di Sergio Polini

Matrici dinamiche

Tempo fa, su MC-Link, un utente aveva chiesto come realizzare matrici dinamiche in Turbo Pascal, cioè matrici le cui dimensioni potessero essere decise al momento dell'esecuzione invece che già in fase di compilazione. Si tratta di un argomento interessante, anche perché ho avuto modo di constatare che non tutti sanno che vi sono almeno due diversi metodi. Vi propongo quindi le due soluzioni, approfittando di questa «duplicità» per mostrare anche come realizzare moduli (unit) la cui interfaccia sia effettivamente indipendente dalla implementazione

Nel numero scorso vi ho proposto la prima parte di una breve «introduzione alla programmazione orientata all'oggetto»; mettendo a nostra disposizione il Turbo C++ 1.0 e il Turbo Pascal 6.0, con il suo Turbo Vision, la Borland ci ha invitato in modo fin troppo convincente a guardare più da vicino a quelle nuove avanzate tecniche di programmazione che già il Turbo Pascal 5.5 aveva messo alla portata di tutti. Ho cercato di mostrare come nella OOP ritornino, potenziati, i meccanismi tipici della programmazione modulare, in particolare l'*information hiding*, ovvero la creazione di moduli in cui rimangano nascosti i dettagli della implementazione, essendo consentito solo un accesso mediante apposite funzioni di interfaccia. In questo numero il discorso è continuato fino a trattare di ereditarietà e polimorfismo; a partire da gennaio la rubrica cercherà di portarvi pian piano alla programmazione orientata all'oggetto con esempi «sul campo»: non si tratta tanto di affrontare argomenti «difficili», quanto piuttosto di convincervi che vale la pena di mettere un po' in discussione abitudini magari da tempo consolidate. Intanto, quasi per preparare i nuovi argomenti, vi propongo un esempio di *information hiding*, con relativi benefici: come cambiare radicalmente l'implementazione di un tipo di dato senza per questo dover riscrivere il programma che ne fa uso.

Somme e moltiplicazioni

Una matrice, quante siano le sue dimensioni, viene sempre rappresentata in forma lineare: un array bidimensionale viene mantenuto in memoria come un array ad una sola dimensione, di tanti elementi quant'è il prodotto delle righe per le colonne. Una matrice 4x4 viene quindi rappresentata come un array di 16 elementi. Sono in realtà possibili due diverse implementazioni, dette *row-major* e *column-major*: nel primo caso troviamo prima gli elementi della prima riga, poi quelli della seconda, ecc.; nel secondo si va di colonna in colonna (figura 1). In Pascal si adotta la prima soluzione.

Mentre scriviamo il nostro programma, usiamo ovviamente una notazione che ci consente di ragionare in termini del tipo di dato così come l'abbiamo dichiarato: se vogliamo usare un array [1..3,1..4] di interi chiamato A, possiamo accedere al terzo elemento della seconda riga con A[2][3], o ancora più semplicemente con A[2,3] (la prima notazione significa «il terzo elemento del secondo elemento di un array di 3 array, ognuno dei quali è un array di quattro integer», e discende dritta dritta dalla rappresentazione *row-major*). Come si fa a tradurre quelle notazioni in qualcosa che possa agire su un'area di memoria in cui il nostro dato è rappresentato in maniera diversa? Ci pensa il compilatore; quel [2,3] diventa [(2-1)*4+3], cioè: si accede all'array «vero», che è ad una sola dimensione, con un indice dato dalla somma dell'indice di colonna e del prodotto del numero delle colonne per l'indice di riga diminuito di uno. Ad esempio, nel nostro caso, si usa un indice pari a 7, in quanto l'elemento [2,3] è preceduto dai quattro elementi della prima riga e da due elementi della seconda (ho un po' semplificato, in quanto le cose si complicano quando gli indici

