di Sergio Polini

Smalltalk/V
L'ambiente di sviluppo

La volta scorsa abbiamo visto le classi astratte della nostra piccola gerarchia. Ora cominceremo ad esaminare le classi «concrete», quelle cioè di cui è possibile creare istanze. Vedremo anche un primo esempio di uso delle classi in un breve demo, ma avremo soprattutto occasione di discutere di alcuni importanti aspetti della implementazione delle strutture di dati della OOP

Nel Pascal ogni array deve avere una dimensione prefissata; altri linguaggi, come il Modula-2, consentono anche array «dinamici». Già nella prova del Turbo Pascal 3.0 (MC di aprile 1987) avevo mostrato come il compilatore della Borland, grazie alla possibilità di omettere la specificazione del tipo nei parametri-variabile, permette di scrivere funzioni e procedure in grado di operare su array di dimensione qualsiasi, più o meno come si può fare con gli *open arrays* del Modula-2; ma rimane comunque un limite: gli elementi di un array devono tutti appartenere allo stesso tipo, che deve essere ovviamente noto e determinato già al momento della compilazione.

Un array «è un» gruppo di oggetti, e come tale «è una» collezione. Ciò che lo distingue da altre collezioni è la possibilità di accesso in lettura/scrittura ai suoi elementi mediante indici. Le collezioni indicizzate della OOP sono quindi una sorta di array, con due importanti differenze: non solo la classe *TIndexedCollection* — come la classe analoga che si trova in Smalltalk — può avere un qualsiasi numero di elementi, ma questi possono essere di qualsiasi tipo, purché derivato dal tipo base della gerarchia. Questo è almeno l'approccio più rigoroso, tipico appunto di Smalltalk e di altre implementazioni che ne rispettano la filosofia, dall'Objective-C di Cox alla «class library» del Borland C++; sono possibili approcci diversi, come quello del Turbo Vision, in cui alle collezioni «polimorfiche» vengono preferite collezioni «specializzate» (*TStringCollection*, *TResourceCollection*), allo scopo di usare tipi tradizionali come *string*; anche nel Turbo Vision, tuttavia, troviamo una vera e propria collezione polimorfica: si tratta di *TGroup*, che può contenere oggetti di qualsiasi classe purché derivata da *TView*. È questo uno dei motivi per cui mi sembra utile sottolineare che nella *TCollection* del Turbo Vision trovate solo una delle possibili implementazioni delle collezioni, ma altre sono praticabili; e il modo migliore ritengo sia quello di toccare con mano una di queste alternative.

La unit INDXCOLL

Trovate nella figura 1 il sorgente della unit INDXCOLL; in essa vengono precisate interfaccia e implementazione della classe *TIndexedCollection* e del suo iteratore, *TIndexedCollectionIterator*. Le collezioni indicizzate vengono create mediante la chiamata del constructor *Init*, che ha due argomenti: *Size* per il numero iniziale di elementi della collezione, *Delta* per ogni accrescimento. Come vedremo, quando occorre aggiungere un elemento in più rispetto a quelli che la collezione può contenere, la collezione viene «espansa» in modo da far posto ai nuovi arrivi; poiché si tratta di un'operazione relativamente onerosa, è possibile specificare in *Delta* un numero di spazi da aggiungere ogni volta, in modo da non dover ripetere troppo spesso l'espansione (identico è il significato degli argomenti *ALimit* e *ADelta* del constructor delle collezioni del Turbo Vision). L'implementazione del constructor è semplice: il valore dei due argomenti viene salvato nelle variabili d'istanza private *Sz* e *Dlt*, mentre alla variabile *A*, di tipo *PObjPArray* (dichiarato nella unit COLLECT del mese scorso), viene assegnato l'indirizzo di un'area di memoria allocata con *GetMem* e tale da contenere *Size* puntatori ad oggetti. L'array di puntatori ad oggetti così creato viene inizializzato con tutti *Null* (i puntatori all'oggetto indefinito creato dal codice di inizializzazione della unit BASE).

Prima di passare al destructor, apparentemente semplice immagine speculare del constructor, sarà bene vedere come vengono aggiunti nuovi elementi alla collezione.

