Private e protected

di Sergio Polini (MC1166 su MC-Link)

Il Turbo Vision e il Turbo Pascal per Windows incalzano. Grazie a MC-Link, ho potuto constatare che avete già cominciato ad esplorare non solo le possibilità del Turbo Vision, ma anche le modalità di conversione da DOS a Windows di programmi realizzati secondo le specifiche SAA-CUA (ne avevamo parlato a febbraio, in occasione della prova del Turbo Pascal 6.0). Ho quindi deciso di accelerare la discussione delle tecniche di programmazione di una gerarchia di classi, in modo da poter passare quanto prima ad applicazioni «vere». In questo appuntamento e nel successivo vi proporrò tutto quanto rimane del codice della nostra piccola gerarchia, dando più spazio del solito ai listati. Come potrete verificare, con ciò non rinunceremo a discutere né di importanti questioni sintattiche né dei benefici della OOP

Come già sapete, la keyword **private** deriva dal C++. Venne introdotta da Stroustrup nel 1986; in origine vi era solo una keyword **public**, in quanto tutti i componenti della definizione di una classe erano considerati privati per default, esclusi appunto quelli che seguivano **public**. La situazione era analoga a quella che troviamo ora nel Turbo Pascal, nel senso che gli elementi di una classe potevano essere solo o «pubblici» o «privati». Ciò comportava alcuni inconvenienti, efficacemente illustrati nelle *Release Notes* della versione 1.1 del C++ AT&T: a volte si era costretti ad usare variabili d'istanza o metodi pubblici, ma con a lato un commento del tipo «Non usare a meno che non sia assolutamente necessario!».

Vediamo perché. L'utente di una classe (il programmatore che dichiara oggetti appartenenti a quella classe) ha bisogno di una interfaccia chiara e completa, ma è meglio non acceda direttamente alla sua implementazione; in questo modo, infatti, non solo si mette al riparo da insidiosi bug, ma può anche evitare di dover modificare il suo programma nel caso che l'implementazione venga cambiata. È ben diverso il caso di chi voglia derivare un'altra classe da quella data: qui è praticamente necessario accedere all'implementazione, a meno di non rassegnarsi ad inefficienze ingiustificabili. Nel 1986, quindi, Stroustrup aggiunse al C++ le keyword **private** e **protected**: con quest'ultima si denotano quegli aspetti dell'implementazione che devono rimanere inaccessibili nell'*uso* di una classe, ma disponibili per la *derivazione* da questa di una nuova classe.

Chiacchierando con Anders e con Zack

Ho avuto modo di incontrare per la prima volta Anders Hejlsberg (il padre del Turbo Pascal) a Milano il 27 marzo scorso. Come vi ha raccontato a maggio Corrado Giustozzi, lui ed io abbiamo potuto discorrere con Anders e con Gene Wang in occasione della giornata organizzata dalla Borland Italia per la presentazione dei nuovi linguaggi, durante la pausa per il pranzo. A fine giornata, in piena smobilitazione, Tommaso Masi mi chiede se avevo parlato ad Anders della questione «protected». Me ne ero dimenticato! Salto quindi sul palco per bloccare Anders: gli ricordo l'evoluzione del C++, gli accenno dei

```
unit FixdColl;

interface

uses Base, IndxColl;

type
  PFixedSizeCollection = ^TFixedSizeCollection;
  TFixedSizeCollection = object(TIndexedCollection)
    constructor Init(Size: word);
    function Name: string; virtual;
  end;

implementation

constructor TFixedSizeCollection.Init(Size: word);
begin
  TIndexedCollection.Init(Size, 0);
end;

function TFixedSizeCollection.Name: string;
begin
  Name := 'FixedSizeCollection';
end;

end.
```

Figura 1
La unit FIXDCOLL, che definisce e implementa le collezioni indicizzate non espandibili.