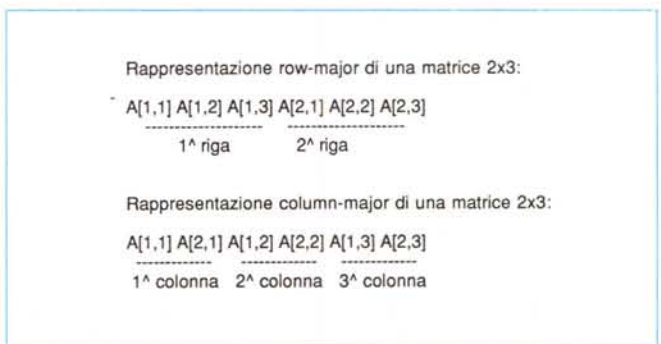

Figura 1 - Un array, anche multidimensionale, viene sempre rappresentato in memoria come un array monodimensionale; sono possibili due diverse rappresentazioni, «riga per riga» o «colonna per colonna». Il Pascal usa la prima.

di riga e colonna non partono da 1, e soprattutto quando le dimensioni sono più di due; a chi volesse approfondire, segnalo: Alfred V. Aho, Ravi Sethi e Jeffrey D. Ullman, *Compilers. Principles, Techniques and Tools*, Addison-Wesley, 1986; oppure Ellis Horowitz e Sartaj Sahni, *Fundamentals of Data Structures in Pascal*, Pitman, 1984).

Vi prego di notare che qui ritroviamo proprio un esempio di *information hiding*: il compilatore ci «nasconde» la rappresentazione di un tipo di dato, offrendoci al tempo stesso una «interfaccia» attraverso la quale da un lato possiamo usare una notazione comoda e chiara, dall'altro potremmo compilare senza problemi i nostri programmi con un compilatore che adottasse una rappresentazione anche molto diversa da quella che vi ho descritto.

Volendo ora «aggiungere al linguaggio» un tipo di dato che non c'è, la soluzione più immediata consiste nel riprodurre con qualche variazione quei meccanismi: dichiareremo un tipo «matrice dinamica» come array monodimensionale che, per poter assumere le dimensioni volute al momento dell'esecuzione, dovrà necessariamente essere allocato nello heap; creeremo poi una interfaccia che consenta di accedere ad un qualsiasi suo elemento designandolo con gli indici di riga e di colonna, «astraendo» dal fatto che in realtà la dimensione «vera» è una sola.

Il tutto si può presentare più o meno come la unit MATRIX1 che vi propongo in figura 2; non mi pare che il codice abbia bisogno di particolari commenti, se non per un aspetto. La funzione *MatElem* viene usata per avere accesso a un elemento della matrice, dati i valori degli indici di riga e di colonna; potrebbe ritornare un intero, da interpretare come l'indice dell'array monodimensionale che viene usato per rappresentare in memoria la matrice, ma ciò comporterebbe due inconvenienti. In primo luogo, si darebbe accesso alla implementazione, che invece vogliamo tenere nascosta (e vedremo tra breve perché); rimarrebbe poi da operare per l'accesso vero e proprio, ad esempio con una istruzione del tipo "M.Dati^ [MatElem(M,r,c)]"; tanto vale accedere subito all'elemento. La funzione non può però ri-

Figura 2 - Un primo approccio alla realizzazione di matrici dinamiche: un array di (righe × colonne) elementi allocato nello heap, con una funzione di accesso che converte gli indici di riga e colonna in un indice per l'array.

```
Unit Matrix1;

interface

type
  TRealArray = array[1..65520 div SizeOf(real)] of real;
  PRealArray  = ^TRealArray;
  TRealMat = record
    NumRighe, NumColonne: integer;
    Dati: PRealArray;
  end;
  PReal = ^real;

procedure InitMat(var M: TRealMat; r, c: integer);
function MatElem(var M: TRealMat; r, c: integer): PReal;
procedure GetMat(var M: TRealMat);
procedure WriteMat(var M: TRealMat);
procedure FreeMat(var M: TRealMat);

implementation

procedure InitMat(var M: TRealMat; r, c: integer);
begin
  GetMem(M.Dati, r * c * sizeof(real));
  M.NumRighe := r;
  M.NumColonne := c;
end;

function MatElem(var M: TRealMat; r, c: integer): PReal;
begin
  MatElem := Addr(M.Dati^[(M.NumColonne * (r - 1)) + c]);
end;

procedure GetMat(var M: TRealMat);
var
  i, j: integer;
begin
  for i := 1 to M.NumRighe do
    for j := 1 to M.NumColonne do begin
      Write('[',i,',',j,']: '); Readln(MatElem(M,i,j)^);
    end;
end;

procedure WriteMat(var M:TRealMat);
var
  i, j: integer;
begin
  for i := 1 to M.NumRighe do begin
    for j := 1 to M.NumColonne do
      Write(MatElem(M,i,j)^:9:5);
    Writeln;
  end;
end;
procedure FreeMat(var M: TRealMat);
begin
  FreeMem(M.Dati, M.NumRighe * M.NumColonne * sizeof(real));
  M.NumRighe := 0;
  M.NumColonne := 0;
  M.Dati := nil;
end;

end.
```