Il metodo *AddAt* accetta come argomenti l'oggetto da aggiungere e l'indice che questo avrà nella collezione; l'indice deve essere compreso tra 1 e *Sz*, altrimenti scatta un errore di esecuzione 213 (codice identico a quello dell'errore «Collection index out of range» del Turbo Vision); se la condizione è rispettata, all'elemento con l'indice specificato viene assegnato l'indirizzo dell'oggetto. Il metodo *At* accetta invece come unico argomento l'oggetto da aggiungere e ne assegna l'indirizzo al primo elemento libero; se non ve ne sono, la collezione viene espansa chiamando il metodo *Grow*. Mentre *At* ritorna l'indirizzo del suo argomento, *AddAt* ritorna quello dell'oggetto eventualmente rimpiazzato, dell'oggetto cioè il cui indirizzo fosse stato in precedenza assegnato all'elemento con indice pari al secondo argomento; in questo modo è possibile distinguere tra «aggiunta» e «sostituzione» di un elemento.

Ma il «vero» motivo è un altro.

Oggetti di proprietà

Forse ricorderete la prova congiunta del Turbo Pascal 5.5 e del Quick Pascal 1.0 della Microsoft (MC di settembre 1989). Misi tra l'altro in evidenza che le istanze di classi nel Quick Pascal, in quanto ispirato dall'Object Pascal del Mac, dovevano essere sempre create con *New*; in altri termini, tutte le variabili appartenenti ad un tipo **object** del Quick Pascal erano variabili dinamiche,

```
unit IndxColl;

interface

uses Base, Collect;

type
  PIndexedCollection = ^TIndexedCollection;
  TIndexedCollection = object(TCollection)
    constructor Init(Size, Delta: word);
    destructor Done; virtual;
    function Name: string; virtual;
    function InitIterator: PIterator; virtual;
    function Add(var o: TObject): PObject; virtual;
    function Remove(var o: TObject): PObject; virtual;
    function At(i: word): PObject;
    function AddAt(var o: TObject; i: word): PObject; virtual;
    function RemoveAt(i: word): PObject; virtual;
    function GetSize: word;
    procedure Grow;
  private
    A: PObjPArray;
    Sz, Dlt: word;
  end;
  PIndexedCollectionIterator = ^TIndexedCollectionIterator;
  TIndexedCollectionIterator = object(TIterator)
    constructor Init(var c: TIndexedCollection);
    destructor Done; virtual;
    function More: boolean; virtual;
    function Next: PObject; virtual;
  private
    IdxColl: PIndexedCollection;
    CurrentIndex: word;
  end;

implementation

constructor TIndexedCollection.Init(Size, Delta: word);
var
  i: word;
begin
  Sz := Size;
  Dlt := Delta;
  GetMem(A, Sz * SizeOf(PObject));
  for i := 1 to Sz do A^[i] := Null;
end;

destructor TIndexedCollection.Done;
begin
  FreeMem(A, Sz * SizeOf(POBject));
end;

function TIndexedCollection.Name: string;
begin
  Name := 'IndexedCollection';
end;

function TIndexedCollection.InitIterator: PIterator;
var
  Iterator: PIndexedCollectionIterator;
begin
  New(Iterator, Init(Self));
  InitIterator := PIterator(Iterator);
end;

function TIndexedCollection.Add(var o: TObject): PObject;
var
  i: word;
begin
  i := 1;
  while (i <= Sz) and (A^[i] <> PObject(Null)) do Inc(i);
  if i > Sz then Grow;
  A^[i] := @o;
  Add := A^[i];
end;

function TIndexedCollection.Remove(var o: TObject): PObject;
var
  P: PObject;
  i: word;
begin
  for i := 1 to Sz do
    if A^[i]^.IsEqual(o) then begin
      P := A^[i];
      A^[i] := Null;
      Remove := P;
      Exit;
    end;
  Remove := Null;
end;

function TIndexedCollection.At(i: word): PObject;
begin
  if (i < 1) or (i > Sz) then RunError(213);
  At := A^[i];
end;

function TIndexedCollection.AddAt(var o: TObject; i: word): PObject;
var
  P: PObject;
begin
  if (i < 1) or (i > Sz) then RunError(213);
  P := A^[i];
  A^[i] := @o;
  AddAt := P;
end;

function TIndexedCollection.RemoveAt(i: word): PObject;
var
  P: PObject;
begin
  if (i < 1) or (i > Sz) then RunError(213);
  P := A^[i];
  A^[i] := Null;

  RemoveAt := P;
end;


function TIndexedCollection.GetSize: word;
begin
  GetSize := Sz;
end;

procedure TIndexedCollection.Grow;
var
  NewA: PObjPArray;
  NewSize, i: word;
begin
  if Dlt = 0 then RunError(214);
  NewSize := Sz + Dlt;
  GetMem(NewA, NewSize * SizeOf(PObject));
  for i := 1 to Sz do NewA^[i] := A^[i];
  for i := Sz+1 to NewSize do NewA^[i] := Null;
  FreeMem(A, Sz * SizeOf(PObject));
  A := NewA;
  Sz := NewSize;
end;

constructor TIndexedCollectionIterator.Init(var c:TIndexedCollection);
begin
  IdxColl := @c;
  CurrentIndex := 1;
end;

destructor TIndexedCollectionIterator.Done;
begin
end;

function TIndexedCollectionIterator.More: boolean;
begin
  More := CurrentIndex <= IdxColl^.Sz;
end;

function TIndexedCollectionIterator.Next: PObject;
begin
  if CurrentIndex <= IdxColl^.Sz then begin
    Next := IdxColl^.A^[CurrentIndex];
    Inc(CurrentIndex);
  end
  else
    Next := Null;
end;

end.
```

