Insiemi e dizionari

di Sergio Polini
(MC1166 su MC-Link)

Su MC-Link mi è stato chiesto cosa è quel «window superclassing» cui avevo accennato in occasione della prova del Turbo Pascal per Windows. Si tratta di una tecnica adottata nella programmazione sotto Windows di cui non fanno cenno né i manuali dell'SDK né il libro di Petzold. È possibile, quindi, che se lo sia chiesto anche qualcuno di voi. Dirò brevemente che si tratta di un metodo un po' complicato per ottenere (pur con qualche limitazione) una nuova classe di finestre a partire da un'altra, per adeguarne le funzionalità a specifiche esigenze. Si tratta cioè di riprodurre un aspetto tanto fondamentale quanto semplice e immediato nella OOP: la derivazione di una classe da un'altra. Potete trovare fin da ora su MC-Link un file WSUBSUP1.ZIP, che contiene esempi sia di programmazione tradizionale sia del più semplice ricorso alla potenza della OOP con il Turbo Pascal per Windows. È comunque un tema che riprenderemo anche su queste pagine

Come ho detto già la volta scorsa, queste ultime due puntate della serie dedicata alla nostra piccola gerarchia di classi sono particolarmente ricche di listati. Non vedo l'ora infatti (e magari anche voi...) di passare ad esempi di «OOP applicata». Non mi soffermerò quindi più dello stretto necessario sul commento dei sorgenti, preferendo anche questa volta insistere piuttosto sia su particolari tecniche di programmazione sia sui benefici della OOP.

Produttività

Mi capita spesso di sentirmi chiedere se valga davvero la pena «convertirsi» alla OOP. Quando ho necessità di essere sintetico, rispondo più o meno così.

Supponiamo di avere una libreria di funzioni per l'interfaccia utente. Tra le altre, vi sono funzioni che consentono di gestire l'input di stringhe generiche (cioè non formattate) in modo molto sofisticato: scrolling orizzontale e verticale, inserimento, cancellazione e spostamento di caratteri anche a blocchi mediante selezioni operate con mouse o tastiera, ecc.

Il mio problema è che mi serve l'input di stringhe più specifiche, per importi, numeri di telefono con prefisso, codici postali o fiscali, ecc.

Mi serve cioè di «estendere» quella libreria di funzioni.

Se ne ho i sorgenti, me li studio, e se ci capisco qualcosa ho due alternative:
— modificare le funzioni per l'input di stringhe;
— duplicare quelle funzioni, affiancando a quelle originali altre che facciano quello che mi serve.

Gli inconvenienti sono: tempo, rischio di inserire bug nella libreria, duplicazioni di codice.

Se non ho i sorgenti, devo fare tutto da zero o quasi. Con l'ulteriore problema che devo trovare il modo di «incastrare» correttamente le mie funzioni nella struttura della libreria.

Sono cose che credo siano capitate un po' a tutti.

Il beneficio della OOP è questo: parto dalla definizione di una «classe» (ad esempio EDIT) e ne estendo la funzionalità semplicemente derivando da quella un'altra classe (ad esempio NUMEDIT), operazione per la quale mi si richiede solo di specificare in cosa la nuova classe si differenzia dalla prima (ad esempio, nella funzione *InputChar* rifiutare i caratteri non numerici). Per far questo:
— non ho bisogno dei sorgenti della libreria;
— non ho bisogno di alterare in alcun modo struttura e funzionalità della classe originaria;
— non ho bisogno né di modificare né di ricompilare quelle parti del mio programma che usano la classe originaria;
— non ho bisogno né di modificare né di ricompilare neppure quelle parti del mio programma che possono usare indifferentemente la classe originaria e quella derivata (ad esempio quelle che visualizzano le maschere per l'input di dati, magari sotto forma di dialog box).

Una flessibilità che si traduce in minor codice, minor tempo, minore possibilità di bug. Ovvero, in positivo, maggiore produttività.

Ma rispondo così solo quando devo contare le parole. C'è dell'altro.

Hashing

Le collezioni fin qui viste (indicizzate, di dimensione fissa o variabile, più o meno ordinate) potranno esservi sembrate

niente più che una generalizzazione dei tradizionali array. Magari perché vi siete lasciati ingannare dal fatto che, per la loro implementazione, mi sono servito di array di puntatori. Il breve esempio visto la volta scorsa, tuttavia, era inteso a mostrarvi che una *TSortedCollection*, indipendentemente dalla sua implementazione, si presta benissimo alla soluzione di problemi che altrimenti avrebbero potuto richiedere la realizzazione di strutture di dati dinamiche come le liste.

