di Sergio Polini

Qualche considerazione per cominciare

Come promesso, ecco la prima di una serie di chiacchierate sul Turbo Pascal, destinate ad aiutarvi ad acquisire sempre maggiore familiarità con questo compilatore. Non sarà un vero e proprio corso, ma piuttosto un insieme di spunti, con i quali cercheremo ogni volta di mettere in evidenza gli aspetti tipici del linguaggio, di approfondire temi solo accennati nel manuale, di proporre esempi concreti e «usabili» di quello che si può fare con il Turbo Pascal. Cercheremo anche di introdurre gradualmente i meno esperti alle sofisticate tecniche di programmazione usate nei toolbox e di svelare qualche «trucco del mestiere».
Come prima cosa, dopo qualche considerazione di carattere generale (di cui proprio non posso fare a meno), vedremo da vicino alcune delle direttive di compilazione e una piccola modifica al compilatore.

Buone abitudini

Ogni programma in Pascal comincia con una lunga serie di «dichiarazioni»: bisogna dichiarare tutte le costanti e variabili globali, tutte le funzioni e procedure che verranno usate nel corpo principale del programma. Non si può cominciare subito a scrivere una istruzione dopo l'altra, ma è necessario cercare prima di farsi un'idea non troppo vaga di quello che il programma dovrà fare e di come dovrà farlo.

Questa apparente rigidità risulta all'inizio ostica a quanti sono abituati a scrivere «di getto», ma aiuta ben presto ad assumere buone abitudini. Soprattutto se si ha in animo di realizzare programmi di una qualche complessità, e se si vuole costruire man mano un insieme di routine che possano essere usate più volte senza dover ogni volta ricominciare da zero.

Cominciamo dalla dichiarazione di costanti e variabili. Il Pascal consente una notevole flessibilità per quanto riguarda le strutture di dati che possono essere definite e manipolate e, come avremo modo di verificare, la prima mossa deve essere sempre la scelta di quelle più idonee al proprio scopo. Una scelta sbagliata può condizionare pesantemente tutta la fase di sviluppo di un progetto. Vedremo invece che una volta messo a punto questo primo aspetto non sarà difficile decidere anche quali algoritmi usare, e anzi rimandare senza tema la loro messa a punto (basterà prevedere una funzione o procedura che si faccia carico dei dettagli) e concentrarsi sul flusso principale del programma.

Bisogna sì dichiarare tutte le funzioni e procedure prima di usarle, ma non è necessario (né consigliabile) mettersi subito a scriverle riga per riga. In realtà questa dovrebbe essere proprio l'ultima fase. I Grandi Maestri (Dijkstra, Knuth, Wirth, ecc.) consigliano infatti di procedere più o meno così:

a) farsi un'idea generale di quello che il programma dovrà fare e delle strutture di dati più adatte;
b) individuare le singoli azioni che il programma deve compiere, prevedendo, ogni qualvolta appaia conveniente, una funzione o procedura che si faccia carico dei dettagli;
c) passare poi allo sviluppo delle singole funzioni e procedure con lo stesso metodo, e quindi scomponendo in altre funzioni e procedure quelle più complesse;
d) tracciato così uno schema analitico dell'insieme, si può finalmente passare alle singole istruzioni, cominciando dalle funzioni e procedure più «interne» per finire poi con il corpo principale del programma; si potrà anche a questo punto tornare sul lavoro di analisi fatto all'inizio, ora che si ha un'idea molto più precisa sul tutto, per correggere o migliorare là dove valga la pena di farlo.

Proviamo ad accennare ad un esempio: un programma che si prenda 10 interi, li metta in ordine crescente e stampi poi il risultato della sua fatica. Decidiamo di usare un array e scriviamo:

```
program SortInt;
const
  NUM = 10;
var
  a: array[1..NUM] of integer;
```

Visto che abbiamo scelto un esempio molto semplice, possiamo anche permetterci di scrivere subito il «main body» del programma e la dichiarazione delle procedure che verranno utilizzate, continuando con:

```
procedure Immissione;
  begin end;
procedure Ordinamento;
  begin end;
procedure Stampa;
  begin end;
begin
  Immissione;
  Ordinamento;
  Stampa
end.
```

E ora non rimangono che i «dettagli».