Figura 1 - La unit INDXCOLL, che definisce le collezioni indicizzate.

```
unit Numbers;

interface

uses Base, Magnitud;

type
  PNumber = ^TNumber;
  TNumber = object(TMagnitude)
    constructor Init(n: real; Decimals: word);
    destructor Done; virtual;
    function Name: string; virtual;
    function Hash: word; virtual;
    function IsEqual(var o: TObject): boolean; virtual;
    function IsLessThan(var m: TMagnitude): boolean; virtual;
    procedure PrintOn(var f: text); virtual;
    function Val: real;
  private
    v: real;
    d: word;
  end;

implementation

constructor TNumber.Init(n: real; Decimals: word);
begin
  v := n;
  d := Decimals;
end;

destructor TNumber.Done;
begin
end;

function TNumber.Name: string;
begin
  Name := 'Number';
end;

function TNumber.Hash: word;
var
  x: real;
  f: array[1..3] of word absolute x;
begin
  x := v;
  Hash := f[1] + f[2] + f[3];
end;

function TNumber.IsEqual(var o: TObject): boolean;
begin
  IsEqual := (TypeOf(o) = TypeOf(Self)) and (v = (PNumber(@o))^.v);
end;

function TNumber.IsLessThan(var m: TMagnitude): boolean;
begin
  IsLessThan := v < (PNumber(@m))^.v;
end;
procedure TNumber.PrintOn(var f: text);
begin
  Write(f, v:0:d);
end;

function TNumber.Val: real;
begin
  Val := v;
end;

end.
```

◄ *Figura 2*
La unit NUMBERS, che definisce i numeri come classe derivata da TMagnitude.

```
program IndxDemo;
(*$X+*)
uses Base, Numbers, IndxColl;
var
  A: array[1..10] of TNumber;
  F1, F2: TIndexedCollection;
  i: integer;
begin
  Writeln(MemAvail);
  F1.Init(5,5);
  F2.Init(5,5);
  F1.PrintOn(StdOut);
  F1.PrintOn(StdOut);
  if F1.IsEqual(F2) then Writeln('Uguali') else Writeln('Diverse');
  for i := 1 to 6 do begin
    A[i].Init(i,0);
    F1.Add(A[i]);     (* if F1.Add(A[i]) = nil then ; *)
    F2.Add(A[i]);     (* if F2.Add(A[i]) = nil then ; *)
  end;
  F1.PrintOn(StdOut);
  F2.PrintOn(StdOut);
  if F1.IsEqual(F2) then Writeln('Uguali') else Writeln('Diverse');
  F1.Remove(A[4]);  (* if F1.Remove(A[4]) = nil then ; *)
  F1.PrintOn(StdOut);
  F2.PrintOn(StdOut);
  if F1.IsEqual(F2) then Writeln('Uguali') else Writeln('Diverse');
  F1.Done;
  F2.Done;
  Writeln(MemAvail);
end.
```

▲ *Figura 3*
Un esempio di uso delle collezioni indicizzate.

mentre il Turbo Pascal consente (grazie ai constructor mutuati dal C++) anche variabili allocate nel data segment al momento della compilazione. La gerarchia di classi che ho preparato per voi sfrutta appieno questa maggiore flessibilità: ad una collezione possono essere aggiunti oggetti sia «statici» che dinamici, sia variabili dichiarate in una sezione **var** che variabili allocate con *New*.