tornare direttamente l'elemento, in quanto se così si facesse si potrebbero leggere i vari elementi, ma non si saprebbe come assegnare loro dei valori. La funzione ritorna quindi un puntatore ad un elemento, che può essere usato sia per «lettura» che per «scrittura», come esemplificato nelle procedure *GetMat* e *WriteMat*.

Sistemi di equazione

Proviamo ora se funziona. Il programma in figura 3 risolve sistemi di equazioni mediante il metodo di eliminazione di Gauss. Il blocco principale chiede quante sono le equazioni, per poi creare sia la matrice dei coefficienti che il vettore dei termini noti. I relativi valori vengono immessi mediante la procedura *GetMat* della unit MATRIX1, i calcoli vengono eseguiti da una procedura *Gauss* che assegna ad un vettore *Soluzione* i valori delle variabili.

Non ho ora lo spazio (né questa è la sede più opportuna) per illustrare il funzionamento dell'algoritmo. Vi rimando quindi al testo di Alan Miller, *Programmi scientifici in Pascal* (Gruppo Editoriale Jackson, 1983), dove potrete trovare anche metodi più efficaci. Vi segnalo invece alcuni aspetti della implementazione. Ad esempio, il programma non usa solo matrici, ma anche vettori dinamici: per i termini noti e per le soluzioni. L'uso di essi è piuttosto semplice, e può avvalersi di tipi di dati dichiarati nella stessa unit MATRIX1.

Facciamo però un po' di conti. La dimensione massima dell'array è data dal numero di *real* che occupano complessivamente l'area massima che un tipo di dato può avere in Turbo Pascal (65520 byte), cioè 10920 numeri. Si può quindi lavorare con matrici di 104 righe e 104 colonne. Non male. Supponiamo però di aver bisogno di matrici tridimensionali; in questo caso, il massimo cui potremmo arrivare sarebbe 22x22x22 (andrebbe inoltre ovviamente riscritta la funzione *MatElem*). I limiti sarebbero poi un po' più bassi se si volessero usare numeri *double* o *extended* invece dei *real*; con gli *extended* avremmo un massimo di 80x80 per matrici bidimensionali e di 18x18x18 per matrici tridimensionali. Potremmo quindi aver bisogno, prima o poi, di una diversa implementazione.

