di Sergio Polini

Le insidie del nodo zero

Mancano ormai solo gli ultimi frammenti: poche routine, poche righe da aggiungere ad una procedura già vista il mese scorso, e potremo lasciarci QUED alle spalle. Queste ultime battute ci offriranno l'occasione di mettere l'accento sugli aspetti più delicati della manipolazione di liste circolari doppie: la cancellazione di nodi, la copia o lo spostamento di nodi da un punto all'altro della lista. Sono proprio questi i punti che con maggiore probabilità vi offrono l'occasione di sbagliare.
Come già detto più volte, anche se non voleste usare QUED per i fini per cui è stato scritto (un editor di linea usabile in file batch), le routine contenute in QLIST.INC possono essere agevolmente inserite in altri programmi; con gli accorgimenti che vedremo tra breve potrete ormai sentirvi padroni delle liste circolari doppie, tanto che... il mese prossimo dovremo necessariamente cominciare a trattare strutture di dati un po' diverse

La volta scorsa abbiamo cominciato a vedere l'ultimo file del sorgente di QUED, QCOMD.INC. Poiché si tratta del file più lungo, ne abbiamo visto solo una parte. Le procedure principali sono *Q_EseguiGlob*, che esegue i comandi globali, e *Q_Esegui*, che esegue tutti i comandi (quelli globali una riga per volta, essendo ripetutamente chiamata dall'altra), dopo aver chiamato *Q_Default*. Il «nocciolo» di *Q_Esegui* è rappresentato da una istruzione **case** che, secondo il comando digitato dall'utente, compie una serie di azioni: tutte molto semplici, vuoi perché è semplice il comando (come «Quit», ad esempio), vuoi perché il lavoro viene in realtà svolto da altre procedure. Alcune di queste sono contenute nello stesso file QCOMD.INC, altre in QFILE.INC o, soprattutto, in QLIST.INC. Abbiamo già visto come vengono implementati alcuni dei comandi che non richiedono routine contenute in questi file, ora vedremo tutte quelle che restano.

Prompt, Silent, Help, help

Sono comandi «di stato»: l'esecuzione di ognuno di essi si riduce alla «inversione» del valore di un flag. Se *MostraPrompt* vale FALSE, l'utente deve fare un po' d'attenzione quando usa i comandi «append», «change» o «insert»; l'immissione di nuove righe di testo deve terminare con un punto prima che si possano dare altri comandi, altrimenti quelli che sembrano comandi verranno in realtà aggiunti come nuove righe al testo in memoria. Se *Mostra Prompt* vale TRUE, invece, quando l'editor aspetta un comando lo annuncia visualizzando un prompt («>» per default, ma può essere cambiato con l'opzione −p quando si fa partire il programma). All'inizio *MostraPrompt* vale FALSE, per non «imbrattare» lo schermo se l'esecuzione viene comandata mediante un file batch, come nell'esempio proposto nel numero di luglio dello scorso anno.

MostraTotRighe governa invece la visualizzazione del numero di righe lette da o scritte su un file: normalmente il numero viene visualizzato, al fine di dare conferma del buon esito dell'operazione di I/O; se *MostraTotRighe* vale invece FALSE tutto avviene «al buio», come potrebbe essere preferibile quando si lancia QUED da un file batch. Si tratta di una logica un po' opposta a quella del prompt, ma un errore di lettura/scrittura può anche essere considerato troppo grave per rimanere nascosto per default (e poi... l'*ed* di Unix funziona così!). In ogni caso l'utente può cambiare quando vuole lo stato di quei flag con i comandi «P» («Prompt») e «S» («Silent»), o con opzioni nella riga comando.