Anche in questo caso, tuttavia, si tratta solo di una delle diverse scelte possibili. La «class library» del Borland C++ e la gerarchia del Turbo Vision seguono infatti un approccio diverso: ogni oggetto aggiunto ad una collezione deve essere allocato con *New*, anche solo per fare una copia «dinamica» di un oggetto altrimenti «statico». Nel gergo di Scotts Valley, ciò si richiede perché ogni oggetto aggiunto ad una collezione diventa di proprietà della collezione.

Sarò sincero: non è un punto di vista che mi sento di condividere pienamente. La Borland sostiene che, in caso contrario, si potrebbe distruggere (deallocare) un oggetto già aggiunto ad una collezione, lasciando così questa in uno stato «molto confuso». In realtà è sempre possibile distruggere un oggetto appartenente ad una collezione, tanto che la Borland raccomanda espressamente di non farlo, ma di chiedere alla collezione stessa di distruggere l'oggetto (ad esempio, nel Turbo Vision, con il metodo *FreeItem*); anche in questo modo, però, si avrebbero problemi se si aggiungessero ad una collezione oggetti già appartenenti ad un'altra (il problema si presenta soprattutto con il C++, che fa abbondante uso dei cosiddetti «copy constructor», con i quali è possibile creare, tra l'altro, una collezione che sia la copia di un'altra): chiedendo ad una di distruggere un suo oggetto, sarebbe l'altra a restare in uno stato «molto confuso».

Preferirei dire che l'approccio Borland è solo più comodo. Se tutti gli oggetti da aggiungere ad una collezione vengono appositamente allocati per essa, al punto da poter essere considerati «di sua proprietà», si semplifica il recupero della memoria occupata dalla collezione quando di essa non abbiamo più bisogno.

Torniamo al destructor di *TIndexedCollection*: nel nostro caso si limita a rilasciare la memoria occupata dall'array di puntatori ad oggetti, senza minimamente curarsi degli oggetti «puntati»; se questi fossero stati allocati dinamicamente ad uso esclusivo della collezio-

```
486120
IndexedCollection(Nil,Nil,Nil,Nil,Nil)
IndexedCollection(Nil,Nil,Nil,Nil,Nil)
Uguali
IndexedCollection(1,2,3,4,5,6,Nil,Nil,Nil,Nil)
IndexedCollection(1,2,3,4,5,6,Nil,Nil,Nil,Nil)
Uguali
IndexedCollection(1,2,3,Nil,5,6,Nil,Nil,Nil,Nil)
IndexedCollection(1,2,3,4,5,6,Nil,Nil,Nil,Nil)
Diverse
486120
```

Figura 4
L'output del programma di figura 3.

ne, andrebbero distrutti prima di chiamare *TIndexedCollection.Done*; in caso contrario, continuerebbero ad occupare inutilmente memoria nello heap, in quanto i puntatori a loro sarebbero stati distrutti.

Nella «class library» del Borland C++ e nella gerarchia del Turbo Vision, invece, il destructor di una collezione distrugge per prima cosa i suoi elementi, semplificando quindi in qualche modo la gestione della memoria da parte del programmatore. Il prezzo non è alto: è solo da evitare di aggiungere ad una collezione oggetti che non siano stati creati con *New*.

Alle nostre collezioni è possibile aggiungere anche oggetti «statici» (come vedremo tra breve), e ricade quindi sul programmatore il compito di distruggere, se necessario, gli eventuali oggetti «dinamici». È questo il motivo per cui *AddAt*, come anche *Remove* e *RemoveAt* (che tolgono dalla collezione, rispettivamente, un dato oggetto o l'oggetto ad un dato indice), ritornano l'indirizzo dell'oggetto sostituito o rimosso: per consentirci di distruggerlo se si trattava di un oggetto allocato dinamicamente di cui non abbiamo più bisogno.

Ho insistito su questo aspetto perché, a differenza di quanto avviene in Smalltalk (che è dotato di meccanismi di *garbage collection*), nel C++ e nel Turbo Pascal con oggetti la gestione della memoria dinamica rimane interamente a carico del programmatore; è necessario quindi essere ben consapevoli delle implicazioni delle diverse possibili implementazioni, per evitare sia errori provocati da puntatori che non puntano più a nulla, sia sprechi di memoria causati da blocchi prima allocati e poi non più «puntati» da alcun puntatore.

