Abgabetermin: Mi, 21. 06. 2023

☒ **Gr. 1,** Dr. S. Wagner ☐ **Gr. 2,** Dr. D. Auer ☐ **Gr. 3,** Dr. G. Kronberger

Name Elias Wurm **Aufwand in h** 3

Punkte _____ **Kurzzeichen Tutor / Übungsleiter*in** _____ / _____

1. Ein neuer Behälter für die MiniLib (12 Punkte)

Die MiniLib bietet bereits die Behälterklasse *MLVector* mit entsprechendem Iterator. Eine Behälterklasse auf Basis einer dynamischen Liste fehlt jedoch noch.

Analysieren Sie zunächst die Klassen *MLCollection* und *MLIterator* und leiten Sie davon Ihre Lösung für eine dynamische Liste ab.

a) Implementieren Sie eine neue Behälterklasse *MLList*, die eine einfach-verkettete, nichtzyklische dynamische Liste realisiert. Implementieren Sie alle notwendigen Methoden (siehe *MLCollection*).

b) Ergänzen Sie zudem eine Methode *Prepend*.

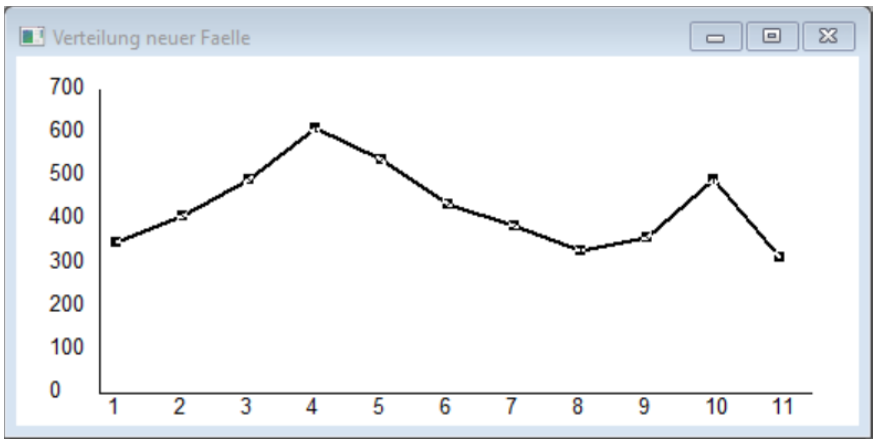
c) Entwickeln Sie einen Iterator *MLListIterator*, der die Liste vom ersten bis zum letzten Knoten durchläuft.

Testen Sie die Liste und den Iterator ausführlich (d.h. alle Methoden) und füllen Sie verschiedene Listen mit Objekten verschiedener MiniLib-Klassen.

2. Liniendiagramme (12 Punkte)

Entwickeln Sie ein MiniLib-Programm, das eine Datenreihe aus einer Textdatei *input.txt* einliest und als Liniendiagramm darstellt. Die erste Zeile der Textdatei enthält den Diagrammtitel (z.B. `Verteilung neuer Faelle`), die restlichen Zeilen enthalten jeweils einen ganzzahligen Wert aus dem Wertebereich 0 bis 1000. Ein Fenster stellt die Datenreihe als Liniendiagramm dar.

Beispiel:



Passen Sie die *y*-Werte an die jeweilige Fensterhöhe an, sodass möglichst die gesamte Höhe für die Darstellung der Linien genutzt wird (z.B. der maximale Wert der Datenreihe aufgerundet auf die Hunderterstelle nimmt die gesamte Höhe in Anspruch).

