di Sergio Polini

Liste auto-ordinate

Variabili che nascono dal nulla, variabili il cui valore non è un valore, ma un indirizzo, strutture di dati che si mantengono ordinate da sole e secondo l'ordine che più ci piace, ecc.: il mondo dell'allocazione dinamica della memoria è indubbiamente affascinante, e soprattutto mette a nostra disposizione tecniche di programmazione di notevole potenza senza le quali molti problemi non potrebbero trovare soluzione, salvo ricorrere a marchingegni complicati e inefficienti.
È tuttavia anche un mondo un po' misterioso per chi vi si accosti per la prima volta; basti pensare che tra le variabili dichiarate in un programma c'è solo una traccia minuscola di quelle strutture dinamiche che lo stesso programma è in grado di creare e gestire.
Cercheremo quindi di «svelare l'arcano»: non meri discorsi teorici, per i quali si possono trovare ottimi testi (ad esempio: Programmare in Pascal *di Peter Grogono, o il solito* Algoritmi + Strutture di dati = Programmi *di Wirth), ma un breve programma in cui inseriremo una nostra versione delle procedure New e Dispose del Pascal*

Alloc, il programma versione semplificata di uno proposto da Wirth, non fa altro che creare e gestire una «lista»; quelle due procedure, insieme ad una terza che chiameremo DumpMem, ci consentiranno di farci un'idea concreta di come vengono allocate e rilasciate variabili dinamiche, anche mediante una rappresentazione visiva di quello che accade nell' «heap», sostituito, per l'occasione, da un normale array di interi (i due file ALLOC.PAS e ALLOC.INC sono disponibili anche su MC-Link).
Le nostre New e Dispose non useranno gli stessi algoritmi del Turbo Pascal; adotteremo invece una variante degli algoritmi «A» e «B» di Knuth *(Fundamental Algorithms,* 2ª ed., pp. 435-440), in quanto sono i più semplici tra quelli non banali. Non ci dilungheremo sulle tecniche di «dynamic storage management», perché queste son cose che interessano più chi fa i compilatori che chi li usa; chi volesse un approfondimento può consultare il testo di Knuth, oppure anche i *Fundamentals of Data Structures in Pascal* di Horowitz e Sahni.

Statico e dinamico

Il manuale definisce «statiche» le variabili dichiarate come globali o locali in un programma; in realtà vi sono diversi gradi di staticità, visto che le variabili globali esistono per tutta l'esecuzione di un programma mentre quelle locali, come abbiamo visto la volta scorsa, vengono generate nello stack all'inizio di una funzione o procedura per poi venire scartate quando questa termina. Quello che si vuol dire è che di una variabile statica, globale o locale che sia, si conosce sempre l'indirizzo: la variabile globale Glbl1 del mese scorso rimane sempre nella sua locazione 260 del data segment, la variabile locale Lcl, una volta creata, sarà accessibile in BP-4 ogni volta che verrà creata.

Altro aspetto: le variabili statiche occupano sempre una stessa quantità di memoria. Se ho dichiarato un array di 10 interi e me ne servono invece 11 non posso farci nulla (salvo riscrivere e ricompilare il programma); se me ne servono solo 5 ho irrimediabilmente sprecato un po' di memoria (salvo riscrivere ecc.). Uno dei casi in cui si ricorre a variabili dinamiche è proprio questo, quando non posso determinare in partenza quanti dati di un certo tipo mi occorreranno.

Torniamo un attimo a quanto detto la volta scorsa: il Turbo Pascal pone le istruzioni eseguibili nel code segment, le variabili globali nel data segment; questi due segmenti sono contigui e posti «in basso» nella RAM; lo stack viene invece posto nella parte «alta» della RAM; in mezzo c'è lo heap. Questo comporta che tra il data segment e lo stack c'è un'area di memoria la cui ampiezza non dipende dal programma, ma dalla RAM fisicamente disponibile. Quest'area può anche non essere utilizzata, oppure posso valermene per creare nuove variabili: ogni volta che mi serve una nuova variabile riservo per questa uno spazio nello heap, per poi magari liberarlo di nuovo quando la variabile non mi serve più.