L'iteratore

Come ho già detto più volte, il vantaggio degli iteratori è rappresentato dalla possibilità di eseguire operazioni su ognuno degli elementi di una collezione, anche di quelle non indicizzate come i set (normalmente non «iterabili»). Sembrerebbe quindi inutile prevedere un iteratore anche per le collezioni indicizzate. In realtà, a parte considerazioni di coerenza ed omogeneità, la presenza di un iteratore consente di godere di uno dei principali vantaggi della OOP, la riusabilità del codice. La unit INDXCOLL, infatti, può limitarsi ad ereditare senza ridefinirli (e senza nemmeno nominarli) i metodi *IsEqual* e *PrintOn* della classe *TCollection*, solo ridefinendo il metodo *InitIterator*.

L'implementazione della classe *TIndexedCollectionIterator* non mi pare richieda particolari commenti. Sottolineo solo che il constructor richiede come argomento la collezione da sottoporre a iterazione, il cui indirizzo viene assegnato alla variabile d'istanza *IdxColl*. Il metodo *InitIterator* di *TIndexedCollection* chiama quindi quel constructor passandogli *Self* come argomento, ne ottiene un puntatore all'iteratore, e ritorna un puntatore alla classe astratta degli iteratori mediante un *cast*.

Il breve demo che vedremo tra un attimo è anche un esempio di uso degli iteratori, anche se questi non si vedono: il loro uso è infatti implicito nelle chiamate dei metodi *IsEqual* e *PrintOn*.

Prima di vedere le collezioni all'opera, tuttavia, è necessario derivare da *TObject* classi di oggetti «concreti» per avere qualcosa da aggiungere ad una collezione.

Come ho già detto più volte, la nostra gerarchia cerca di rispettare la filosofia dello Smalltalk, in cui tutto (o quasi) deriva da un'unica classe base. Non vi meraviglierete quindi se, come in Smalltalk, deriviamo da *TObject*, passando per *TMagnitude*, anche i numeri.

Un esempio numerico

La unit NUMBERS (figura 2) definisce i numeri come grandezze confrontabili, quindi come classe derivata da *TMagnitude*.

In Smalltalk (e anche nella gerarchia di classi di Gorlen) vi sono diverse classi numeriche; in considerazione della concisione che ci siamo imposti, la classe *TNumber* raccoglie in sé sia numeri interi che reali in virgola fissa; il constructor prevede infatti due argomenti: il valore e il numero di decimali; quest'ultimo, che sarà zero per gli interi, viene usato solo dal metodo *PrintOn*.

L'implementazione è piuttosto semplificata, come si conviene ad una classe che vi propongo solo a titolo di esempio. È decisamente poco elegante, in particolare, il metodo *Hash*, che calcola un valore intero come somma dei sei byte della rappresentazione del tipo *real* in Turbo Pascal.

Nella figura 3 vediamo appunto un esempio di uso delle classi fin qui illustrate. Notiamo subito l'uso della direttiva $X: si tratta di una innovazione introdotta con il Turbo Pascal 6.0 che consente di trattare le funzioni come procedure, come se non ritornassero alcun valore. Abbiamo visto che i metodi *Add* e *Remove* ritornano l'indirizzo dell'oggetto aggiunto o rimosso dalla collezione, di cui però non abbiamo bisogno nel programma INDXDEMO; è quindi effettivamente comodo poter trattare tali metodi come se fossero implementati come procedure. Chi avesse ancora il Pascal 5.5 può comunque usare la più pesante sintassi mostrata nei commenti al listato.

Il programma non fa grandi cose: dichiara un array di *TNumber* i cui valori vengono aggiunti contemporaneamente a due collezioni indicizzate; da una di esse viene poi rimosso uno dei numeri. Successivi invii dei messaggi *IsEqual* e *PrintOn* consentono di monitorare le diverse fasi, come illustrato nella figura 4.