Fidarsi è bene, ma non fidarsi è meglio

Sembra facile scrivere una routine per l'input di 10 interi; magari così:

```
procedure Immissione;
var
 i: integer;
begin
  for i := 1 to NUM do begin
    Write(i:3,'^ numero: ');
    Readln(a[i])
  end
end;
```

Il guaio è che tutto funzionerà bene solo se ogni volta che il programma che lo chiederà digiteremo niente altro che un numero senza decimali compreso tra –32768 e 32767. Se gli proponiamo, ad esempio, un 80000, il programma si bloccherà sparando un messaggio d'errore.

Il controllo dell'input è un problema classico, per il quale sono possibili numerose soluzioni. Vi segnalo in particolare la procedura GetLine del MicroCalc (il mini-spreadsheet distribuito insieme al Turbo Pascal): adotta una tecnica sicuramente interessante, su cui magari torneremo un'altra volta. Ora invece useremo la direttiva «I» del compilatore: normalmente è attiva per default e ogni errore di I/O causa appunto l'arresto del programma; se disattivata, con {$Io}, un errore provoca solo la sospensione di ogni operazione di input o output fino a che non venga chiamata una funzione predefinita IoResult, che ritorna il codice di errore. IoResult ritorna invece 0 se tutto è andato bene.

Il manuale (pag. 116) propone l'esempio di una procedura che cerca di aprire un file e mantiene il controllo della situazione se il file non viene trovato, ma la direttiva «I» può tornare utile anche in altre situazioni. Potremmo infatti riscrivere la nostra procedura così:

```
type
  Str80 = string[80];

function GetInt(Msg: Str80): integer;
var
  n: integer;
begin
  {$I-}
  repeat
    Write(Msg);
    Readln(n)
  until IOResult = 0;
  {$I+}
  GetInt := n
end;

procedure Immissione;
var
  i: integer;
  s: Str80;
begin
  for i := 1 to NUM do begin
    Str(i:3,s);
    a[i] := GetInt(s + '^ numero: ')
  end
end;
```

e risolvere il nostro problema (la soluzione in realtà non è perfetta. Perché?).

Ai confini degli array

Poi ci serve una procedura Ordinamento. Proviamo con questa, tratta dal classico *Algorithms + Data Structures = Programs* di Wirth (tradotto in italiano dalla Tecniche Nuove):

```
procedure Ordinamento;
  var
    i,j,l,r,m,x: integer;
  begin
    for i := 2 to NUM do begin
      x := a[i];
      l := 1;
      r := i - 1;
      while l <= r do begin
        m := (l+r) div 2;
        if x < a[m] then
          r := m - 1
        else
          l := m + 1
      end;
      for j := i downto l do
        a[j+1] := a[j];
      a[l] := x
    end
  end;
```

Non è certo l'algoritmo migliore (Wirth ne propone molti più efficienti), ma... fa al caso nostro: ho infatti inserito un piccolo errore, di quelli tanto piccoli che sono sempre in agguato.

Vi scrivete dunque una qualsiasi procedura Stampa e provate a far girare il vostro programma; date in input i numeri da 9 a 0 e ottenete una bella fila di nove 8 seguiti da un 9!

Dove sarà mai l'errore? Se non avete usato la direttiva «R» siete un po' nei guai: può essere sbagliato l'algoritmo, potete aver scritto da qualche parte «i» invece di «j» o «l» (elle) invece di «1» (uno), ecc.

In realtà è sbagliato l'ultimo ciclo for: «for j: = i ecc.» invece di «for j: = i – 1 ecc.». Questo fa sì che, quando i vale NUM, si assegna a [NUM] ad un elemento a [NUM+1] dell'array, elemento che ovviamente non esiste.

