso alle istanze di quella classe. Può dirsi che si tratta di una regola analoga a quella che vieta l'accesso a variabili non inizializzate, in quanto un **constructor** è la sede più naturale per inizializzare le istanze di una classe, ma dimenticare di chiamare un **constructor** avrebbe effetti ancora più gravi, in quanto in esso sono sempre presenti, accanto al codice scritto dal programmatore, anche istruzioni «invisibili» (generate cioè automaticamente dal compilatore) che provvedono ad una particolare e importantissima inizializza-

```
unit Domestic;

interface

uses Animali;

type
  PDomestico = ^TDomestico;
  TDomestico = object(TAnimale)
    constructor Init(LaSpecie, IlVerso, IlNome, IlPadrone: string);
(*$IFDEF STATIC*)
    destructor Done;
(*$ELSE*)
    destructor Done; virtual;
(*$ENDIF*)
    procedure ChiSei; virtual;
  private
    Nome: ^string;
    Padrone: ^string;
  end;

implementation

constructor TDomestico.Init(LaSpecie, IlVerso, IlNome, IlPadrone: string);
begin
  TAnimale.Init(LaSpecie, IlVerso);
  GetMem(Nome, Length(IlNome)+1);
  Nome^ := IlNome;
  GetMem(Padrone, Length(IlPadrone)+1);
  Padrone^ := IlPadrone;
end;

destructor TDomestico.Done;
begin
  FreeMem(Nome, Length(Nome^)+1);
  FreeMem(Padrone, Length(Padrone^)+1);
  TAnimale.Done;
end;

procedure TDomestico.ChiSei;
begin
  TAnimale.ChiSei;
  Writeln('  il mio nome e'' ', Nome^,' e il mio padrone e'' ', Padrone^);
end;

end.
```

Figura 2 - La unit DOMESTIC, nella quale si definisce TDomestico, *classe di tutti gli animali domestici derivata da* TAnimale *(figura 1). Anche qui il* ***destructor*** *sarà statico o virtuale secondo le opzioni di compilazione.*

zione. Nella figura 3 potete trovare una descrizione della rappresentazione interna di variabili che siano istanza delle classi *TAnimale* e *TDomestico,* definite nelle unit riportate nelle figure 1 e 2. Notate che, subito dopo i puntatori a stringa *LaSpecie e IlVerso,* le istanze di *TAnimale* hanno ulteriori due byte che contengono l'offset nel segmento dati della *Virtual Method Table* (VMT), cioè della «tabella dei metodi virtuali», e quindi di quella struttura di dati interna con la quale viene implementato il polimorfismo. Il compilatore genera una VMT per ogni classe, e ogni volta che viene chiamato un metodo virtuale, con una sintassi del tipo "<istanza>.<metodo>" (oppure: "<puntatore a istanza>^< metodo>"), viene chiamata la procedura o funzione il cui indirizzo è contenuto nella VMT della classe a cui appartiene l'istanza, cui si accede proprio mediante quei due byte.

Ciò consente appunto di chiamare ogni volta il metodo giusto, non solo attraverso le istanze (e qui non ci sono problemi, perché il compilatore conosce bene la classe cui appartiene ogni istanza), ma anche attraverso puntatori ad una classe «base» cui sia stato assegnato l'indirizzo dell'istanza di una classe «derivata». Si tratta di un «aggiramento di tipo» reso possibile dal fatto che, nelle istanze di classi derivate, l'offset della VMT si trova nella stessa posizione che nelle istanze della classe base; in altri termini, chiamando il metodo *ChiSei* attraverso un puntatore del tipo «puntatore alla classe *TAnimale*» (come nel programma nella figura 4), per prima cosa si cerca l'offset della VMT nel nono e decimo byte della rappresentazione in memoria di qualsiasi istanza della classe *TAnimale* o di tutte le classi da questa derivate; nella tabella è contenuto l'indirizzo della versione specifica di ogni classe di ciascun metodo virtuale.

Il ruolo dei **constructor** è fondamentale proprio perché quelle istruzioni invisibili cui accennavo prima provvedono a mettere l'offset della VMT in quei due byte.