Analoga la situazione per i messaggi d'errore: normalmente *ErroriInChiaro* vale FALSE, e quindi quando qualcosa non va viene solo mostrato un ermetico punto interrogativo. È un po' come per il prompt, nel senso che da un lato un tale ermetismo può risultare ostico solo le prime volte, dall'altro si può cambiare il comportamento del programma in ogni momento invertendo il flag con «H» («Help»). Si sa che comunque non serve chiudere la stalla dopo che sono scappati i buoi: non sarebbe molto utile chiedere messaggi in chiaro *dopo* essere rimasti interdetti davanti ad un misterioso punto interrogativo. Ecco quindi che il comando «H» per prima cosa visualizza un messaggio relativo all'ultimo errore che si è verificato, e abbiamo anche un comando «h» («help») che si limita a questo, senza modificare il flag *ErroriInChiaro*.

substitute

Se i comandi di stato sono i più semplici, il comando «s» è invece quello più ingombrante: si serve sia di una apposita procedura in QCOMD.INC che di alcune routine di QLIST.INC. Questo accade perché si tratta di un comando molto flessibile: può agire su tutto il testo su tutte le righe «marcate» per l'esecuzione come comando globale, o su una sola riga; in ogni riga può sostituire solo la prima o tutte le occorrenze della stringa indicata. Se eseguito come comando locale provoca l'emissione di

un messaggio d'errore se non trova nella riga corrente la stringa da sostituire, ma se eseguito come comando globale considera corretta anche un'esecuzione che non opera alcuna sostituzione (nel primo caso, infatti, potrei aver sbagliato riga; nel secondo potrei aver voluto solo sostituire, ad esempio, tutti i «Febbraio» con «febbraio», e perché non ci sia errore basta che non ci sia alcun febbraio maiuscolo, che ci fosse prima o no).

La ricerca e la sostituzione di stringhe sono argomenti piuttosto delicati, nel senso che si possono usare numerosi diversi algoritmi, alcuni dei quali piuttosto complicati, e che non è affatto facile decidere quale sia l'algoritmo migliore. Il Turbo Pascal offre comunque molte comode procedure di manipolazione di stringhe, con le quali si può agevolmente trovare una soluzione al problema. È quanto viene fatto in QUED; chi volesse un approfondimento può provare a cominciare con gli *Algorithms* di Robert Sedgewick (Addison-Wesley).

file-name, read, write

Si può far leggere un file a QUED indicandone il nome nella riga comando, o anche in un secondo momento con il comando «r» («read»). In ogni caso il nome del file letto viene conservato nella variabile *NomeFile*. Le righe lette verranno «appese» a quella corrente o a quella indicata subito prima del comando «r»; ciò consente di inserire il contenuto di un file nel punto desiderato dell'eventuale testo in memoria. Se non c'è alcun testo in memoria, per riga corrente si intende quella puntata da *NodoZeroPtr*. È questo uno dei vantaggi delle liste circolari con nodo «zeresimo»: l'aggiunta di nuovi nodi avviene nello stesso modo sia che si tratti di inserire nodi in una lista che già ne ha alcuni sia che si tratti di creare con quei nodi una nuova lista; proprio perché, grazie al nodo «zeresimo», una lista non è mai davvero vuota. Il prezzo di tali vantaggi, come vedremo subito, è solo la necessità di un po' di cautela.

Lo scopo della variabile *NomeFile* è quello di fornire una nome di file di default per i comandi «read» e «write»: non è necessario digitare espressamente il nome del file che si vuole leggere o scrivere, se questo coincide con quello conservato in *NomeFile*; il comando «f» («file-name») serve appunto a verificare il contenuto di questa variabile, oppure a cambiarlo se seguito da un nuovo nome (che le routine di analisi lessicale parcheggiano nella variabile *Stringa1*).

Il comando «w» («write») scrive su un file tutto il testo in memoria o anche (se preceduto dalla indicazione di una riga iniziale e di una finale) solo una sua parte. Se non viene indicato un nome di file si usa quello memorizzato in *NomeFile*.