Se mettiamo un {$R+} all'inizio del programma l'esecuzione si interromperà, comparirà il messaggio «Index out of range», e il cursore si posizionerà subito dopo «a[j+1]». Facile a questo punto capire cosa è successo.

Attenzione: in questo caso avevamo potuto constatare che qualcosa non andava fin dalla prima esecuzione del programma, ma errori di questo genere hanno spesso conseguenze molto più subdole. Provate a modificare il programma come segue (senza la direttiva «R»):

1) dichiarate una variabile Spia di tipo integer subito dopo l'array dei 10 interi; la dichiarazione delle variabili sarà quindi:

```
var
  a: array[1..NUM] of integer;
  Spia: integer;
```

2) assegnate a Spia un valore (ad es. 9999) diverso da quelli che poi metterete nell'array;

3) fatevi stampare dal programma il valore di Spia dopo l'ordinamento; il corpo principale del programma sarà quindi simile a:

```
begin
  Spia := 9999;
  Immissione;
  Ordinamento;
  Stampa;
  Write('Spia = ',Spia)
end.
```

3) eseguite il programma dando i soliti numeri da 9 a 0.

Vedrete che non solo otterrete la solita serie di nove 8 seguiti da un 9, ma anche Spia avrà assunto il valore di 8! Succede infatti che l'assegnazione ad un elemento inesistente di un array si traduce nella modifica di una qualche locazione di memoria, e potrebbe anche succedere che l'array viene manipolato correttamente dal programma, ma quella modifica causa altrove comportamenti erratici tali da mettere a dura prova le vostre capacità di debugging.

Morale: il manuale (pagg. 65 e 315) consiglia di attivare la direttiva «R» durante la fase di sviluppo di un programma e di eliminare i vari {$R+} (che rallentano un po' l'esecuzione) solo alla fine, quando si è sicuri che tutto funzio-

na a dovere. È un consiglio da non trascurare.

Non tutti lo sanno

La funzione booleana KeyPressed ritorna «true» (cioè «vero») se è stato premuto un tasto, «false» in caso contrario; è quindi molto utile quando si desidera sospendere l'esecuzione del programma per farla riprendere con la pressione di un tasto, oppure interrompere un ciclo anche prima del suo naturale completamento.

Si può scrivere qualcosa come:

```
Write('Premi un tasto.');
repeat until KeyPressed;

oppure:

repeat
  i := 1 + 1
until KeyPressed or (i = 100);
```

La direttiva «C» governa l'interpretazione di due caratteri di controllo: se attiva (come per default), Ctrl-S sospende l'output su video fino a che non si prema un tasto e Ctrl-C interrompe l'esecuzione di un programma non appena viene chiamata una procedura di I/O; se disattivata, con {$C–}, i due caratteri non hanno alcun effetto.

Il manuale chiarisce che la direttiva «C» vale per tutto un programma: non può cioè essere disattivata e attivata solo per porzioni di codice come invece si può fare con le direttive «I» o «R». Quello che il manuale non spiega è che se non si pone un {$C–} all'inizio del programma la chiamata della funzione KeyPressed può non funzionare. Ecco un esempio:

```
program Ciclo;
var
  i: integer;
begin
  i := 1;
  repeat
    { Writeln(i); }
    i := i + 1
  until KeyPressed or (i = 10000);
  Writeln('i = ',i)
end.
```

Se digitate il programma così com'è, vedrete che potrete sia interromperne l'esecuzione con un Ctrl-C, sia uscire dal ciclo premendo un qualsiasi altro tasto (senza per questo arrestare il programma, che proseguirà quindi mostrandovi il valore cui è arrivata i). Se però togliete le parentesi graffe funzionerà solo il Ctrl-C.

Succede questo: se la direttiva «C» è attiva, ogni volta che viene chiamata una procedura Read o Write una routine interna va a controllare se è stato premuto un Ctrl-C o un Ctrl-S, e così facendo «consuma» i caratteri digitati, vanificando la chiamata della funzione KeyPressed. Per potersi servire di questa occorre quindi inserire nel programma un {$C–}.