L'impossibilità di ridurre le classi tipiche della OOP a semplici varianti delle strutture di dati tradizionali è ancora più evidente con quelle che vi propongo ora. *TBag* è un aggregato non indicizzato di oggetti, che vanno semplicemente considerati come messi tutti nello stesso «sacco»; posso solo verificare la presenza o meno di un oggetto. Nella nostra versione semplificata, non posso nemmeno sapere quante volte uno stesso oggetto ricorre nel «sacco» (in Smalltalk la classe Bag ha anche un metodo *occurrencesOf*:). Più utile la classe *TSet*, che deriva da *TBag*, ma ne differisce in quanto ogni oggetto vi può comparire una sola volta: prima di aggiungere un elemento all'insieme si verifica che non ve ne sia già uno uguale. Un *TSet* è quindi un *TBag*, «senza duplicazioni». Un *TDictionary* è un insieme come gli altri, ma ammette solo elementi di tipo *TAssociation*; ognuno di questi è una coppia chiave-valore, in cui la «chiave» deve essere un oggetto derivato da *TMagnitude* e il «valore» può essere un oggetto qualsiasi (cioè qualsiasi cosa derivata da *TObject*).

Anche per le classi *TBag* e derivate l'implementazione può essere ricondotta a strutture di dati tradizionali: alla base di tutto vi è il solito array di puntatori, in cui aggiunta e cancellazione di elementi avvengono con un meccanismo di *hashing*. Gli elementi non vengono aggiunti incrementando l'indice dell'array, in quanto ciò rischierebbe di rendere poco efficiente l'operazione che più frequentemente ci si attende di compiere: la ricerca della presenza o meno di un oggetto nel contenitore (nel caso peggiore, occorrerebbe esaminare tutti gli elementi dell'array prima di concludere che quello che ci interessa non è presente). Di ogni elemento si calcola quindi un numero detto *valore di hash*, che viene ricondotto ad un numero h non superiore all'indice massimo mediante l'operatore *mod*; l'elemento viene poi inserito nell'indice h dell'array se qui non ve n'è già un altro (si-

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

uses Base, Collect;

type
  PBag = ^TBag;
  TBag = object(TCollection)
    constructor Init(Size: word);
    destructor Done; virtual;
    function Name: string; virtual;
    function InitIterator: PIterator; virtual;
    function Add(var o: TObject): PObject; virtual;
    function Remove(var o: TObject): PObject; virtual;
    function Find(var o: TObject): PObject; virtual;
  private
    A: PObjPArray;
    Sz, Count: word;
    procedure Grow;
  end;
  PSet = ^TSet;
  TSet = object(TBag)
    function Name: string; virtual;
    function Add(var o: TObject): PObject; virtual;
  end;
  PBagIterator = ^TBagIterator;
  TBagIterator = object(TIterator)
    constructor Init(var c: TBag);
    destructor Done; virtual;
    function More: boolean; virtual;
    function Next: PObject; virtual;
  private
    Bg: PBag;
    CurrentIndex: word;
    BCount: word;
  end;


  implementation

  constructor TBag.Init(Size: word);
  var
    i: word;
  begin
    Sz := Size;
    Count := 0;
    GetMem(A, Sz * SizeOf(PObject));
    for i := 1 to Sz do A^[i] := Null;
  end;


  destructor TBag.Done;
  begin
    FreeMem(A, Sz * SizeOf(PObject));
  end;

  function TBag.Name: string;
  begin
    Name := 'Bag';
  end;


  function TBag.InitIterator: PIterator;
  var
    Iterator: PBagIterator;
  begin
    New(Iterator, Init(Self));
    InitIterator := PIterator(Iterator);
  end;
```