Variabili-indirizzo

Problema: si accede ad una variabile mediante il suo indirizzo, ma una variabile dinamica può nascere (e poi morire, rinascere, ecc.) in qualsiasi punto dello heap. Ecco quindi la necessità di variabili di tipo puntatore: ad ogni variabile dinamica è associato un puntatore, ovvero una variabile il cui valore, determinato al momento della creazione della variabile dinamica, non è altro che l'indirizzo di questa (sotto quest'aspetto un puntatore assomiglia molto ad un parametro-variabile, al punto che in altri linguaggio, come il C, i parametri-variabile *sono* puntatori).

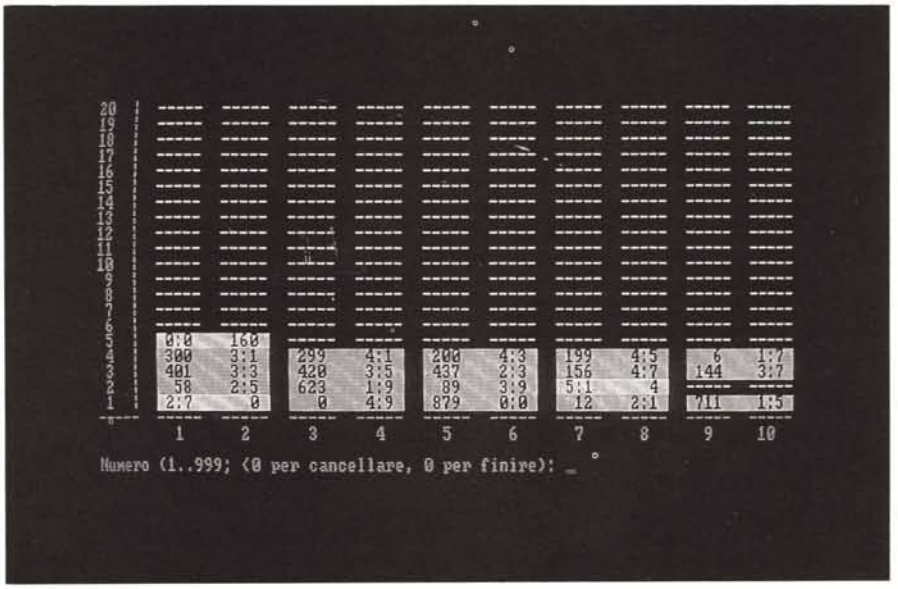
Nella figura viene mostrata una schermata del nostro programma: nella parte superiore è rappresentato l'array di interi che usiamo come simulazione dello heap, ogni singolo intero è identificato da due numeri, la riga e la colonna in cui si trova.

Per poter creare una variabile dinamica la procedura New (in ALLOC.INC, listato n. 1) deve in primo luogo sapere quali zone del mio array-heap sono libere, quali già occupate e quanto è ampia ogni zona libera; stabiliamo quindi che ogni blocco libero contenga nella sua seconda casella l'indicazione della sua dimensione e nella prima l'indirizzo — riga e colonna — del blocco libero suc-

cessivo ("0:0" se questo non esiste). Non possiamo permettere che questo tipo di informazione scompaia, neppure quando non vi sono più blocchi liberi; riserviamo quindi le prime due caselle dell'array-heap, in modo che non saranno mai «allocabili»: la prima conterrà l'indirizzo del primo blocco libero ("0:0" quando non ce ne sono più), ma la seconda uno zero; in questo modo le due caselle, pur costituendo un blocco libero, vengono viste da New e Dispose come un blocco avente «dimensione nulla», non disponibile quindi per la creazione di variabili dinamiche.

I blocchi liberi costituiscono un primo esempio di «lista»: possono essere dislocati dovunque nel nostro array ma, grazie all'indirizzo del blocco successivo (un puntatore a tutti gli effetti) nella prima casella, le procedure New e Dispose li possono visitare uno dopo l'altro come se fossero adiacenti.