Per un errore tipografico sul numero scorso di MCmicrocomputer (n. 108, giugno 1991) abbiamo attribuito un titolo errato a questa rubrica. Il titolo corretto è ***«Le collezioni indicizzate»****. Ce ne scusiamo con i lettori.*

```
unit OrdColl;
(*$X+*)
interface

uses Base, Collect, IndxColl;

type
  POrderedCollection = ^TOrderedCollection;
  TOrderedCollection = object(TIndexedCollection)
    constructor Init(Size, Delta: word);
    destructor Done; virtual;
    function Name: string; virtual;
    function Add(var o: TObject): PObject; virtual;
    function Remove(var o: TObject): PObject; virtual;
    function AddAt(var o: TObject; i: word): PObject;
virtual;
    function RemoveAt(i: word): PObject; virtual;
    function First: PObject;
    function Last: PObject;
    function GetCount: word;
  private
    LastIndex: word;
  end;

implementation

constructor TOrderedCollection.Init(Size, Delta: word);
begin
  TIndexedCollection.Init(Size, Delta);
  LastIndex := 0;
end;

destructor TOrderedCollection.Done;
begin
  TIndexedCollection.Done;
end;

function TOrderedCollection.Name: string;
begin
  Name := 'OrderedCollection';
end;

function TOrderedCollection.Add(var o: TObject): PObject;
begin
  Inc(LastIndex);
  if LastIndex > GetSize then Grow;
  TIndexedCollection.AddAt(o, LastIndex);
  Add := @o;
end;

function TOrderedCollection.Remove(var o: TObject):
PObject;
begin
  if not o.IsEqual(Last^) then RunError(215);
  Remove := TIndexedCollection.Remove(o);
  while (LastIndex > 0) and (At(LastIndex) =
PObject(Null)) do
    Dec(LastIndex);
end;

function TOrderedCollection.AddAt(var o: TObject; i:
word): PObject;
begin
  RunError(216);
end;

function TOrderedCollection.RemoveAt(i: word): PObject;
begin
  RunError(216);
end;

function TOrderedCollection.First: PObject;
begin
  First := At(1);
end;

function TOrderedCollection.Last: PObject;
begin
  Last := At(LastIndex);
end;

function TOrderedCollection.GetCount: word;
begin
  GetCount := LastIndex;
end;

end.
```

Figura 2 - La unit ORDCOLL, per collezioni in cui venga sempre mantenuto l'ordine in cui i vari elementi sono stati aggiunti (liste LIFO).

```
unit SortColl;
(*$X+*)
interface

uses Base, Magnitud, IndxColl, OrdColl;

type
  PSortedCollection = ^TSortedCollection;
  TSortedCollection = object(TOrderedCollection)
    function Name: string; virtual;
    function Add(var o: TObject): PObject; virtual;
    function Remove(var o: TObject): PObject; virtual;
  end;

implementation

function TSortedCollection.Name: string;
begin
  Name := 'SortedCollection';
end;

function TSortedCollection.Add(var o: TObject): PObject;
var
  i: word;
  p: PObject;
begin
  Add := TOrderedCollection.Add(o);
  i := GetCount;
  while (i>1)
  and (PMagnitude(At(i))^.IsLessThan(PMagnitude(At(i-
1))^)) do begin
    p := TIndexedCollection.AddAt(o, i-1);
    TIndexedCollection.AddAt(p^, i);
    Dec(i);
  end;
end;

function TSortedCollection.Remove(var o: TObject):
PObject;
var
  i, Count: word;
  p: PObject;
begin
  i := 1;
  Count := GetCount;
  while (i <= Count) and (not At(i)^.IsEqual(o)) do
Inc(i);
  if i <= Count then begin
    while i < Count do begin
      p := TIndexedCollection.AddAt(o, i+1);
      TIndexedCollection.AddAt(p^, i);
      Inc(i);
    end;
    Remove := TOrderedCollection.Remove(o);
    Exit;
  end;
  Remove := Null;
end;

end.
```

Figura 3 - La unit SORTCOLL, per collezioni contenenti oggetti derivati da TMagnitude e ordinati secondo il metodo IsLessThan.

problemi che avevo incontrato nel realizzare la gerarchia di classi che vi sto propinando, gli chiedo perché ha percorso solo «metà del cammino», aggiungendo al Pascal **private** e non anche **protected**. Con l'impermeabile già in mano, Anders mi risponde cortesemente ma velocemente che il C++ è un esempio che «more is worse» («più è peggio»), ma che in effetti certi problemi sono reali e l'evoluzione del Turbo Pascal non è terminata. Vedremo magari in una prossima versione.