I destructor: meglio virtuali

I **destructor** servono ad annullare le inizializzazioni portate a termine dai **constructor,** ma non nel senso che cancellano il contenuto di quei due byte. Le unit delle figure 1 e 2 propongono classi analoghe a quelle già viste il mese scorso, con qualche differenza. Si nota subito che, invece di prevedere classi distinte per gatti, cani e lupi, questa volta vengono previste solo la classe di tutti gli animali e la classe, da questa derivata, di tutti gli animali domestici. Soprattutto per semplificare, in quanto la principale differenza risiede nel tipo dei campi-dati: stringhe il mese scorso, puntatori a stringa questa volta. Si tratta di una scelta che è ben probabile in un programma reale: invece che riservare ogni volta 256 byte per una stringa che potrebbe essere molto più corta, si preferisce usare puntatori che, grazie alla procedura *GetMem,* puntino ad un blocco di memoria allocata dinamicamente e della dimensione effettivamente necessaria. In questi casi sarà compito del **constructor** provvedere alla inizializzazione, cioè all'allocazione di memoria mediante *GetMem,* come si fa appunto nelle unit ANIMALI e DOMESTIC.

La memoria così allocata viene automaticamente rilasciata quando il programma termina; va però considerato il caso di variabili locali ad una procedura: si avrebbe un'allocazione di memoria che verrebbe rilasciata solo alla fine del programma, non all'uscita dalla procedura; avremmo variabili che continuano ad occupare memoria pure quando, sotto ogni altro aspetto, non esistono più. Un vero e proprio spreco. Qui intervengono i **destructor** che, come può vedersi dalle figure 1 e 2, si fanno carico di rilasciare la memoria allocata dai **constructor.**

È molto importante ricordare di rendere virtuali i **destructor.** Ciò non è sempre necessario, in quanto se ne può fare a meno in classi «isolate», che cioè non derivino da altre e da cui non si prevede che vengano derivate altre classi; ma va anche considerato che, quando definiamo una classe, potremmo non essere in grado di escludere la possibilità di derivarne altre in futuro, o che lo facciano altri (magari gli utenti della nostra unit, della quale non possono modificare i sorgenti perché... non li hanno).

Nel dubbio, meglio virtuali. Le unit ANIMALI e DOMESTIC possono essere compilate definendo o no «STATIC» (con «Options/Compiler/Conditional defines» nell'ambiente integrato, o con l'opzione «/dSTATIC» se si usa il compilatore separato). Se si definisce «STATIC» i **destructor** saranno statici; la conseguenza ci viene illustrata dal programma della figura 4, il cui output è riportato nella figura 5: chiamando la funzione *MemAvail* prima e dopo di ogni altra cosa, si può rilevare che mancano all'appello 32 byte (chi avesse ancora il

```
Classe TAnimale
---------------
Struttura di ogni istanza:
- Specie (puntatore a stringa):            4 byte
- Verso (puntatore a stringa):             4 byte
- Offset VMT nel data segment :            2 byte
SizeOf:                                   10 byte
Dimensione di un blocco allocato con New: 16 byte

Classe TDomestico
-----------------
Struttura di ogni istanza:
- Specie (puntatore a stringa):            4 byte
- Verso (puntatore a stringa):             4 byte
- Offset VMT nel data segment :            2 byte
- Nome (puntatore a stringa):              4 byte
- Padrone (puntatore a stringa):           4 byte
SizeOf:                                   18 byte
Dimensione di un blocco allocato con New: 24 byte
```

Figura 3 - Occupazione di memoria di istanze delle classi TAnimale *e* TDomestico. *Viene indicata anche la dimensione dei blocchi allocati con* New *in quanto questa, nel Turbo Pascal 6.0, è diversa da quella riportata dalla funzione* SizeOf; *è una conseguenza della nuova gestione dell'heap, descritta nella prova apparsa sul numero di febbraio.*

Turbo Pascal 5.5 dovrebbe eliminare **private** dalle unit e spostare la dichiarazione dei campi-dati privati prima di quella dei metodi; il risultato numerico sarebbe diverso a causa della mutata gestione dell'heap, che ho illustrato il mese scorso in occasione della prova del nuovo compilatore; mancherebbe comunque ugualmente una manciata di byte).