Quando si esegue un comando «read» viene assegnato il valore TRUE alla variabile *Cambiamenti*: tutte le routine che alterano il testo in memoria fanno la stessa cosa per consentire ad altri comandi, come «quit», di verificare che il testo cambiato sia stato salvato prima di perderlo. *Cambiamenti* diventa invece FALSE se con «write» si scrive su disco tutto il testo.

Ed eccoci alla «cautela». Come vedremo subito, le routine di analisi sintattica devono accettare come numero di riga valido uno zero, che indica ovviamente la riga «zeresima». Si può aggiungere testo dopo tale riga (ad esempio con «Or»), ma altre operazioni sono assolutamente da evitare: non ha senso, in particolare, scrivere su un file una «riga» che in realtà non appartiene al testo in memoria. L'unica soluzione è quindi di effettuare i necessari controlli quando si devono eseguire operazioni che non possono agire indifferentemente sulla riga «zeresima» e su quelle «normali». Il codice per il comando «write» comincia appunto con un test di questo tipo.

append, insert, delete, change, copy, move

Con «a» («append») si aggiungono nuove righe; se viene indicata una riga (con un numero o con una stringa in essa contenuta), le nuove righe vengono inserite dopo di questa, altrimenti dopo la riga corrente. Viene utilizzata a questo scopo una procedura dichiarata in QLIST.INC, che si incarica pure di rendere «corrente» l'ultima riga aggiunta.

Quando il programma parte, se non si è letto un file, «append» è probabilmente il primo comando usato; le righe vengono in tal caso aggiunte al nostro famoso nodo zero. Un'operazione del genere è anche possibile quando abbiamo già un testo in memoria, se vogliamo inserire nuove righe all'inizio: si può usare un comando «Oa» per dire al programma di aggiungere le righe dopo la «zeresima», ovvero prima della prima.

Il comando «i» («insert») è molto simile; la differenza è che le nuove righe vengono inserite *prima* della riga indicata o di quella corrente. Per inserire nuove righe all'inizio del testo si possono quindi usare due comandi, «Oa» oppure «1i». La somiglianza tra i due comandi è comunque tale che «insert» usa le stesse routine di «append», solo passando loro come parametro *RP2^.Prev* invece di *RP2* (avevamo già visto qualche mese fa che si potevano unificare in questo modo le routine di «inserimento prima» e «inserimento dopo»: un vantaggio delle liste doppie). C'è però bisogno anche qui di un po' di cautela: se si accettasse un comando come «Oi», l'utente scoprirebbe di aver «aggiunto alla fine» invece di «inserire all'inizio» come probabilmente voleva; bisogna quindi per prima cosa controllare che *RP2* non sia uguale a *NodoZeroPtr*.

Con «d» («delete») il pericolo è ancora maggiore: non si può correre il rischio che venga cancellato proprio il nodo zero. Avevamo già visto che le routine di analisi sintattica (in QPARS.INC) si incaricano di controllare che la sublista indicata dall'utente non passi per il nodo zero (ad esempio con «34,2p»: non si possono stampare le righe dalla trentaquattresima alla seconda), ma ora bisogna anche

controllare che la sublista non *parta* dal nodo zero. Il resto è facile: si mette a posto la riga corrente secondo quanto previsto nella apposita tabella vista il mese scorso, si chiama una procedura di QLIST.INC, si assegna TRUE a *Cambiamenti*.

Analoga l'implementazione del comando «c» («change»), anche perché non si fa altro che chiamare le stesse routine utilizzate da «delete» e «append».

Anche «m» e «t» («move» e «copy to») vengono implementati in modo simile: non si può né spostare né duplicare il nodo zero, e quindi si fa in primo luogo il solito controllino prima di chiamare una procedura di QLIST.INC e di assegnare TRUE a *Cambiamenti*.