La lista che vogliamo costruire avrà un funzionamento analogo; in particolare anche questa dovrà essere «ancorata» in qualche modo. In altri termini, possiamo sì creare variabili dinamiche, come tali non note al momento della compilazione del programma, ma per potervi poi accedere abbiamo bisogno di un punto di partenza, di una variabile di tipo puntatore di cui sia noto l'indirizzo già al momento della compilazione (e quindi statica) e il cui valore sia l'indirizzo di una zona dello heap contenente a sua volta l'indirizzo dell'inizio della lista. È meno complicato di quanto sembra. Procediamo con ordine.



Record.next

Il tipo di dato «record» consente di trattare come un'unica entità un insieme di più informazioni anche tra loro eterogenee. Un record i cui campi siano «Nome», «Cognome», «Prefisso» e «NumeroDiTelefono» consente di accedere in una volta sola all'insieme di informazioni che costituiscono un indirizzo telefonico; posso così avere un unico array di tali record invece che quattro array distinti, uno per campo. Comodo. Ma la vera ragion d'essere dei record è un'altra: la possibilità di dichiarare campi «puntatore a record dello stesso tipo». Quella variabile - ancora di cui dicevamo sopra (spesso chiamata «root», cioè radice) può quindi limitarsi a puntare solo al primo elemento della lista; questo elemento contiene a sua volta un puntatore (spesso chiamato «next») all'elemento successivo e così via, sino all'ultimo elemento, il cui «next» non punterà a nulla (avrà valore «nil»): è grazie a questa concatenazione di puntatori che possono scorrere una lista partendo dalla «root».

Diamo un'occhiata al nostro programma ALLOC.PAS (listato n. 2).

Dicevamo che costituisce più che altro uno strumento espositivo, in quanto usiamo un array di interi invece che lo heap vero e proprio, ma non è poi così diverso da un vero programma Pascal: accanto ad alcune righe potete trovare, tra parentesi graffe, le modifiche che bisognerebbe apportare per farlo girare usando il vero heap e le vere procedure New e Dispose (le righe a destra delle quali vi è una coppia di parentesi graffe senza nulla in mezzo andrebbero semplicemente eliminate).

La principale differenza risiede nella dichiarazione del tipo NodoPtr: in Pascal sarebbe un puntatore a (cioè l'indirizzo di) un record di tipo Nodo, noi ne facciamo un integer in quanto non sarà altro che un indice per il nostro array-heap; la procedura DumpMem nel listato 1 traduce questo indice in una coppia riga-colonna, che potreste anche interpretare come coppia segmento-offset (avevamo visto la volta scorsa che in ambiente MS-DOS ogni indirizzo relativo allo heap è sempre costituito da un segmento e un offset). Una variabile di tipo Nodo viene quindi mostrata sullo schermo come due caselle dell'array-heap: nella prima il Dato e nella seconda il Next, cioè il puntatore a (l'indirizzo di) un elemento successivo. Nel caso questo non esista, uno "0:0" rappresenta il «nil».

Notate che ho cercato di distinguere con il colore le coppie di caselle che tengono in fila i blocchi di memoria liberi da quelle che contengono le nostre variabili di tipo Nodo; ho anche variato la posizione dei puntatori (nella prima casella di un blocco libero c'è l'indirizzo del blocco successivo; nei Nodi il puntatore all'elemento successivo è nella seconda casella) sperando di rendere chiara la rappresentazione anche per chi dispone di un video monocromatico.

Il programma per prima cosa inizializza la «root»: la chiamata "New(Root)" crea una variabile di tipo Nodo nel primo posto disponibile (riga 1:colonna 3) e assegna il suo indirizzo alla variabile Root; l'istruzione successiva assegna il valore «nil» ("0:0" nel nostro caso); al campo Next della variabile di tipo Nodo «puntata» da Root. La variabile appena

creata non ha un nome: vi si accede mediante Root, con Heap[Root] in ALLOC.PAS, con "Root^" (ovvero: «ciò cui punta Root») in un normale programma Pascal.