```
function TBag.Add(var o: TObject): PObject;
var
  h: word;
begin
  if Count >= (Sz div 4) * 3 then Grow;
  h := (o.Hash mod Sz) + 1;
  while A^[h] <> PObject(Null) do
    if h = 1 then h := Sz else Dec(h);
  A^[h] := @o;
  Inc(Count);
  Add := A^[h];
end;


function TBag.Remove(var o: TObject): PObject;
var
  h,k,r: word;
begin
  h := (o.Hash mod Sz) + 1;
  while (not A^[h]^.IsEqual(o)) and (A^[h] <> PObject(Null)) do
    if h = 1 then h := Sz else Dec(h);
  if A^[h] = PObject(Null) then begin
    Remove := Null;
    Exit;
  end

  else begin
    Dec(Count);
    Remove := A^[h];
    repeat
      A^[h] := Null;
      k := h;
      repeat
        if h = 1 then h := Sz else Dec(h);
        if A^[h] = PObject(Null) then Exit;
        r := (A^[h]^.Hash mod Sz) + 1;
      until not(((h <= r) and (r < k)) or ((r < k) and (k < h))
                or ((k < h) and (h <= r)));
      A^[k] := A^[h];
    until false;
  end;
end;


function TBag.Find(var o: TObject): PObject;
var
  h: word;
begin
  h := (o.Hash mod Sz) + 1;
  while (A^[h] <> PObject(Null)) and (not A^[h]^.IsEqual(o)) do
    if h = 1 then h := Sz else Dec(h);
  Find := A^[h];                              (* Null o A^[h] *)
end;


procedure TBag.Grow;
var
  OldSize, i: word;
  OldA: PObjPArray;
begin
  OldSize := Sz;
  OldA := A;
  Sz := Sz + Sz;
  Count := 0;
  GetMem(A, Sz * SizeOf(PObject));
  for i := 1 to Sz do A^[i] := Null;
  for i := 1 to OldSize do
    if OldA^[i] <> PObject(Null) then Add(OldA^[i]^);
  FreeMem(OldA, OldSize * SizeOf(PObject));
end;
```

```
function TSet.Name: string;
begin
  Name := 'Set';
end;

function TSet.Add(var o: TObject): PObject;
var
  h: word;
begin
  if Count >= (Sz div 4) * 3 then Grow;
  h := (o.Hash mod Sz) + 1;
  while (A^[h] <> PObject(Null)) and (not A^[h]^.IsEqual(o)) do
    if h = 1 then h := Sz else Dec(h);
  if A^[h] = PObject(Null) then begin
    A^[h] := @o;
    Inc(Count);
    Add := A^[h];
  end
  else
    Add := Null;
end;

constructor TBagIterator.Init(var c: TBag);
begin
  Bg := @c;
  CurrentIndex := 1;
  BCount := 0;
end;

destructor TBagIterator.Done;
begin
end;


function TBagIterator.More: boolean;
begin
  More := BCount < Bg^.Count;
end;


function TBagIterator.Next: PObject;
begin
  while (CurrentIndex <= Bg^.Sz) and (Bg^.A^[CurrentIndex] = PObject(Null)) do
    Inc(CurrentIndex);
  if CurrentIndex <= Bg^.Sz then begin
    Next := Bg^.A^[CurrentIndex];
    Inc(CurrentIndex);
    Inc(BCount);
  end
  else
    Next := Null;
end;

end.
```

Figura 1 - La unit BAGSET, che definisce e implementa le classi TBag e TSet.

```
unit Diction;

interface

uses Base, Assoc, Collect, BagSet;

type
  PDictionary = ^TDictionary;
  TDictionary = object(TSet)
    function Name: string; virtual;
    function Add(var o: TObject): PObject; virtual;
    function Find(var o: TObject): PObject; virtual;
  end;

implementation

function TDictionary.Name: string;
begin
  Name := 'Dictionary';
end;

function TDictionary.Add(var o: TObject): PObject;
begin
  if TypeOf(o) = TypeOf(TAssociation) then Add := TSet.Add(o)
  else Add := Null;
end;

function TDictionary.Find(var o: TObject): PObject;
var
  Iterator: PIterator;
  P: PAssociation;
begin
  Iterator := InitIterator;
  while Iterator^.More do begin
    P := PAssociation(Iterator^.Next);
    if P^.GetKey^.IsEqual(o) then begin
      Dispose(Iterator, Done);
      Find := P;
      Exit;
    end;
  end;
  Dispose(Iterator, Done);
  Find := Null;
end;