Edit, edit

Si legge un file dal disco, si apportano modifiche, si cancellano alcune righe e se ne aggiungono altre, e poi si vuole cominciare a lavorare su un altro file. Non è necessario uscire a rientrare dal programma, e nemmeno cancellare tutto con «1,$d» e leggere il nuovo file con «r». I comandi «E» ed «e» pensano a tutto, il primo ignorando gli eventuali cambiamenti apportati al testo già in memoria, il secondo rifiutandosi di caricare un nuovo testo se il precedente non è stato salvato.

In apparenza è tutto molto facile, in quanto non si tratta di altro che di un «delete» seguito da un «read». Ma le apparenze, si sa, ingannano. Avevamo visto la volta scorsa il ciclo principale del programma, e notato che il puntatore alla riga corrente (*NodoCorrPtr*) viene salvato in *CopiaNCPtr* prima di interpretare ed eseguire un comando. Questo perché molti comandi cambiano la riga corrente, ma potrebbero cambiarla anche prima di portare a buon fine il loro compito; se si verifica un errore possiamo sempre tornare «dove eravamo» proprio grazie a *CopiaNCPtr*. Il problema nasce proprio dal fatto che i comandi «edit» e «Edit» sono la combinazione di due altri comandi: potrebbe andar bene il «delete», ma potrebbe fallire il «read» (ad esempio perché si è indicato dopo «e» o «E» il nome di un file che non esiste). In questo caso tornare alla riga puntata da *CopiaNCPtr* sarebbe catastrofico, in quanto si tratterebbe di una

```
(* QCOMD.INC *)

procedure Q_NomeFile(var S: integer);
begin
  S := OK;
  if Stringa1 <> '' then NomeFile := Stringa1
  else if NomeFile = '' then S := ERRNOMEF
end;

procedure Q_Append(RP: NPtr; var Stato: integer);
var
  InpStr: AnyStr;  Fine: boolean;
begin
  NodoCorrPtr := RP; Stato := OK; Fine := FALSE;
  while (not Fine) and (Stato = OK) do begin
    if eof(StdIn) then Fine := TRUE
    else readln(StdIn,InpStr);
    if Stato = OK then
      if InpStr = FINEAPP then Fine := TRUE
      else begin
        Q_AddRiga(InpStr, NodoCorrPtr, Stato);
        if Stato = OK then Cambiamenti := TRUE
      end
  end
end;

procedure Q_nPrint(RP1, RP2: NPtr; var Stato: integer);
(* V. numero di gennaio *)

procedure Q_Print(RP1, RP2: NPtr; var Stato: integer);
(* V. numero di gennaio *)

procedure Q_Subst(RP1, RP2: NPtr; gFlag: boolean;
            var nFlag, pFlag: boolean; Glob: boolean; var Stato: integer);
var
  p: integer;  s: AnyStr;  Trovata, Sostituita: boolean;
begin
  s := RP1^.Txt^; Stato := OK; Sostituita := FALSE; p := 1;
  while (RP1 <> RP2^.Next) and (Stato = OK) do begin
    Trovata := FALSE;
    while (p <= length(s)) and (not Trovata) do
      if copy(s,p,length(Stringa1)) = Stringa1 then Trovata := TRUE
      else p := p + 1;
    if Trovata then begin
      if length(s) - length(Stringa1) + length(Stringa2) <= 255 then begin
        freemem(RP1^.Txt, length(s)+1);
        delete(s, p, length(Stringa1)); insert(Stringa2, s, p);
        Q_CreaRiga(s, RP1, Stato);
        if Stato = OK then begin
          Sostituita := TRUE; Cambiamenti := TRUE; NodoCorrPtr := RP1
        end;
        p := p + length(Stringa2)
      end
      else Stato := ERRLUNST
    end;
    if (Trovata and not gFlag) or (not Trovata) then begin
      RP1 := RP1^.Next; s := RP1^.Txt^; p := 1
    end
  end;
  if Glob and (not Sostituita) then begin nFlag := FALSE; pFlag := FALSE end;
  if (Stato = OK) and (not Sostituita) and (not Glob) then Stato := ERRCERCA
end;

procedure Q_Default(Cmd: char; var RP1, RP2: NPtr);
(* V. numero di gennaio *)

procedure Q_Esegui(Cmd: char; RP1, RP2, RP3: NPtr;
                   gFlag, nFlag, pFlag, Glob: boolean; var Stato: integer);
var
  p: NPtr; n: integer;
begin
  Stato := OK;
  if not Glob then Q_Default(Cmd, RP1, RP2);
  case Cmd of
    ERRMSG: if UltErr <> 0 then begin                              (* help *)
              writeln(StdErr, Msg[UltErr]); flush(StdErr)
            end;
    HELP  : begin                                                  (* Help *)
              if UltErr <> 0 then begin
                writeln(StdErr, Msg[UltErr]); flush(StdErr)