La soluzione sembra facile e immediata, ma comporta un grosso inconveniente: la possibilità di arrestare un programma con Ctrl-C è estremamente comoda durante tutta la fase di sviluppo, quando può capitare di commettere errori magari banali ma fastidiosi. Esempio:

```
program Logaritmo;
{$C-}
var
  x: real;
begin
  x := 0.0;
  repeat
    x := x + 3.0;
    Writeln(ln(x))
  until x = 100.0
end.
```

È chiaro che non essendo 100 un multiplo di 3 la condizione di uscita dal loop non potrà mai essere verificata, e quindi il programma continuerà a stamparvi il logaritmo di tutti i multipli di 3 fino a un qualche numero di 11 cifre prima di bloccarsi per l'overflow. O siete disposti ad aspettare che il vostro PC si calcoli qualche miliardo di logaritmi, o siete rassegnati a resettare ogni volta che capita qualcosa del genere, oppure dovete trovare il modo di non rinunciare al Ctrl-C.

Il modo c'è. Esiste infatti una variabile booleana predefinita non documentata CBREAK che consente di attivare/disattivare la direttiva «C» localmente: assegnandole il valore «false» si ottiene, ai fini della «usabilità» della funzione KeyPressed, lo stesso effetto che si ha con {$C–}, e viceversa. Possiamo quindi scrivere:

```
program Logaritmi;
var
  x,y: real;
begin
  x := 0.0;
  CBREAK := false;
  repeat
    x := x + 3.0;
    writeln(ln(x):20);
  until KeyPressed;
  CBREAK := true;
  y := 0.0
  repeat
    y := y + 3.0;
    writeln(ln(y):40);
  until y = 100.0
end.
```

Il primo ciclo potrà essere interrotto con la pressione di un qualsiasi tasto, il secondo con un Ctrl-C (CBREAK è stata «scoperta» da un membro del Turbo User Group, David Barker, che ne ha dato notizia sul numero 9 di *TUG Lines*, la rivista del Group).

Domande inutili

Appena parte il Turbo Pascal vi chiede subito se volete caricare il file con i messaggi d'errore (TURBO.MSG). È talmente comodo avere i messaggi in chiaro che si risponde sempre di sì, e alla lunga quella domanda diventa noiosa (il Turbo Pascal è nato quando 64K sembravano tanti, e risparmiarne uno e mezzo poteva avere un senso; ma oggi che se non hai almeno 256K non sei nessuno ...).

Seguite le istruzioni del riquadro (valide per le versioni 3.01 sotto PC-DOS), digitando solo i caratteri sottolineati, e non dovrete più preoccuparvene.

MC

Come ottenere il caricamento automatico del file TURBO.MSG

```
A>DEBUG TURBO.COM <return>
-E 2F5E <return>
xxxx:2F5E  E8.0C <spazio>  20.FF <spazio>
xxxx:2F60  DB.EB <spazio>  0D.1E <return>
-W <return>
Writing xxxx bytes
-Q <return>

A>DEBUG TURBOBCD.COM <return>
-E 2E7F <return>
xxxx:2E7F  E8.0C <spazio>
xxxx:2E80  FF. <spazio>  DB.EB <spazio>  0D.1E <return>
-W <return>
Writing xxxx bytes
-Q <return>

A>DEBUG TURBO-87.COM <return>
-E 2A27 <return>
xxxx:2A27  E8.0C <spazio>
xxxx:2A28  57.FF <spazio>  E0.EB <spazio>  0D.1E <return>
-W <return>
Writing xxxx bytes
-Q <return>
```

Le modifiche provengono direttamente dalla Borland, e quindi sono «sicure». Se tuttavia non avete familiarità con il DEBUG è vivamente consigliabile operare su COPIE dei file, per non rischiare di pasticciare gli originali.