end.
```

Figura 2 - La unit DICTION. I «dizionari» sono insiemi i cui elementi appartengono tutti alla classe TAssociation.

tuazione indicata col nome di «collisione»), altrimenti si decrementa h fino a trovare uno spazio vuoto. La ricerca è molto rapida: si ricalcola h e si parte dall'elemento con indice h; si termina non appena si trova un elemento uguale a quello cercato (che risulta quindi presente) o uno spazio vuoto (che indica l'assenza dell'elemento, in quanto se presente sarebbe stato messo proprio lì).

Si tratta di una tecnica molto efficiente a condizione di non ridursi mai ad un array «troppo pieno», che provocherebbe un eccessivo numero di collisioni. È stato calcolato che è sufficiente tenere costantemente almeno un quarto dell'array vuoto (cfr. Donald E. Knuth, *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*, Addison-Wesley, 1973, dal quale ho tratto gli algoritmi per BAGSET); è questo il motivo per cui, quando si aggiunge un elemento, si verifica per prima cosa che l'array non sia già pieno per il 75%: in caso affermativo la dimensione dell'array viene raddoppiata.

L'unica operazione un po' delicata è l'eliminazione di un elemento dall'array. Immaginiamo di aver inserito due volte un oggetto K, prima all'indice h e poi all'indice h-1 dell'array A. Se per eliminare K ci limitassimo ad azzerare A[h], una successiva ricerca di K ci direbbe che non è presente nell'array, in quanto troverebbe vuoto A[h], nonostante che in realtà ci sia un K in A[h-1]. Occorre quindi anche far scorrere in avanti il secondo K, da A[h-1] a A[h]. Analogamente nel caso in cui vi siano più di due K nell'array.

Sappiamo ora finalmente perché ogni classe della nostra gerarchia definisce o eredita un metodo *Hash*: proprio per consentire l'implementazione appena descritta di *TBag* e delle classi da questa derivate. Anche in Smalltalk e nell'Objective-C si usano tecniche analoghe, come dimostra la presenza anche lì di metodi *Hash*. Mi sono quindi sof-

```
unit Assoc;

interface

uses Base, Magnitud;

type
  PAssociation = ^TAssociation;
  TAssociation = object(TMagnitude)
    constructor Init(var AKey: TMagnitude; var AValue: TObject);
    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 GetKey: PObject;
    function GetValue: PObject;
  private
    Key: PMagnitude;
    Value: PObject;
  end;


  implementation

  constructor TAssociation.Init(var AKey: TMagnitude; var AValue: TObject);
  begin
    Key := @AKey;
    Value := @AValue;
  end;


  destructor TAssociation.Done;
  begin
  end;


  function TAssociation.Name: string;
  begin
    Name := 'Association';
  end;
```

```
function TAssociation.Hash: word;
begin
  Hash := Key^.Hash;
end;

function TAssociation.IsEqual(var o: TObject): boolean;
begin
  IsEqual := (TypeOf(o) = TypeOf(Self)) and
             (Key^.IsEqual(PAssociation(@o)^.Key^));
end;

function TAssociation.IsLessThan(var m: TMagnitude): boolean;
begin
  IsLessThan := Key^.IsLessThan(PAssociation(@m)^.Key^);
end;

procedure TAssociation.PrintOn(var f: text);
begin
  Write(f, '(');
  Key^.PrintOn(f);
  Write(f, ',');
  Value^.PrintOn(f);
  Write(f, ')');
end;


function TAssociation.GetKey: PObject;
begin
  GetKey := Key;
end;


function TAssociation.GetValue: PObject;
begin
  GetValue := Value;
end;