              end;
              ErroriInChiaro := not ErroriInChiaro
            end;
    PROMPT: MostraPrompt := not MostraPrompt;                      (* Prompt *)
    QUITIF: if not Cambiamenti then Stato := FINEDATI               (* quit *)
            else Stato := ERRNOSALV;
```

```
  QUIT   : Stato := FINEDATI;                                    (* Quit *)
  SILENT: MostraTotRighe := not MostraTotRighe;                  (* Silent *)
  EDIT, EDITIF:                                                  (* Edit, edit *)
          if (Cmd = EDITIF) and Cambiamenti then Stato := ERRNOSALV
          else begin
            Q_NomeFile(Stato);
            if Stato = OK then begin
              if NodoZeroPtr^.Next <> NodoZeroPtr then
                DelSubLista(NodoZeroPtr^.Next, NodoZeroPtr^.Prev);
              NodoCorrPtr := NodoZeroPtr;
              CopiaNCPtr := NodoZeroPtr;
              Q_ReadFile(NodoCorrPtr, Stato)
            end;
            if Stato = OK then Cambiamenti := FALSE
          end;
  NOMFIL: if Stringa1 = '' then                                  (* file-name *)
            if NomeFile <> '' then begin
              writeln(StdOut,NomeFile); flush(StdOut)
            end
            else Stato := ERRNOMEF
          else NomeFile := Stringa1;
  DEL    : if RP1 <> NodoZeroPtr then begin                      (* delete *)
            NodoCorrPtr := RP2^.Next;
            if NodoCorrPtr = NodoZeroPtr then NodoCorrPtr := RP1^.Prev;
            DelSubLista(RP1, RP2);
            Cambiamenti := TRUE
          end
          else Stato := ERRSUBLST;
  NUMERA: begin                                                  (* number *)
            nFlag := FALSE; pFlag := FALSE; Q_nPrint(RP1, RP2, Stato)
          end;
  FINECMD,PRINT : begin                                          (* print *)
            nFlag := FALSE; pFlag := FALSE; Q_Print(RP1, RP2, Stato)
          end;
  SUBST : Q_Subst(RP1, RP2, gFlag, nFlag, pFlag, Glob, Stato);   (* subst *)
  APPEND: Q_Append(RP2, Stato);                                  (* append *)
  INS    : if RP2 <> NodoZeroPtr then                            (* insert *)
             Q_Append(RP2^.Prev, Stato)
          else Stato := ERRNUM;
  CHANGE: if RP1 <> NodoZeroPtr then begin                       (* change *)
            NodoCorrPtr := RP1^.Prev;
            DelSubLista(RP1, RP2);
            Q_Append(NodoCorrPtr, Stato)
          end
          else Stato := ERRSUBLST;
  NUMLIN: begin                                                  (* '=' *)
            p := NodoZeroPtr^.Next; n := 0;
            while p <> RP2^.Next do begin
              n := n + 1; p := p^.Next
            end;
            writeln(StdOut,n); flush(StdOut)
          end;
  MOVETO: if RP1 = NodoZeroPtr then Stato := ERRSUBLST           (* move *)
          else begin
            MuoviSubLista(RP1, RP2, RP3, Stato); NodoCorrPtr := RP2;
            if Stato = OK then Cambiamenti := TRUE
          end;
  COPYTO: if RP1 = NodoZeroPtr then Stato := ERRSUBLST           (* copy *)
          else begin
            CopiaSubLista(RP1, RP2, RP3, Stato); NodoCorrPtr := RP2;
            if Stato = OK then Cambiamenti := TRUE
          end;
  READF : begin                                                  (* read *)
            Q_NomeFile(Stato);
            if Stato = OK then Q_ReadFile(RP2, Stato);
            if Stato = OK then Cambiamenti := TRUE
          end;
  WRITEF: if RP1 = NodoZeroPtr then Stato := ERRSUBLST           (* write *)
          else begin
            Q_NomeFile(Stato);
            if Stato = OK then Q_WriteFile(RP1, RP2, Stato);
            if (Stato = OK) and (RP1 = NodoZeroPtr^.Next)
              and (RP2 = NodoZeroPtr^.Prev) then Cambiamenti := FALSE
          end
end;
  if Stato = OK then
    if nFlag then Q_nPrint(NodoCorrPtr, NodoCorrPtr, Stato)
    else if pFlag then Q_Print(NodoCorrPtr, NodoCorrPtr, Stato)
end;

procedure Q_EseguiGlob(ch: char; RP1, RP2: NPtr;
                       gFlag, nFlag, pFlag: boolean; var Stato: integer);
(* V. numero di gennaio *)
```