Inserimento e cancellazione

Poi si continua inserendo e cancellando interi (il campo Dato è appunto di tipo integer). Badate che, per esigenze di spazio, il programma non è a prova di errore: se inserite interi troppo grandi (maggiori di 999 o, peggio, maggiori di 32767) guastate l'estetica delle schermate o addirittura provocate un errore; analogamente se inserite più interi di quanti ne può contenere l'array-heap. La regola è per il resto semplice: un intero positivo viene inserito nella lista, un numero negativo provoca la cancellazione dalla lista del corrispondente intero positivo. Vengono ignorati sia l'inserimento di un intero che è già in lista, che la cancellazione di uno che non c'è.

L'inserimento avviene «al posto giusto». Si parte da Root e si segue la concatenazione di puntatori Next in cerca di un campo Dato maggiore del numero immesso; la procedura Cerca usa due variabili locali di tipo NodoPtr: la prima (p1) punta ad un elemento della lista, la seconda (p2) al successivo; se il campo Dato della variabile puntata da p2 è minore del numero immesso, i due puntatori vengono fatti scorrere di un elemento: "p1 := p2" fa sì che p1 punti all'elemento cui prima puntava p2 (gli si assegna cioè l'indirizzo che era il valore di p2), "p2 := Heap[p1+1]" (ovvero: "p2 := p1^.Next") fa sì che a sua volta p2 punti all'elemento successivo a quello cui puntava prima, elemento cui ora punta p1. Quando si trova un Dato maggiore del numero immesso, questo sarà puntato da p2, il precedente da p1: la procedura Inserisci crea una nuova variabile di tipo Nodo ponendone l'indirizzo in p3, assegna al Next del Nodo puntato da p1 questo indirizzo, assegna al Dato della nuova variabile il numero e al Next l'indirizzo dell'elemento con il Dato maggiore. Se questo non esiste si usa «nil».

La cancellazione avviene in maniera analoga. La procedura Elimina si serve anch'essa di due variabili locali di tipo NodoPtr e le fa scorrere lungo la lista fino a trovare l'elemento il cui campo Dato contenga il numero che si vuole togliere dalla lista; quando lo trova, p2 conterrà l'indirizzo del Nodo cercato e p1 quello del precedente; per far sparire il Nodo puntato da p2 basta assegnare al campo Next dell'elemento puntato da p1, invece che l'indirizzo di questo