end.
```

Figura 3 - La unit ASSOC. Ogni istanza di TAssociation è una coppia (chiave, valore).

fermato sull'*hashing* sia perché relativamente diffuso nel mondo OOP, sia perché potrebbe magari risultarvi interessante comunque, anche cioè nel caso ... vi ostiniate a programmare vecchia maniera.

In realtà, purché siano rispettati ragionevoli criteri di efficienza, non ha alcuna importanza quale sia l'implementazione adottata: dal punto di vista di chi usa una classe, ciò che conta è la sua interfaccia, che può restare invariata anche cambiando totalmente l'implementazione (si potrebbero usare alberi binari, o altro).

Ancora produttività

La gerarchia del Turbo Vision (come anche quella dell'ObjectWindows del Turbo Pascal per Windows) non comprende classi come *TBag, TSet o TDictionary*, ma sarebbe facile aggiungerle.

È anche possibile usare le collezioni indicizzate del Turbo Vision come se fossero *TBag* o *TSet* agendo sul campo boolean *Duplicated* (falso per default, in modo da escludere «duplicazioni», può essere reso vero per consentirle) e tralasciando di usare i metodi che danno accesso agli elementi mediante indici.

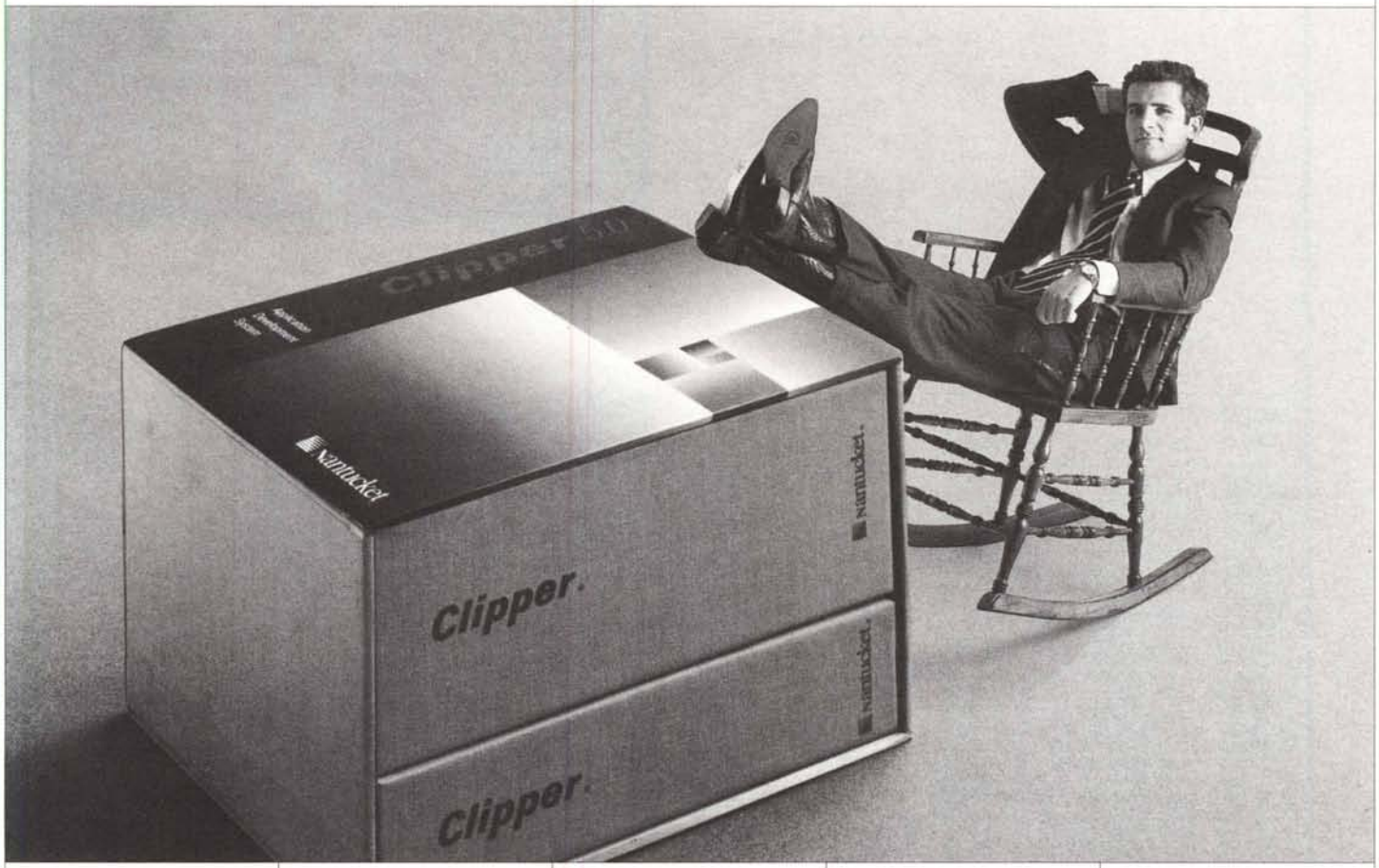
Quello che soprattutto importa, è che si sappia sempre più rinunciare alle strutture di dati tradizionali. Finché si usano queste, si è praticamente costretti a «reinventare la ruota» ogni volta: tante strutture quanti sono i tipi di dati che vogliamo «strutturare». A me è capitato, in passato, di creare liste o alberi diversi per quasi tutti i programmi che ho realizzato, nonostante si trattasse in fondo sempre di niente altro che liste e alberi: l'unica cosa che rimaneva più o meno costante (e non sempre!) erano gli algoritmi impiegati, ma tutto il resto era ogni volta diverso. Smalltalk ci ha insegnato che è possibile ragionare in tutt'altro modo, muovendo dall'assunto che ogni struttura di dati non è altro che una «collezione», cioè un «gruppo di oggetti». Definita quindi una classe astratta che valga come prototipo per tutte le altre, è facile sia trovare la classe che meglio fa al caso nostro tra quelle da questa derivate, sia, ove necessario, derivare classi ulteriori. Grazie al polimorfismo, una qualsiasi collezione definita per oggetti appartenenti ad una classe può ospitare senza modifiche anche oggetti che siano istanze di classi da questa derivate. Abbiamo quindi ampia libertà di creare nuove classi di oggetti senza necessità di modificare le nostre collezioni.