Il 2 maggio Corrado ed io, dopo aver partecipato alla *Borland Languages Conference* di San Francisco, abbiamo potuto visitare la sede della Borland a Scotts Valley. Durante il pranzo mi sono trovato accanto a Zack Urlocker (product manager del Turbo Pascal) e ne ho approfittato per riprendere con più calma il discorso. Il Turbo Pascal adotta la **unit** come «unità di protezione»: ciò che viene dichiarato **private** è infatti liberamente accessibile dalla sezione **implementation** della stessa **unit** ma inaccessibile da altre **unit** o dal **program**. Sembrerebbe quasi che la keyword **private** sia in realtà una sorta di **protected**, nel senso che, mentre è ristretto l'*uso* delle sezioni private di una classe, non vi sono difficoltà nella *derivazione* di altre classi, *purché la derivazione avvenga nella stessa unit*. Questa è però una vera e propria limitazione: uno degli aspetti più interessanti della OOP è proprio dato dalla possibilità di derivare nuove classi da altre classi *già compilate*, di cui cioè *sia disponibile la sola interfaccia*. Ne segue la stessa situazione descritta nelle *Release Notes* del C++ 1.1: si è spesso costretti a rinunciare alla keyword **private** e ad inserire commenti del tipo che riportavo sopra. Se se ne vuole una conferma, basta guardare il manuale del Turbo Vision: nella descrizione della classe *TCollection* (tanto per fare un esempio; ce ne sono molti altri), accanto a tutti i campi-dati si precisa che devono intendersi *read only*, cioè «a sola lettura». Questo vuol dire che se ne sconsiglia vivamente l'*uso*, si sconsiglia cioè di modificarne direttamente il valore, ma ciò nonostante non sono stati dichiarati **private**; il motivo è, con evidenza, che era necessario consentire l'accesso a quei campi da parte di classi derivate definite in altre unit. Vedremo la prossima volta che, per ovviare a questi problemi senza rinunciare alla keyword **private** (e senza incorrere in intollerabili inefficienze), sono stato costretto a di-

```
unit Strings;

interface

uses Base, Magnitud;

type
  PString = ^TString;
  TString = object(TMagnitude)
    constructor Init(s: string);
    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 Str: string;
  private
    p: ^string;
  end;

implementation

constructor TString.Init(s: string);
begin
  GetMem(p, Length(s)+1);
  p^ := s;
end;

destructor TString.Done;
begin
  FreeMem(p, Length(p^)+1);
end;

function TString.Name: string;
begin
  Name := 'String';
end;

function TString.Hash: word;
var
  h,i: word;
begin
  h := 0;
  for i := 1 to Length(p^) do begin
    h := h xor Ord(p^[i]);
    if h >= 32768 then
      h := (h shl 1) + 1
    else
      h := h shl 1;
  end;
  Hash := h;
end;

function TString.IsEqual(var o: TObject): boolean;
begin
  IsEqual := (TypeOf(o) = TypeOf(Self)) and (p^ =
(PString(@o))^.p^);
end;


function TString.IsLessThan(var m: TMagnitude): boolean;
begin
  IsLessThan := p^ < (PString(@m))^.p^;
end;

procedure TString.PrintOn(var f: text);
begin
  Write(f, '''', p^, '''');
end;

function TString.Str: string;
begin
  Str := p^;
end;

end.
```

Figura 4 - La unit STRINGS, che definisce stringhe derivate da TMagnitude e quindi confrontabili secondo il metodo IsLessThan.

chiarare in una stessa unit BAGSET le classi *TBag* e *TSet*.

Zack si è detto d'accordo e, come Anders, ha riconosciuto che il problema è reale e che avrebbe potuto essere risolto in una futura versione. Se la risposta di Anders mi aveva lasciato qualche dubbio (poteva essere un modo elegante di troncare una discussione che rischiava di distoglierlo da altri impegni), il modo con cui Zack mi ha detto quanto vi ho riferito mi fa ritenere che ... possiamo sperare!

Figura 5
Un programma che mostra l'uso della unit SORTCOLL: la variabile Directory conterrà i nomi dei file della directory corrente in ordine alfabetico.

```
Program SortDemo;
(*$X+*)
uses Dos, Base, SortColl, Strings;
var
  DirInfo: SearchRec;
  Directory: TSortedCollection;
  S: PString;
begin
  Directory.Init(20, 5);
  FindFirst('*.*', Archive, DirInfo);
  while DosError = 0 do begin
    S := New(PString, Init(DirInfo.Name));
    Directory.Add(S^);
    FindNext(DirInfo);
  end;
  Directory.PrintOn(StdOut);
end.
```

Le collezioni ordinate

La figura 1 vi propone il sorgente della unit FIXDCOLL: non mi ci soffermo, in quanto si tratta di collezioni indicizzate non espandibili realizzate mediante una semplice derivazione da *TIndexedCollection* (il parametro *Delta* del constructor è sempre zero).

Le figure 2 e 3 illustrano invece le unit ORDCOLL e SORTCOLL. Nella prima vengono definite le *TOrderedCollection*: collezioni indicizzate in cui viene mantenuto l'ordine di inserimento dei diversi elementi, al punto che è possibile toglierne solo l'ultimo elemento che vi è stato aggiunto (liste LIFO, o stack; le *OrderedCollection* dello Smalltalk sono più generali, ma, come vi ho detto più volte, la nostra gerarchia è solo un «demo»). Nella seconda unit troviamo le *TSortedCollection*, in cui inserimento e eliminazione di elementi sono tali da non alterare un ordine dato dal confronto tra essi secondo il metodo *IsLessThan* di ognuno (gli elementi devono quindi appartenere a classi derivate da *TMagnitude*).