Listato 1

```
{ ALLOC.INC }

const
  MAXMEM = 200; NULL = 0;
var
  Heap: array[1..MAXMEM] of integer;
procedure InitMem;
  begin
    FillChar(Heap,MAXMEM * 2,0); Heap[1] := 3; Heap[4] := MAXMEM-2
  end;
procedure New(var np: NodoPtr);
  var
    p,q,n,k: integer;
  begin
    np := NULL;
    n := (sizeof(Nodo) + 1) div sizeof(integer); q := 1; p := Heap[q];
    while p <> NULL do begin
      if Heap[p+1] >= n then begin
        k := Heap[p+1] - n;
        if k = 0 then Heap[q] := Heap[p]
        else begin
          Heap[q] := p + n;
          Heap[p+n] := Heap[p];
          Heap[p+n+1] := k;
        end;
        np := p;
        exit
      end
      else
        q := p
    end
  end;
procedure Dispose(np: NodoPtr);
  var
    p,q,n: integer;
  begin
    n := (sizeof(Nodo) + 1) div sizeof(integer); q := 1; p := Heap[q];
    while (p <> NULL) and (p < np) do begin
      q := p; p := Heap[q]
    end;
    if (np + n = p) then begin
      n := n + Heap[p+1]; Heap[np] := Heap[p]
    end
    else
      Heap[np] := p;
    if (q + Heap[q+1] = np) then begin
      Heap[q+1] := Heap[q+1] + n; Heap[q] := Heap[np]
    end
    else begin
      Heap[q] := np; Heap[np+1] := n
    end
  end;

procedure DumpMem;
  const
    Trama: string[68] =
    '-----  -----  -----  -----  -----  -----  -----  -----  -----  -----';
  var
    i,p,q,r,c,m: integer;
  begin
    clrscr; gotoxy(1,21); write('-----  ',Trama);
    gotoxy(10,22);
    write('1      2      3      4      5      6      7      8      9      10');
    for i := 20 downto 1 do begin
      gotoxy(1,i); write(21-i:2,'  |  ')
    end;
    TextColor(Cyan);
    for i := 20 downto 1 do begin
      gotoxy(8,i); write(Trama)
    end;
    p := 1; TextColor(Black); TextBackground(Cyan);
    while p <> NULL do begin
      q := p - 1; c := ((q mod 10) + 1) * 7 + 1; r := 20 - (q div 10);
      gotoxy(c,r);
      m := Heap[p+1]; p := Heap[p];
      if p <> NULL then begin
        q := p - 1; c := (q mod 10) + 1; r := (q div 10) + 1
      end
      else begin
        c := 0; r := 0
      end;
      write(r:2,':',c,m:7,' ')
    end;
    p := Root; TextColor(Black); TextBackground(Yellow);
    while p <> NULL do begin
      q := p - 1; c := ((q mod 10) + 1) * 7 + 1; r := 20 - (q div 10);
      gotoxy(c,r);
      m := Heap[p]; p := Heap[p+1];
      if p <> NULL then begin
        q := p - 1; c := (q mod 10) + 1; r := (q div 10) + 1
      end
      else begin
        c := 0; r := 0
      end;
      write(m:4,'   ',r:2,':',c,' ')
    end;
    TextColor(Green); TextBackground(Black); gotoxy(1,24)
  end;
```

Nodo, quello del successivo (p1^.Next := p2^.Next). In questo modo un successivo scorrimento della lista «salterà» il Nodo eliminato. La chiamata della procedura Dispose serve a restituire al pool di blocchi liberi la memoria occupata dal Nodo.

Per ottenere la situazione illustrata nella schermata pubblicata all'inizio dell'articolo abbiamo introdotto i seguenti numeri: 879, 12, 711, 58, 623, 89, 555, 112, 401, 420, 437, 156, 144, 300, 299, 200, 199, 6; abbiamo quindi cancellato 555 e 112. Potete notare che la prima casella (riga 1:colonna 1) contiene l'indirizzo del primo blocco libero successivo (riga 2:colonna 7); questo contiene un indirizzo analogo (5:1) e la sua dimensione (4); il blocco che comincia in 5:1 è l'ultimo libero (0:0 nella prima casella) e consta di 160 caselle. La variabile puntata da Root (in 1:3) contiene l'indirizzo del Nodo con il Dato più piccolo (in 4:9 c'è 6) e questo, nella sua seconda casella, l'indirizzo di quello con il Dato successivo secondo un ordine crescente (in 1:7 c'è 12), e così via. Se fate girare il programma potete soprattutto notare come cambiano i puntatori ogni volta che inserite o cancellate un numero, è così vedere da vicino come la lista si mantiene ordinata «da sola».

Applicazioni

Il programma ALLOC.PAS non è in sé molto utile, eppure già si può intravedere un possibile impiego di procedure come Cerca e Elimina. Basta pensare ad un programma che abbia bisogno di tenere costantemente ordinati i suoi dati: metterli in un array e poi ordinare questo dopo ogni input sarebbe sicuramente troppo oneroso. Per illustrare la potenza di strutture come le liste concatenate occorrerebbe certo proporre esempi più concreti, cosa che ora non abbiamo voluto fare per mostrare innanzitutto quello che «sta dietro» la gestione di strutture dinamiche di dati; torneremo sull'argomento nei prossimi mesi.