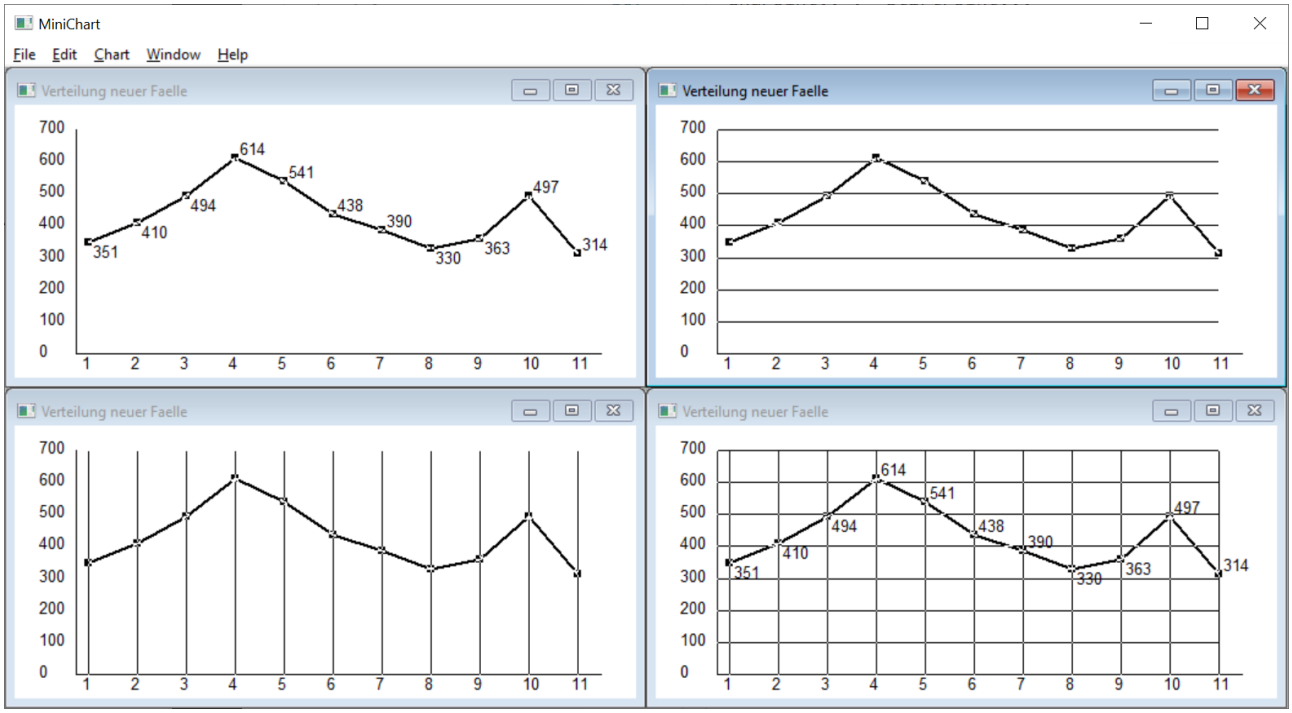
Die Anordnung der Datenpunkte entlang der *x*-Achse und die Beschriftung der *x*-Achse erfolgt unabhängig von der Fensterbreite.

Die Beschriftung der x-Achse erfolgt durch eine fortlaufende Nummer (beginnend mit 1), die Beschriftung der *y*-Achse erfolgt in 100er-Schritten.

Implementieren Sie darüber hinaus verschiedene Darstellungsformen (Beschriftung der Linie mit Werten der Datenreihe, horizontale und vertikale Rasterlinien), die über Menüeinträge im Menu *Chart* für das Diagramm im aktiven Fenster eingestellt werden können.

Zum Abschluss implementieren Sie die Möglichkeit, den Wert eines Datenpunkts durch eine Mausbewegung zu verändern. Hierfür sollten Sie die *MLWindow*-Methoden *OnMousePressed*, *OnMouseMove* und *OnMouseReleased* überschreiben. Beachten Sie, dass die y-Skalierung dabei nicht geändert werden muss, selbst wenn der maximale Wert der Datenreihe durch die Mausbewegung verändert wird.

Beispiele:



Hinweise:

1. Geben Sie für alle Ihre Lösungen immer eine „Lösungsidee“ an.
2. Dokumentieren und kommentieren Sie Ihre Algorithmen.
3. Bei Programmen: Geben Sie immer auch Testfälle ab, an denen man erkennen kann, dass Ihr Programm funktioniert, und dass es auch in Fehlersituation entsprechend reagiert.

Aufgabe 1 – Ein neuer Behälter für die MiniLib

Lösungsidee:

Neues Objekt MLList anlegen und Methoden von MLColl ableiten, in Obj die KopfNode speichern und alle Standardmethoden implementieren.

Für Prepend die neue Node am Anfang der Liste einfügen, bei Add am Ende der Liste.

Für den Iterator die KopfNode beim Init speichern und bei jedem Methodenaufruf von Next den aktuell gespeicherten Wert zurückgeben und Next auf die nächste Node setzten.