Facendo un po' di conti, possiamo verificare che i 32 byte sono proprio quelli allocati per le stringhe puntate da *Nome* e *Padrone* nelle due istanze della classe *TDomestico:* «Micio» occupa sei byte (compreso il byte di lunghezza), «Gustavo» otto, «Fido» cinque e «Antonio» otto; poiché la memoria viene allocata per multipli di otto byte, il totale fa proprio 32. Cosa è successo? Semplice: poiché i **destructor** non erano virtuali, la loro chiamata attraverso puntatori alla classe *TAnimale* si è tradotta nella chiamata dei **destructor** di questa, *TAnimale.Done;* è stata quindi rilasciata solo la memoria che era stata allocata per le stringhe puntate da *Specie* e *Verso*. Basta omettere la definizione di «STATIC» per ottenere che i **destructor** siano virtuali e i conti tornino.

La nuova Dispose

Non basta. Poiché il polimorfismo si manifesta quando operiamo sugli indirizzi di oggetti (attraverso puntatori o parametri variabile) invece che sugli oggetti direttamente, il programma della figura 4 definisce «A» come array di puntatori a *TAnimale*. L'array può essere inizializzato sia con gli indirizzi di variabili globali o locali, come nel caso di *Leone*, sia con quelli di variabili allocate dinamicamente con *New*. La creazione di variabili dinamiche comporta problemi analoghi a quelli già visti per i campi puntatore: se si vogliono evitare sprechi, ci si deve preoccupare della liberazione della memoria non più necessaria.

In un programma tradizionale possiamo usare le coppie *New/Dispose* o *GetMem/FreeMem*. Con *GetMem* si può allocare tanta memoria quanta ne viene indicata nel secondo parametro, memoria che può poi essere rilasciata con *FreeMem;* la corrispondenza tra la dimensione del blocco allocato e quella del blocco rilasciato è a carico del programmatore, cui compete assicurare che il numero di byte da rilasciare — specificato nel secondo parametro di *FreeMem* — coincida con quello dei byte allocati con *GetMem*. La mancanza di automatismi è il prezzo che si deve pagare per godere di un prezioso vantaggio: si può allocare solo la memoria che serve, proprio come nel caso dei puntatori a stringa usati nei campi-dati delle classi *TAnimale e TDomestico*. Ad esempio, solo 6 byte per una stringa invece di 256. Quando non occorrano tali ottimizzazioni «manuali», *New* e *Dispose* sono molto più comode e sicure: vengono allocati e rilasciati esattamente tanti byte quanti ne richiede il tipo del dato, e a questo provvede automaticamente il compilatore.

Ecco il nostro problema: abbiamo visto che possiamo assegnare a puntatori a *TAnimale* anche gli indirizzi di oggetti che siano istanze di *TDomestico;* è proprio in questo modo, anzi, che possiamo ottenere che diversi oggetti «rispondano» in modo diverso ad uno stesso «messaggio». Quei puntatori, però, rimangono pur sempre puntatori a *TAnimale*. Vediamo nella figura 6 cosa questo comporti: se si definisce «OLDSYNT», viene usata la *Dispose* tradizionale e mancano 16 byte all'appello. Anche qui è facile fare i conti: si tenta

```
program AnimaliTest;
uses
  Animali, Domestic;
var
  A: array[1..4] of PAnimale;
  Leone: TAnimale;
  Gatto: PDomestico;
  Cane: PDomestico;
  Lupo: PAnimale;
  i: integer;
begin
  Writeln('Memoria dinamica prima della inizializzazione: ', MemAvail);
  Writeln;

  (* Inizializzazione *)
  Leone.Init('leone', 'Roarrr!');
  A[1] := @Leone;
  New(Gatto);
  Gatto^.Init('gatto', 'Miaooo!', 'Micio', 'Gustavo');
  A[2] := Gatto;
  New(Cane, Init('cane', 'BauBau!', 'Fido', 'Antonio'));
  A[3] := Cane;
  A[4] := New(PAnimale, Init('lupo', 'Uhuhuhu!'));

  (* Risposte a messaggi *)
  for i := 1 to 4 do
    A[i]^.ChiSei;

  (* Deallocazione. A[1] viene trattato a parte perche' *)
  (* contiene l'indirizzo di una variabile statica      *)

  A[1]^.Done;
  for i := 2 to 4 do begin
(*$IFDEF OLDSYNT*)
    A[i]^.Done;
    Dispose(A[i]);
(*$ELSE*)
    Dispose(A[i], Done);
(*$ENDIF*)
  end;

  Writeln;
  Writeln('Memoria dinamica dopo la deallocazione: ', MemAvail);
end.
```