Già ora si possono tuttavia anticipare alcuni temi, in primo luogo si può accennare alla flessibilità delle liste: non è necessario percorrere ogni volta una lista dall'inizio, ma si può sostituire la «root» con due puntatori (spesso chiamati «head» e «tail», cioè «testa» e «coda») contenenti l'indirizzo del primo e dell'ultimo elemento (e ottenere così una lista «FIFO»), o attribuire ad ogni Nodo, oltre a Next, anche un puntatore all'elemento precedente (liste «doppie»); oppure si può attribuire al Next dell'ultimo elemento l'indirizzo del primo e ottenere così una lista «circolare»; si può poi trasformare la lista semplice in un «albero» prevedendo più di un puntatore ad elementi successivi, e così via. Tanto per accennare a esempi concreti, il Turbo Editor Toolbox usa liste doppie e i file indice del Database Toolbox sono realizzati mediante alberi. Ne riparleremo.

Listato 2

```
{ ALLOC.PAS }

program Alloc;
{$R+}
type
  NodoPtr = integer;                    {   NodoPtr = ^Nodo;                    }
  Nodo = record
    Dato: integer; Next: NodoPtr
  end;
var
  Root: NodoPtr;
  Num : integer;
{$I ALLOC.INC}                          {                                       }
procedure Cerca(x: integer);
  var
    p1,p2: NodoPtr;
  procedure Inserisci(p: NodoPtr);
    var
      p3: NodoPtr;
    begin
      New(p3);
      Heap[p3] := x; Heap[p3+1] := p;   {       p3^.Dato := x; p3^.Next := p;   }
      Heap[p1+1] := p3                  {       p1^.Next := p3                  }
    end;
  begin
    p1 := Root; p2 := Heap[p1+1];       {     p1 := Root; p2 := p1^.Next;       }
    if p2 = NULL then Inserisci(NULL)   {     if p2 = nil then Inserisci(nil)   }
    else begin
      while (Heap[p2] < x)              {       while (p2^.Dato < x)            }
      and (Heap[p2+1]<>NULL) do begin   {       and (p2^.Next <> nil) do begin  }
        p1 := p2; p2 := Heap[p1+1]      {         p1 := p2; p2 := p1^.Next      }
      end;
      if Heap[p2] <> x then begin       {       if p2^.Dato <> x then begin     }
        if Heap[p2] < x then begin      {         if p2^.Dato < x then begin    }
          p1 := p2; p2 := NULL          {           p1 := p2; p2 := nil         }
        end;
        Inserisci(p2)
      end
    end
  end;

procedure Elimina(x: integer);
  var
    p1,p2: NodoPtr;
  begin
    p1 := Root; p2 := Heap[Root+1];     {     p1 := Root; p2 := Root^.Next;     }
    while p2 <> NULL do begin           {     while p2 <> nil do begin          }
      if Heap[p2] = x then begin        {       if p2^.Dato = x then begin      }
        Heap[p1+1] := Heap[p2+1];       {         p1^.Next := p2^.Next;         }
        Dispose(p2);
        exit
      end
      else begin
        p1 := p2; p2 := Heap[p2+1]      {         p1 := p2; p2 := p2^.Next      }
      end
    end
  end;
procedure PrintList(p: NodoPtr);
  begin
    clrscr;
    while p <> NULL do begin            {     while p <> nil do begin           }
      writeln(Heap[p]:5);               {       writeln(p^.Dato:5);             }
      p := Heap[p+1]                    {       p := p^.Next                    }
    end
  end;
begin
  InitMem;                              {                                       }
  TextColor(Green);                     {                                       }
  New(Root); Heap[Root+1] := NULL;      {   New(Root); Root^.Next := nil;       }
  DumpMem;                              {                                       }
  write('Numero (1..999; 0 per finire): '); readln(Num);
  while Num <> 0 do begin
    if Num > 0 then Cerca(Num)
    else Elimina(-Num);
    DumpMem;                            {                                       }
    write('Numero (1..999; <0 per cancellare, 0 per finire): '); readln(Num);
  end;
  PrintList(Heap[Root+1])               {   PrintList(Root^.Next)               }
end.
```