Il file QCOMD.INC, contenente le routine per l'esecuzione dei comandi di QUED. È riportata solo l'intestazione delle procedure illustrate il mese scorso, ma Q_Esegui, pubblicata in versione «ridotta» a gennaio, viene riproposta per intero.

riga cancellata. È necessario quindi assegnare sia a *NodoCorrPtr* che a *CopiaNCPtr* il valore di *NodoZeroPtr* subito prima del «read».

Conclusioni

E così abbiamo finito. Manca in realtà un ultimo comando, quello dato con un segno di uguale (ritorna il numero di riga della riga indicata, oppure, per default, dell'ultima riga: utile in tal caso per vedere quanto è lungo il testo in memoria), ma è talmente semplice!

Piuttosto, prima di lasciare definitivamente QUED, potremmo tracciare un breve biancio.

Abbiamo discusso di metodi di programmazione, dei criteri per la scelta delle strutture di dati più adatte; abbiamo distinto tra programmi dominati dalla struttura dell'output e programmi dominati dalla struttura dell'input; abbiamo visto cosa sono e come si gestiscono liste sequenziali e concatenate, lineari e circolari, semplici e doppie; abbiamo anche visto (sarebbe meglio però dire «intravisto»...) tecniche di analisi lessicale, sintattica e semantica dell'input.

Le routine illustrate questo mese vi hanno mostrato quanto possa essere semplice creare, cancellare, muovere, copiare i nodi di una lista, ed anche combinare insieme alcune di queste azioni, servendosi delle procedure contenute in QLIST.INC, facilmente portabili ad altri programmi. Vi hanno messo in guardia dai più probabili errori, chiarendo però anche che, in fondo, basta un po' d'attenzione.

A partire dal mese prossimo potremo vedere che liste e puntatori rappresentano strumenti preziosi anche in programmi completamente diversi da QUED, tanto più che basta cambiare il «tipo» o il numero dei campi-puntatore di un nodo per risolvere agevolmente problemi altrimenti ben ardui. Avremo anche modo di approfondire il tema dell'analisi dell'input, in QUED solo abbozzato: parleremo della lettura di un particolare tipo di file ma anche della implementazione di un colloquio meno spartano tra programma e utente. Il tutto sfruttando le potenzialità delle nuove versioni del Turbo Pascal.

Appuntamento a marzo!