Array di puntatori ad array

La figura 4 vi propone proprio questo. Accanto ai tipi *TRealArray* e *PRealArray* (rispettivamente: array di *real* e puntatore ad *array* di *real*), viene dichiarato un tipo *TPRealArray*: array di puntatori ad array di *real*. Una matrice bidimensionale può essere rappresentata infatti anche come un array di tanti puntatori

```
Program SistEq;

uses Matrix1;

var
  CoefMat  : TRealMat;
  NotiVec  : PRealArray;
  Soluzione: PRealArray;
  Errori   : boolean;
  i, n     : integer;

procedure Gauss(var CMat:TRealMat; NVec, SVec:PRealARray; var
Errori:boolean);
var
  TempMat   : TRealMat;
  TempArray : PRealArray;
  Temp, Max : real;
  i,j,k,l,m,n: integer;
begin
  Errori := FALSE;
  n := CMat.NumRighe;
  InitMat(TempMat, n, n);
  GetMem(TempArray, n * SizeOf(real));
  for i := 1 to n do begin
    for j := 1 to n do
      MatElem(TempMat,i,j)^ := MatElem(CMat,i,j)^;
    TempArray^[i] := NVec^[i];
  end;
  for i := 1 to n - 1 do begin
    Max := Abs(MatElem(TempMat,i,i)^);
    l := i;
    m := i + 1;
    for j := m to n - 1 do begin
      if Abs(MatElem(TempMat,j,i)^) > Max then begin
        Max := Abs(MatElem(TempMat,j,i)^);
        l := j;
      end;
    end;
    if Max = 0.0 then Errori := TRUE
    else begin
      if l <> i then begin
        for j := 1 to n do begin
          Temp := MatElem(TempMat,l,j)^;
          MatElem(TempMat,l,j)^ := MatElem(TempMat,i,j)^;
          MatElem(TempMat,i,j)^ := Temp;
        end;
        Temp := TempArray^[l];
        TempArray^[l] := TempArray^[i];
        TempArray^[i] := Temp;
      end;
      for j := m to n do begin
        Temp := MatElem(TempMat,j,i)^ / MatElem(TempMat,i,i)^;
        for k := m to n do
          MatElem(TempMat,j,k)^ := MatElem(TempMat,j,k)^ - Temp *
                                   MatElem(TempMat,i,k)^;
        TempArray^[j] := TempArray^[j] - Temp * TempArray^[i];
      end;
    end;
  end;

  if MatElem(TempMat,n,n)^ = 0.0 then Errori := TRUE
  else begin
    SVec^[n] := TempArray^[n] / MatElem(TempMat, n, n)^;
    repeat
      Temp := 0.0;
      for j := i+1 to n do
        Temp := Temp + MatElem(TempMat,i,j)^ * SVec^[j];
      SVec^[i] := (TempArray^[i] - Temp) / MatElem(TempMat,i,i)^;
      Dec(i);
    until i = 0;
  end;
  FreeMem(TempArray, n * SizeOf(real));
  FreeMat(TempMat);
end;

begin
  Write('Quante equazioni? '); Readln(n);
  InitMat(CoefMat, n, n);
  GetMem(NotiVec, n * SizeOf(real));
  Writeln('Immissione matrice coefficienti:');
  GetMat(CoefMat);
  Writeln('Immissione vettore termini noti:');
  for i := 1 to n do begin
    Write('[',i,']: '); Read(NotiVec^[i]);
  end;
  Writeln;
  GetMem(Soluzione, n * SizeOf(real));
  Gauss(CoefMat, NotiVec, Soluzione, Errori);
  if Errori then Writeln('Errore: matrice singolare')
  else begin
    Writeln('Soluzione:');
    for i := 1 to n do
      Write(Soluzione^[i]:9:5);
    Writeln;
  end;
end.
```

Figura 3 - Un programma per la soluzione di sistemi di equazioni mediante il metodo di eliminazione di Gauss, con uso della unit MATRIX1.

quante sono le righe, ognuno dei quali punti ad un array di tanti *real* quante sono le colonne. Per ottenere che la dimensione della matrice venga determinata al momento dell'esecuzione, occorre anche ora far ricorso all'allocazione di memoria nello heap: si può usare un campo *Dati* che punti all'array di puntatori, analogo all'omonimo campo della unit MATRIX1, che puntava all'array di *real*.