Zeitaufwand: ~1h

Code:

```
unit MLLi;

interface

uses
  MLObj, MLColl;

type
  MLListNodePtr = ^MLListNode;
  MLListNode = record
    obj: MLObject;
    next: MLListNodePtr;
  end;

  (* === class MLList === *)

  MLList = ^MLListObj;
  MLListObj = object(MLCollectionObj)
    head: MLListNodePtr;

    constructor Init;
    destructor Done; virtual;

    function Size: INTEGER; virtual;
    procedure Add(o: MLObject); virtual;
    function Remove(o: MLObject): MLObject; virtual;
    function Contains(o: MLObject): BOOLEAN; virtual;
    procedure Clear; virtual;
    function NewIterator: MLIterator; virtual;
```

```
    procedure Prepend(o: MLObject);
  end;

  (* === class MLListIterator === *)

  MLListIterator = ^MLListIteratorObj;
  MLListIteratorObj = object(MLIteratorObj)
    curNode: MLListNodePtr;

    constructor Init(l: MLList);
    destructor Done; virtual;

    function Next: MLObject; virtual;
  end;

function NewMLList: MLList;

implementation

function NewMLList: MLList;
var
  l: MLList;
begin
  New(l, Init);
  NewMLList := l;
end;

(* === class MLList === *)

constructor MLListObj.Init;
begin
  inherited Init;
  Register('MLList', 'MLCollection');
  head := NIL;
end;

destructor MLListObj.Done;
begin
  Clear;
  inherited Done;
end;

function MLListObj.Size: INTEGER;
var
  count: INTEGER;
```

```
  curNode: MLListNodePtr;
begin
  count := 0;
  curNode := head;
  while curNode <> NIL do
  begin
    curNode := curNode^.next;
    Inc(count);
  end;
  Size := count;
end;

procedure MLListObj.Add(o: MLObject);
var
  newNode: MLListNodePtr;
  curNode: MLListNodePtr;
begin
  New(newNode);
  newNode^.obj := o;
  newNode^.next := NIL;

  if head = NIL then
    head := newNode
  else begin
    curNode := head;
    while curNode^.next <> NIL do
      curNode := curNode^.next;
    curNode^.next := newNode;
  end;
end;

function MLListObj.Remove(o: MLObject): MLObject;
var
  prevNode, curNode: MLListNodePtr;
begin
  if head = NIL then
  begin
    Remove := NIL;
    Exit;
  end;

  prevNode := NIL;
  curNode := head;
  while (curNode <> NIL) and (curNode^.obj <> o) do
  begin
    prevNode := curNode;
```

```
    curNode := curNode^.next;
  end;

  if curNode = NIL then
  begin
    Remove := NIL;
    Exit;
  end;

  if prevNode = NIL then
    head := curNode^.next
  else
    prevNode^.next := curNode^.next;

  Remove := curNode^.obj;
  Dispose(curNode);
end;

function MLListObj.Contains(o: MLObject): BOOLEAN;
var
  curNode: MLListNodePtr;
begin
  curNode := head;
  while curNode <> NIL do
  begin
    if curNode^.obj^.IsEqualTo(o) then
    begin
      Contains := TRUE;
      Exit;
    end;
    curNode := curNode^.next;
  end;
  Contains := FALSE;
end;

procedure MLListObj.Clear;
var
  curNode, nextNode: MLListNodePtr;
begin
  curNode := head;
  while curNode <> NIL do
  begin
    nextNode := curNode^.next;
    Dispose(curNode^.obj, Done);
    Dispose(curNode);
    curNode := nextNode;
```

```
  end;
  head := NIL;
end;

function MLListObj.NewIterator: MLIterator;
var
  iterator: MLListIterator;
begin
  New(iterator, Init(@Self));
  NewIterator := iterator;
end;

procedure MLListObj.Prepend(o: MLObject);
var
  newNode: MLListNodePtr;
begin
  New(newNode);
  newNode^.obj := o;
  newNode^.next := head;
  head := newNode;
end;

(* === class MLListIterator === *)

constructor MLListIteratorObj.Init(l: MLList);
begin
  inherited Init;
  curNode := l^.head;
end;

destructor MLListIteratorObj.Done;
begin
  inherited Done;
end;

function MLListIteratorObj.Next: MLObject;
begin
  if curNode <> NIL then
  begin
    Next := curNode^.obj;
    curNode := curNode^.next;
  end
  else
    Next := NIL;
end;
```

```
end.
```