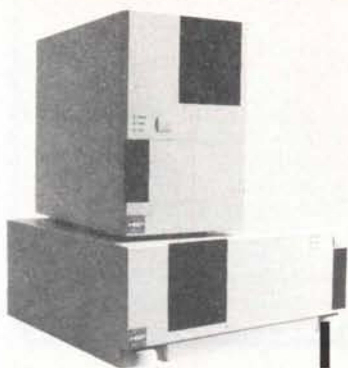
Vorrei sottolineare solo due aspetti. Le collezioni vengono ambedue dichiarate con una dimensione di 5 e un delta di 5; i primi 5 numeri vengono quindi aggiunti senza ricorso al metodo *Grow*, che invece viene invocato quando viene aggiunto il sesto numero. Poiché il delta è 5, la dimensione delle collezioni aumenta a 10, come potete verificare dalla figura 4.

Vediamo così in azione quel meccanismo di ottimizzazione della «espansione» delle collezioni cui avevo fatto cenno prima (e che ritroviamo nel Turbo Vision): l'aggiunta di un settimo, ottavo, nono e decimo numero non richiederebbe ulteriori chiamate di *Grow*.

Vi segnalo infine le due istruzioni *Writeln(MemAvail)* all'inizio e alla fine del programma. Ho cercato di spiegare sopra i possibili trabocchetti nella gestione della memoria; sia in C++, con i suoi constructor e destructor automatici, che in Turbo Pascal, con collezioni di oggetti sia dinamici che qualunque, può capitare con relativa facilità (parlo per esperienza...) di dimenticare di rilasciare memoria dinamica allocata ma non più necessaria. Durante lo sviluppo di una unit o di un programma può quindi risultare utile tenere d'occhio l'uso dello heap con quelle semplici istruzioni, o magari con il più sofisticato meccanismo messoci a disposizione dalla unit GADGETS del Turbo Vision.

È solo un consiglio.

MC



COMPUTER

HSP

COMPUTER

I CAMPIONI DI POTENZA

DESIGNER - 21

AT 16/21 MHz
da L. 550.000
512K FDD1.2 RS232 PRINTER

DESIGNER SX

386 SX
da L. 949.000
512K FDD1.2 RS232 PRINTER

CAD-25

386 25 MHz
da L. 1.299.000
1MB FDD1.2 RS232 PRINTER

PROCAD-33

386 33 MHz
da L. 1.890.000
64K CACHE 1MB FDD1.2 RS232

IPERCAD-486

486 25 MHz
da L. 3.790.000
1MB FDD1.2 RS232 PRINTER

COPROCESSORI

80287-10	L.	160.000
80387-SX	L.	420.000
80387-25	L.	540.000
80387-33	L.	690.000

HARD DISK

20MB 40ms 3,5" MFM	L.	279.000
40MB 24-28ms 3,5" IDE	L.	350.000
80MB 15ms 3,5" IDE	L.	690.000
120MB 16ms 3,5" IDE	L.	790.000
210MB 15ms 3,5" IDE		TELEF.
330MB 16ms 5,25" SCSI		TELEF.
660MB 16ms 5,25" SCSI		TELEF.
1200MB 16ms 5,25" SCSI		TELEF.
CTRL 2 FDD+2HDD IDE	L.	45.000
CTRL 2 FDD+2HDD SCSI		TELEF.

SCHEDE GRAFICHE

SK. VGA 15 BIT 256K	L.	99.000
SK. VGA 16 BIT 512K	L.	140.000
SK. VGA 16 BIT 1MB TSENG	L.	220.000
SK.U VGA TMS34010		TELEF.
SK. UVGA COMP. 8514/A		TELEF.