Figura 4 - Un breve programma che mostra l'elasticità della sintassi estesa della procedura/funzione New, *nonché gli effetti dell'uso di* ***destructor*** *statici e virtuali (secondo le opzioni usate nella compilazione delle unit ANIMALI e DOMESTIC) e delle sintassi tradizionale ed estesa della procedura* Dispose *(secondo che si definisca o meno «OLDSYNT» nella compilazione).*

```
Memoria dinamica prima della inizializzazione: 490752

Sono un leone e il mio verso e': Roarrr!
Sono un gatto e il mio verso e': Miaooo!
  il mio nome e' Micio e il mio padrone e' Gustavo
Sono un cane e il mio verso e': BauBau!
  il mio nome e' Fido e il mio padrone e' Antonio
Sono un Lupo e il mio verso e': Uhuhuhu!

Memoria dinamica dopo la deallocazione: 490720
```

Figura 5 - Output del programma della figura 4, se le unit ANIMALI e DOMESTIC vengono compilate con «STATIC» definito. Essendo così statici i ***destructor****, si nota che mancano all'appello 32 byte: tanti quanti erano stati allocati per le stringhe «Micio», «Gustavo», «Fido» e «Antonio» (otto byte per ognuna, per via della nuova gestione dell'heap).*

```
Memoria dinamica prima della inizializzazione: 490608

Sono un leone e il mio verso e': Roarrr!
Sono un gatto e il mio verso e': Miaooo!
  il mio nome e' Micio e il mio padrone e' Gustavo
Sono un cane e il mio verso e': BauBau!
  il mio nome e' Fido e il mio padrone e' Antonio
Sono un Lupo e il mio verso e': Uhuhuhu!

Memoria dinamica dopo la deallocazione: 490592
```

Figura 6 - Se le unit vengono compilate senza definire «STATIC», ma il programma con «OLDSYNT» definito, mancano all'appello 16 byte: otto byte per ognuna delle due istanze della classe TDomestico*, corrispondenti ai due puntatori a stringa che in essa si aggiungono ai due ereditati da* TAnimale.

di rilasciare con *Dispose* due istanze di *TDomestico* e una di *TAnimale,* ma in tutti e tre i casi vengono rilasciati tanti byte quanti ne occupa una istanza di *TAnimale;* rimangono otto byte per ognuna delle due istanze di *TDomestico,* quelli occupati dai puntatori *Nome* e *Padrone.*

Per ottenere un bilancio in pareggio basta anche qui omettere la definizione di «OLDSYNT». In questo modo viene usata la nuova *Dispose,* che ammette come secondo parametro il **destructor** dell'oggetto. Non è necessario che questo sia virtuale, ma basta che esista la VMT per la classe; a questo provvede comunque il compilatore, che genera una VMT per ogni classe in cui sia definito un **constructor,** o almeno un metodo virtuale, o anche solo un **destructor.** Con la nuova sintassi di *Dispose,* infatti, viene prima eseguito il **destructor**, quindi si passa alla routine di deallocazione della memoria l'informazione circa l'effettiva dimensione dell'oggetto da rilasciare, contenuta nei primi due byte della VMT della sua classe.