Tests:

```
program MLListTest;

uses MLLi, MLObj, MLInt, MLColl, MetaInfo;

procedure RunMLListTests;
var
  list: MLList;
  int2: MLInteger;
  iterator: MLIterator;
  next: MLObject;
begin
  list := NewMLList;

  int2 := NewMlInt(2);

  list^.Add(NewMLInt(1));
  list^.Add(int2);
  list^.Add(NewMLInt(3));

  writeln('Size: ', list^.Size); // Output: 3

  writeln('Removed int2: ', list^.Remove(int2)^.AsString);

  writeln('Contains int2: ', list^.Contains(int2)); // Output: False
  Dispose(int2, Done);

  iterator := list^.NewIterator;
  next := iterator^.Next;

  while next <> NIL do
  begin
    writeln('Iterator value: ', next^.asString); // Output: 1, 3
    next := iterator^.Next;
  end;
  writeln;
  Dispose(iterator, Done);

  list^.Prepend(NewMLInt(4));
```

```
  iterator := list^.NewIterator;
  next := iterator^.Next;

  while next <> NIL do
  begin
    writeln('Iterator value: ', next^.asString); // Output: 4, 1, 3
    next := iterator^.Next;
  end;
  writeln;
  Dispose(iterator, Done);

  list^.Clear;

  writeln('Size: ', list^.Size); // Output: 0

  Dispose(list, Done);
end;

begin
  RunMLListTests;
  WriteMetaInfo;
end.
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Size: 3
Removed int2: 2
Contains int2: FALSE
Iterator value: 1
Iterator value: 3

Iterator value: 4
Iterator value: 1
Iterator value: 3

Size: 0


============================================================
 Meta information for MiniLib application
------------------------+-----------------------------------
 Class hierarchy        | Number of dynamic objects
                        +---------+---------+-------------
                        | created | deleted | still alive
------------------------+---------+---------+-------------
 MLObject               |       0 |       0 |           0
   MLCollection         |       0 |       0 |           0
     MLList             |       1 |       1 |           0
   MLInt                |       4 |       4 |           0
   MLIterator           |       2 |       2 |           0
------------------------+---------+---------+-------------
 Number of classes:  5  | Summary: all objects deleted
============================================================

Heap dump by heaptrc unit of C:\_data\fh-repos\2023SS_ADF\UE10\MLListTest.exe
37 memory blocks allocated : 2300/2432
37 memory blocks freed     : 2300/2432
0 unfreed memory blocks : 0
True heap size : 163840 (112 used in System startup)
True free heap : 163728
```

Aufgabe 2 - Liniendiagram

Lösungsidee:

Das Auslesen und speichern der gelesenen Werte wird vor aufrufen der Application durchgeführt. Danach wird der gelesene Titel für das neue Window verwendet und im Window die gelesenen Daten gespeichert. Im Konstruktor des Windows wird auch der höchste Wert ermittelt (hätte man auch schon in der Application machen können), da der sich beim neu zeichnen nie ändert.

Die gesamten Chart Kalkulationen und zeichnen passieren alles in der Draw Methode (mit mehr Zeitaufwand hätte man das auch aufteilen und optimieren können, da die Methode ziemlich groß geworden ist und teilweise redundanten Berechnungen ausgeführt werden), dafür wird anhand der Höhe und Breite des Fensters mit der Anzahl der Werten und dem höchsten Wert der Abstand der unterschiedlichen Werte am Screen berechnet damit danach die Achsen und Achsen Beschriftung so viel vom Screen einnimmt wie möglich. Nachdem werden die Werte eingetragen, dafür wird die y Position relativ zum Koordinaten Ursprung und Max Höhe gerechnet.