OFFERTISSIMA

S.G.VGA 16 BIT 1MByte
con MON. 14" VGA 1024x768

L. 749.000

SOFTWARE

APPLICATIVI PER WINDOWS			
WINDOWS 3.0	ING.	L.	189.000
WORD PER WINDOWS	ING.	L.	390.000
EXCEL 2.2 PER WINDOWS	ING.	L.	570.000
TOOLBOOK PER WINDOWS	ING.	L.	540.000
WORD PROCESSOR			
WORD 5	ING.	L.	390.000
WORDSTAR 6.0	ING.	L.	490.000
QUATTRO PRO 2.0	ING.	L.	590.000
LINGUAGGI			
TURBO BASIC	ING.	L.	120.000
QUICK BASIC	ING.	L.	115.000
TURBO C	ING.	L.	249.000
QUICK C 2.3	ING.	L.	115.000
TURBO PASCAL 5.5	ING.	L.	180.000
TURBO PROLOG	ING.	L.	180.000
DATA BASE			
DATA BASE 4.2	ING.	L.	842.000
DBASE IV 1.1	ING.	L.	890.000
FOX BASE + 286 2.1	ING.	L.	349.000
FOX BASE + 386 2.1	ING.	L.	549.000
PARADOX 3.0	ING.	L.	839.000
CAD			
AUTO SKETCH V. 2.0	ING.	L.	160.000
AUTODESK ANIMATOR	ING.	L.	430.000
DESIGN CAD 2D 4.0	ING.	L.	275.000
DESIGN CAD 3D 3.0	ING.	L.	365.000
EASY CAD 2.3	ING.	L.	189.000
COMMUNICATIONS			
CARBON COPY PCPLUS	ING.	L.	189.000
CROSS TALK XVI	ING.	L.	195.000
LAPLINK III	ING.	L.	165.000
PRO COM PLUS 1.18	ING.	L.	119.000
DESKTOP PUBLISHING			
OMNIPAGE 386	ING.		TELEF.
PAGEMAKER 3.01	ING.		TELEF.

MONITOR

VGA 14" MONO PW	L.	200.000
VGA 14" COLORE	L.	450.000
VGA 1024x768		TELEF.
NEC 3D 1024x768	L.	990.000
NEC 4D		TELEF.
NEC 5D		TELEF.
VGA 17" 1024x763 FLAT	L.	1.500.000

ACCESSORI

MOUSE 800DPI, ITALIANO	L.	59.000
HANDY SCANNER LOGITECH da	L.	290.000
HANDY SCANNER COLORI		TELEF.
SCANNER MONO PIANO FISSO A4		TELEF.
SCANNER COLORI PIANO FISSO A4		TELEF.
TAVOLETTE GRAFICHE 12"x12"	L.	390.000
TAVOLETTE GRAFICHE 12"x18"	L.	750.000
DISCHETTI 1,44MB	L.	1.600
DISCHETTI 720KB	L.	900
FAX SAMSUNG CON TELEFONO	L.	890.000

NOTEBOOK
A4 kg. 2,8

286	IMB HD20 VGA	2.900.000
286	IMB HD40 VGA	L. 3.200.000
386SX	IMB HD20 VGA	L. 3.300.000
386SX	IMB HD40 VGA	L. 3.400.000

MODEM

SK. 300/2400	L.	159.000
EST. 300/2400	L.	210.000
SK. MOD. 2400 FAX G2	L.	290.000
SK. MOD. 2400 FAX G3	L.	390.000

STAMPANTI

CITIZEN 1200 PLUS	80C 120S 9A	L.	269.000
EPSON LX400	80C 150S 9A	L.	350.000
SAMSUNG	80C 300S 9A	L.	380.000
EPSON LX1050	80C 162S 9A	L.	650.000
CITIZEN MSP15E	136C 160S 9A	L.	480.000
CITIZEN PRODOT9X	136C 3000S 9A	L.	750.000
EPSON FX 1050	136C 220S 9A	L.	843.000
CITIZEN 124D	80C 144S 9A		TELEF.
CITIZEN SWIFT 24	80C 192S 24A		TELEF.
EPSON LQ400	80C 150S 24A	L.	540.000
PANASONIC KX 1123 24A		L.	480.000
NEC P20	80C 192S 24A	L.	570.000
EPSON LQ860	80C 200S 24A	L.	1.208.000
NEC P60	80C 268S 24A	L.	990.000
SAMSUNG 136C	220C 24A	L.	990.000
CITIZEN SWIFT 24X	136C 192S 24A		TELEF.
EPSON LQ1050+	136C 220S 24A	L.	1.208.000
NEC P70	136C 268C 24A	L.	1.250.000
EPSON LQ1060	136C 300C 24A	L.	1.519.000
TEXAS INSTRUMENTS A4 6PPM LASER		L.	1.990.000
EPSON EPL7100 A4 6PPM LASER		L.	1.750.000
EPSON EPJ200 A3 2PPM INK JET		L.	2.058.000

CONCESSIONARIO SU ROMA

CENTRO ASSISTENZA
TECNICA PC.
PROGETTAZIONE
RETI LOCALI

Via Malta 8 • 00198 Roma
Tel. (06) 8842378/8411987/8450338
dal Lun. al Sab. 9.00-13.00 / 16.30-19.30

GARANZIA 12 MESI - PREZZI IVA ESCLUSA