È per questo che la nuova sintassi di *Dispose* non può essere considerata opzionale.

Strutture di dati

Non pretendo che vi ci siate già abituati, ma «è un» sono due paroline magiche nella OOP. Nell'esempio di questo mese abbiamo chiesto «Chi sei?» ad un leone, a un gatto, a un cane ed a un lupo. Tutti e quattro sono animali, ma solo due sono animali domestici. La relazione «è un» è quella che ci aiuta ad individuare la classe di alcuni oggetti (un leone «è un» animale), come anche la derivazione di una classe da un'altra (un gatto «è un» animale domestico, ma un animale domestico «è un» animale; nel nostro caso, «è un» animale con un nome e un padrone).

Per quanto non sia un processo né breve né facile, prima o poi ci si accorge che ragionare in termini di «è un» è utile non solo quando si mette a punto la classificazione degli oggetti su cui si deve lavorare, ma anche quando si devono individuare le strutture di dati più adatte. Nel programma della figura 4 abbiamo usato un array, ma cosa è un array? Potremmo ben dire che un **array** «è un» gruppo di oggetti accessibili mediante indici, ma a questo punto potremmo anche accorgerci che esistono altri possibili gruppi di oggetti. Un **set**, ad esempio, «è un» gruppo di oggetti senza duplicazioni (non vi possono essere due oggetti uguali) e non accessibili mediante indici; si può solo sapere se un dato oggetto appartiene o no al gruppo. Una **lista** «è un» gruppo di oggetti tra loro concatenati accessibili sequenzialmente, dall'uno all'altro. Uno **stack** «è una» lista in cui si può accedere solo all'ultimo oggetto che vi è stato aggiunto. E così via. Perché non pensare ad una classe «gruppo di oggetti» da cui derivare le altre?

Abbiamo fin qui visto alcuni dei vantaggi che si ricavano dalla definizione di classi «base» e «derivate» e dai metodi virtuali; ci siamo serviti a questo scopo di array di puntatori ad oggetti. Un vantaggio è rappresentato dalla possibilità di creare una classe derivata (come *TDomestico*) disponendo anche solo della **interface** e del codice compilato di una classe base (*TAnimale*). Un gatto «è un» animale, ma si comporta in modo diverso da un leone; ciò nonostante, possiamo estendere il comportamento degli animali senza bisogno di modificare i sorgenti, ma solo derivando una classe da un'altra già compilata. Si può fare lo stesso con le strutture di dati. Avremo praticamente sempre bisogno di «gruppi» di oggetti, di array, liste, insiemi, ecc.

Molte volte buona parte del tempo di programmazione se ne va nella preparazione delle strutture di dati (pensate alle liste), anche quando quelle che ci servono non sono poi così diverse da quelle che altre volte abbiamo usato. Una gerarchia di classi per le strutture di dati sarebbe quindi molto utile.

Programmare per oggetti, infatti, vuol dire in realtà programmare per classi. Non si gode appieno dei vantaggi della OOP se non quando si è in grado di lavorare sulla base di una gerarchia di classi. Di ciò è ben consapevole chi abbia dato un'occhiata non superficiale al linguaggio principe della OOP, a quello Smalltalk in cui tutto è oggetto, anche perché sarà poi facile ritrovare tracce (anche vistose) della sua gerarchia di classi in altri ambienti: nella libreria di Keith Gorlen per il C++, nell'Objective-C di Brad Cox, nel Turbo Vision. Occorre un po' di tempo per convincersi che un approccio di questo tipo può avere effetti notevoli sulla produttività dell'attività di programmazione, e non pretendo certo di avervi già convinto con un paio di favolette su cani e gatti. Posso magari ricordarvi fatti molto significativi che avevo riportato in occasione di alcune prove apparse sulla rivista: la Microsoft ha da tempo scelto la OOP per risolvere i problemi di gestione dei suoi sorgenti, che stavano rischiando di compromettere i suoi utili (prova del Turbo C++ 1.0, MC luglio/agosto 1990); la IBM raccomanda la OOP, e in particolare lo Smalltalk V/PM, per rendere effettivamente praticabile quella programmazione sotto OS/2 Presentation Manager che altrimenti si presenta estremamente ardua (prova Turbo Pascal 6.0, MC febbraio 1991).