Ambedue le classi derivano da *TIndexedCollection*, ma non devono consentire l'accesso alla implementazione delle collezioni indicizzate. Come potete notare, i metodi *AddAt* e *RemoveAt* di *TIndexedCollection* vengono ridefiniti in modo da provocare un errore di esecuzione (in modo analogo a quanto avviene in Smalltalk); se così non fosse, infatti, sarebbe possibile intervenire direttamente sull'array di puntatori e modificarne l'ordine. Per lo stesso motivo il metodo *Remove* viene definito in modo da operare solo sull'ultimo elemento inserito in ORDCOLL, in modo da non lasciare «buchi» nella sequenza ordinata in SORTCOLL (gli elementi successivi a quello rimosso vengono fatti slittare «verso sinistra»). Abbiamo quindi opportunamente nascosto in una sezione **private** di *TIndexedCollection* l'array *A* di puntatori ad oggetti, al quale si può così accedere solo in lettura mediante il metodo *At*. Sarebbe stato meglio poter nascondere l'array in una sezione **protected**. I metodi *Add* e *Remove* di SORTCOLL, infatti, potrebbero essere implementati in modo più efficiente se fossero abilitati a intervenire direttamente sull'array di puntatori, magari con una istruzione *Move*; sono invece «costretti» a operare indirettamente mediante i metodi *Add-At* di *TIndexedCollection*. In alternativa, avrei dovuto rinunciare alla keyword **private** nella unit INDXCOLL; non sarebbe stato un enorme problema (anche nella programmazione normale ci si trova a lavorare spesso «senza rete»: ad esempio quando non si usa la direttiva $R per controllare che l'indice con cui si accede ad un array non cada oltre l'effettiva dimensione di questo), ma è chiaro che una keyword **protected** consentirebbe di ottenere insieme sicurezza ed efficienza.

I benefici della OOP

Immaginiamo di aver bisogno di visualizzare in ordine alfabetico i nomi dei file della directory corrente. Ottenere i nomi è facile, grazie alle procedure *FindFirst* e *FindNext*, ma costruirne un elenco ordinato è tutt'altra cosa. Innanzitutto non ne conosciamo il numero, e quindi abbiamo bisogno di una struttura di dati dinamica; in secondo luogo, dobbiamo pensare all'ordinamento. Potremmo realizzare un array dinamico da sottoporre poi ad un «quicksort», ma forse sarebbe meglio una di quelle «liste auto-ordinate» di cui vi parlavo nel lontano aprile 1988: un programma ALLOC manteneva una lista ordinata di numeri mediante una procedura *Cerca* di appena 27 righe.

Una lista capace però di contenere solo numeri; per tenere in ordine altri tipi dovremmo costruire altre liste.

Questo naturalmente con la programmazione tradizionale. Supponiamo invece ora di avere a disposizione la nostra gerarchia di classi, comprendente anche una classe *TString* come quella illustrata nella figura 4. Il programma che ne viene fuori (figura 5) è lungo in totale 17 righe, delle quali una sola (*Directory.Add(S^);*) si incarica di mantenere la lista ordinata dei nomi dei file. Un bel risparmio! Non solo: la variabile *Directory* appartiene ad un tipo (*TSortedCollection*) che può essere usato per qualsiasi tipo di dati, anche più tipi diversi contemporaneamente (purché derivati da *TObject*). E non crediate che escludo arbitrariamente dal conto delle righe il codice della unit STRINGS, in quanto anche questa unit può essere usata così come è in qualsiasi applicazione, come vedremo la volta prossima.

Se volete un altro esempio, potete riguardare il programma LISTAPPS che vi ho proposto in occasione della prova del Turbo Pascal per Windows: si tratta di una applicazione che visualizza l'elenco ordinato delle intestazioni delle finestre attive, un po' come fa la *Task List* di Windows. Per realizzare l'elenco ho utilizzato la classe *TStrCollection* di ObjectWindows (praticamente identica alla quasi omonima classe del Turbo Vision), e ciò ha consentito di contenere la lunghezza del programma in appena 31 righe. Un gran bel risultato sotto Windows! Nonostante le differenze tra la nostra gerarchia e quelle del Turbo Vision e di ObjectWindows, il principio è lo stesso: ereditarietà e polimorfismo consentono di scrivere codice utilizzabile senza modifiche in una grande varietà di contesti, con ovvio sostanziale incremento della produttività del programmatore.

Quando in OOP si parla di «riusabilità del codice» si fa sul serio!

MC