Per trarre vantaggio da questa importante evoluzione occorre qualcosa di più di quanto potrebbe richiedere l'uso di un nuovo algoritmo o di una nuova libreria di funzioni: è necessario «pensare» in modo diverso alla programmazione. Ne è un esempio il breve programma illustrato il mese scorso (appena una quindicina di righe di codice per ottenere un elenco ordinato dei file nella directory corrente), come anche, e forse più, quel PREMAKE che vi ho proposto ad aprile: poche decine di righe di codice per implementare funzionalità che due anni fa avevano richiesto la costruzione di complicate liste di liste. In entrambi i casi non abbiamo fatto altro che usare la nostra piccola gerarchia in luogo di tecniche tradizionali, conseguendo effettivamente quella maggiore produttività che la OOP promette.

A partire dal mese prossimo cercheremo di ottenere analoghi vantaggi nello sviluppo di applicazioni «vere» con uso di «vere» gerarchie di classi.

MC

PROGRAMMATE IL VOSTRO FUTURO.



Programmatelo in tutta libertà senza porre limiti alla vostra fantasia. Clipper 5.0, la più recente versione del noto sistema di sviluppo, prodotto dalla Nantuket e distribuito in Italia da Algol è lo strumento ideale per sviluppare i vostri programmi con la massima libertà e sicurezza.

Un'ampia gamma di comandi e di funzioni.

Un nuovo linker (RTLink), che permette di superare senza inconvenienti la barriera dei 640 Kb di memoria.

Un pre-processore flessibile che consente di ottimizzare il codice, di avere un maggior controllo dei programmi e di personalizzarne il linguaggio.

Una nuova e migliorata documentazione disponibile On-Line.

Un compilatore ad alte prestazioni che assicura l'alta velocità di esecuzione, la sicurezza del codice sorgente e la possibilità di distribuire le applicazioni sia in ambiente di rete che single-user senza bisogno di software aggiuntivo, fanno di Clipper 5.0 il miglior investimento per il vostro futuro.

L'ambiente di sviluppo può essere integrato con una serie di Utilities che vi permetteranno di migliorare ulteriormente le prestazioni di Clipper 5.0.

FUNCky.

La libreria per Clipper più venduta negli USA. Comprende funzioni per la gestione del disco fisso, mouse, video, stringhe, porta seriale e file.

BLINKER.

Dynamic Overlay Linker, da 5 a 10 volte più veloce di RTLink. Permette di creare versioni demo di un programma che terminano dopo n minuti o che funzionano solo fino ad una certa data.

dbPUBLISHER.

Creazione di applicazioni di publishing.

NETLIB.

Libreria per applicazioni Clipper su rete. Supporta Novell, Lan Manager e reti NetBios compatibili.

SILVERCOMM.

Libreria per la gestione della porta seriale RS-232 con Clipper.

DGE 4.0/SILVERPAINT.

Librerie grafiche per Clipper.

OVERLAY().

Gestione della memoria espansa/estesa con Clipper 5.0

BRIEF/dBRIEF.

Editor programmabile per Clipper.

SUBNTX().

Manipolazione dei file indici NTX.

SPELLCODE.

Controllo sintattico di programmi e funzioni.

Programmate la vostra libertà.

ALGOL

ALGOL S.p.A.
VIA FELTRE 28/6, 20132 MILANO
TEL. 02.26411411 (r.a.)
FAX 02.2154629, B.B.S. 02.26413589

FILIALI:
IVREA, TEL. 0125.424541/2,
FAX 0125.47061

ROMA, TEL. 06.5913971/5919479,
FAX 06.5918745