Non è per caso, quindi, che MC vi propone a partire da questo numero una serie di articoli dedicata allo Smalltalk. Quanto a noi, nei prossimi appuntamenti esamineremo una piccola gerarchia di classi ispirata a quelle già menzionate ed anche a quella proposta dalla Borland nella «class library» del Turbo C++, in modo da poter toccare con mano come può essere implementata e poi usata una gerarchia di strutture di dati (non è escluso che, quando leggerete queste righe, potrete già trovarla su MC-Link). Ci sarà utile per poi poterci muovere con disinvoltura nella più complessa gerarchia del Turbo Vision.

MC

SOFTWARE TECNICO :

- Contabilità Imprese Edili e Studi Tecnici
- Gestione Gare d'Appalto e Lavori Pubblici
- Gestione Albi professionali
- Topografia in 2D e 3D
- Progettazione stradale, cartografia
- Software per Gestione dBase dal CAD e scannerizzazione immagini
- Software Tecnico manutenzione Ascensori
- Manutenzione ed amm.ne Immobili

CAD:

- Architettonici e applicativi AUTOCAD* (alcuni esempi):
- Termotecnica
- Gestione topografica di cave di marmo, cave di inerti, discariche, bacini.
- Terreni agricoli
- Arredamento interni - Cucine - Bagni

AUTOCAD E' UN MARCHIO AUTODESK

SOFTWARE MEDICO:

- Gestione Medici di base
- Ostetricia - Ginecologia
- Medicina generale (Realizzazione di procedure per altre specializzazioni) - Oculistica

SOFTWARE GESTIONALE APPLICATIVO (alcuni esempi)

- Contabilità - Abbigliamento - Ottica

RICERCA OPERATIVA E MODELLI DI

OTTIMIZZAZIONE CON APPLICAZIONI

SPECIFICHE GIA' SVILUPPATE.

SOFTWARE DISPONIBILE PER AMBIENTI MS-DOS, WINDOWS, UNIX.

I PREZZI SONO IVA ESCLUSA

SOFTWARE ORIZZONTALE (esempi):

TURBO C ++ IT	Lit. 300.000
TURBO C ++ P. IT	Lit. 450.000
WINDOWS 3.0 IT	Lit. 260.000
INFORMIX WINGZ	Lit. 620.000
MS-OS 2 1.1	Lit. 580.000
ALDUS PAGE MAKER	Lit. 1.290.000
EXCEL IT	Lit. 670.000
AUTOCAD 10 386	Lit. 6.700.000
CLIPPER 5.0	Lit. 940.000

HARDWARE (esempi):

PC AT 16 Mhz. - 1 MB RAM - 1 FDD 1.2 MB - 1 HD 40 MB - SK VIDEO VGA - MONITOR VGA MONOCROMATICO -

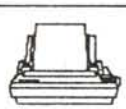
Configurazione completa Lit. 1.490.000

PC 386 33 Mhz. - 2 MB RAM - 1 FDD 1.2 MB - 1 HD 40 MB - SK VIDEO VGA - MONITOR VGA MONOCROMATICO -

Configurazione completa Lit. 3.890.000

Questi sono alcuni esempi delle nostre offerte software-hardware. Per l'invio del catalogo SOFTWARE-HARDWARE telefonare ai numeri sottoindicati.

STAMPANTI (esempi):



PANASONIC LASER	Lit. 2.800.000
OKI 380 24 AGHI	Lit. 810.000
CITIZEN SWIFT 9	Lit. 445.000

SIAMO PRESENTI
A ROMAUFFICIO '91
PAD. 10 STAND 54/55

MICROSYS
Soluzioni software ed hardware

Via Germanico,24
00192 - ROMA
Tel. 06/3251763-4-5

SI RICERCANO RIVENDITORI ED AGENTI