Saranno diverse, e appena un po' più complicate, le procedure per la creazione e la distruzione di una matrice, in quanto in entrambi i casi occorrerà un loop per creare e distruggere gli array di *real* cui puntano i puntatori dell'array principale. Sarà invece più semplice la funzione *MatElem*, tanto che si potrà adottare una notazione che, se non fosse per un paio di circonflessi, sarebbe identica a quella «normale». Nel caso di matrici a più di due dimensioni, si tratterebbe di adottare tanti array di puntatori ad array quante sono le dimensioni meno una, e un array di *real* per l'ultima dimensione.

```
unit Matrix2;


interface

type
  TRealArray  = array[1..65520 div SizeOf(real)] of real;
  PRealArray  = ^TRealArray;
  TPRealArray = array[1..65520 div SizeOf(pointer)] of PRealArray;
  TRealMat    = record
    NumRighe, NumColonne: integer;
    Dati: ^TPRealArray;
  end;
  PReal     = ^real;

procedure InitMat(var M: TRealMat; r, c: integer);
function MatElem(var M: TRealMat; r, c: integer): PReal;
procedure GetMat(var M: TRealMat);
procedure WriteMat(var M: TRealMat);
procedure FreeMat(var M: TRealMat);

implementation

procedure InitMat(var M: TRealMat; r, c: integer);
var
  i: integer;
begin
  GetMem(M.Dati, SizeOf(PRealArray) * r);
  for i := 1 to r do
    GetMem(M.Dati^[i], SizeOf(real) * c);
  M.NumRighe := r;
  M.NumColonne := c;
end;

function MatElem(var M: TRealMat; r, c: integer): PReal;
begin
  MatElem := Addr(M.Dati^[r]^[c]);
end;

procedure GetMat(var M: TRealMat);
var
  i, j: integer;
begin
  for i := 1 to M.NumRighe do
    for j := 1 to M.NumColonne do begin
      Write('[',i,',',j,']: '); Readln(MatElem(M,i,j)^);
    end;
end;

procedure WriteMat(var M: TRealMat);
var
  i, j: integer;
begin
  for i := 1 to M.NumRighe do begin
    for j := 1 to M.NumColonne do
      Write(MatElem(M,i,j)^:9:5);
    Writeln;
  end;
end;

procedure FreeMat(var M: TRealMat);
var
  i: integer;
begin
  for i := 1 to M.NumRighe do
    FreeMem(M.Dati^[i], SizeOf(real) * M.NumColonne);
  FreeMem(M.Dati, SizeOf(PRealArray) * M.NumRighe);
  M.Dati := nil;
  M.NumRighe := 0;
  M.NumColonne := 0;
end;

end.
```

Figura 4 - Una diversa implementazione per le matrici dinamiche: un array di puntatori ad array di real. Le funzioni e procedure di interfaccia sono identiche a quelle della unit MATRIX1.

I vantaggi più immediatamente visibili di una tale soluzione sono due. Se abbiamo bisogno di matrici «grandi», possiamo allocare fino a 16380 puntatori nell'array di puntatori, fino a 10920 *real* nell'array di *real*. Anche troppi. È possibile inoltre che l'accesso agli elementi mediante puntatori risulti più veloce di quello mediante somme e moltiplicazioni. C'è soprattutto però un altro vantaggio che ci viene dall'aver «incapsulato» nella **implementation** delle unit MATRIX1 e MATRIX2 i dettagli dell'accesso ai singoli elementi. Immaginate di aver scritto un programma lungo e complicato che adotti la soluzione della unit MATRIX1. Avete adottato algoritmi delicati (ad esempio, inversione di matrici mediante il metodo di eliminazione di Gauss-Jordan, o soluzione di sistemi di equazioni con coefficienti complessi, anch'essi reperibili nel testo di Miller), avete faticosamente testato il programma evitando tutte le trappole degli imperfetti arrotondamenti dei numeri binari (ad esempio: se A vale 3.4 e B vale 0.05, è probabile che la differenza tra 3.45 e A+B non sia zero come dovrebbe essere), il programma funziona con piena soddisfazione dei suoi utenti. Poi accade che vi serve una matrice più grande, tanto da non «entrare» nel tipo *TRealMat* come definito in MATRIX1. Dovete riscrivere tutto? No, se avete effettivamente «nascosto» dietro la **interface** i dettagli della implementazione. Provare per credere. Nel caso del programma della figura 3, basta sostituire «uses Matrix1» con «uses Matrix2». Basta cambiare una riga. Se il programma sembrerà non funzionare più correttamente come prima, non dovrete ripetere la lunga e faticosa serie di test: basterà che vi limitiate ad assicurarvi del corretto funzionamento della sola nuova unit.

In altri termini, la *programmazione modulare* consente di ridurre i tempi di manutenzione e test di un programma. A partire da gennaio vedremo come la *programmazione orientata all'oggetto* offra benefici anche maggiori.

MC