Zeitaufwand: ~2h

Code:

```
program LineChart;

uses
  MetaInfo, OSBridge,
  MLObj, MLWin, MLAppl, MLVect, MLInt, MLColl;

type
  ChartType = (Simple, VerticalLines, HorizontalLines, GridLines);

  ChartWindow = ^ChartWindowObj;
  ChartWindowObj = object(MLWindowObj)
    data: MLVector;
    cType: ChartType;
    maxVal: integer;

    constructor Init(title: STRING; data: MLVector);
    destructor Done; virtual;
    (*overridden methods*)
    procedure Open; virtual;
    procedure Redraw; virtual;
    procedure OnCommand(commandNr: INTEGER); virtual;
    (*new methods*)
    procedure DrawChart;
    procedure ChangeChartType(cType: ChartType);
```

```
  end; (*OBJECT*)

  ChartApplication= ^ChartApplicationObj;
  ChartApplicationObj = object(MLApplicationObj)
    title: STRING;
    data: MLVector;

    constructor Init(name, filename: STRING);
    destructor Done; virtual;
    (*overridden methods*)
    procedure OpenNewWindow; virtual;
    procedure BuildMenus; virtual;
    (*new methods*)
    procedure ExtractDataFromFile(filename: STRING);
  end; (*OBJECT*)

var
  (*chart types:*)
  simpleLinesCommand, verticalLinesCommand, horizontalLinesCommand,
gridLinesCommand: INTEGER;


(*=== ChartWindow ===*)

function NewChartWindow(title: string; data: MLVector): ChartWindow;
var
  w: ChartWindow;
begin
  New(w, Init(title, data));
  NewChartWindow := w;
end; (*NewChartWindow*)

constructor ChartWindowObj.Init(title: STRING; data: MLVector);
var
  iterator: MLIterator;
  next: MLObject;
begin
  inherited Init(title);
  Register('ChartWindow', 'MLWindow');
  cType := Simple;
  self.data := data;

  maxVal := MLInteger(data^.GetAt(data^.Size))^.AsInteger;

  iterator := data^.NewIterator;
  next := iterator^.Next;
  maxVal := MLInteger(next)^.AsInteger;
  while next <> NIL do
  begin
```

```
    if (MLInteger(next)^.AsInteger > maxVal) then
      maxVal := MLInteger(next)^.AsInteger;
    next := iterator^.Next;
  end;
  maxVal := Round(maxVal / 100) + 1; // round up to nearest multiple of 100

  Dispose(iterator, Done);
end; (*ChartWindowObj.Init*)

destructor ChartWindowObj.Done;
begin
  Dispose(data, Done);
  inherited Done;
end; (*ChartWindowObj.Done*)

procedure ChartWindowObj.Open;
begin
  inherited Open;
  DrawChart;
end; (*ChartWindowObj.Open*)

procedure ChartWindowObj.Redraw;
begin
  DrawChart;
end; (*ChartWindowObj.Redraw*)

procedure ChartWindowObj.OnCommand(commandNr: INTEGER);
begin
  if commandNr = simpleLinesCommand then
    ChangeChartType(Simple)
  else if commandNr = verticalLinesCommand then
    ChangeChartType(VerticalLines)
  else if commandNr = horizontalLinesCommand then
    ChangeChartType(HorizontalLines)
  else if commandNr = gridLinesCommand then
    ChangeChartType(GridLines)
  else
    inherited OnCommand(commandNr);
end; (*ChartWindowObj.OnCommand*)

procedure ChartWindowObj.ChangeChartType(cType: ChartType);
begin
  DrawChart;
  self.cType := cType;
  DrawChart;
end; (*ChartWindowObj.ChangeChartType*)

procedure ChartWindowObj.DrawChart;
```

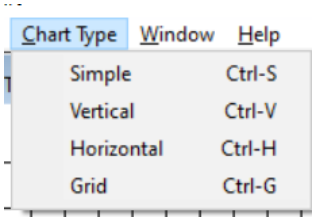
```
var
  currVal, horizontalGap, verticalGap, i: integer;
  origin, dummy1, dummy2: Point;
  curr, prev: Point;
  intStr: string;
begin
  horizontalGap := (Width - 80) div (data^.Size - 1);
  verticalGap := (Height - 50) div maxVal;

  origin.x := 50;
  origin.y := Height - 20;

  // draw axes
  dummy1.x := origin.x + horizontalGap * (data^.Size - 1);
  dummy1.y := origin.y;
  DrawLine(origin, dummy1, 1);

  dummy1.x := origin.x;
  dummy1.y := origin.y - verticalGap * maxVal;
  DrawLine(origin, dummy1, 1);

  // draw grid horizontal
  dummy1.x := origin.x - 30;
  dummy1.y := origin.y - 10;

  for i := 0 to maxVal do
  begin
    Str(i*100, intStr);
    DrawString(dummy1, intStr, 10);
    dummy1.y := dummy1.y - verticalGap;
  end;

  dummy2.x := origin.x + horizontalGap * (data^.Size - 1);
  dummy2.y := origin.y;

  dummy1.x := origin.x;
  dummy1.y := origin.y;

  if (cType = GridLines) or (cType = VerticalLines) then
    for i := 1 to maxVal do
    begin
      dummy1.y := dummy1.y - verticalGap;
      dummy2.y := dummy2.y - verticalGap;
      DrawLine(dummy1, dummy2, 1);
    end;

  // draw grid vertical
  dummy1.x := origin.x - 2;
```

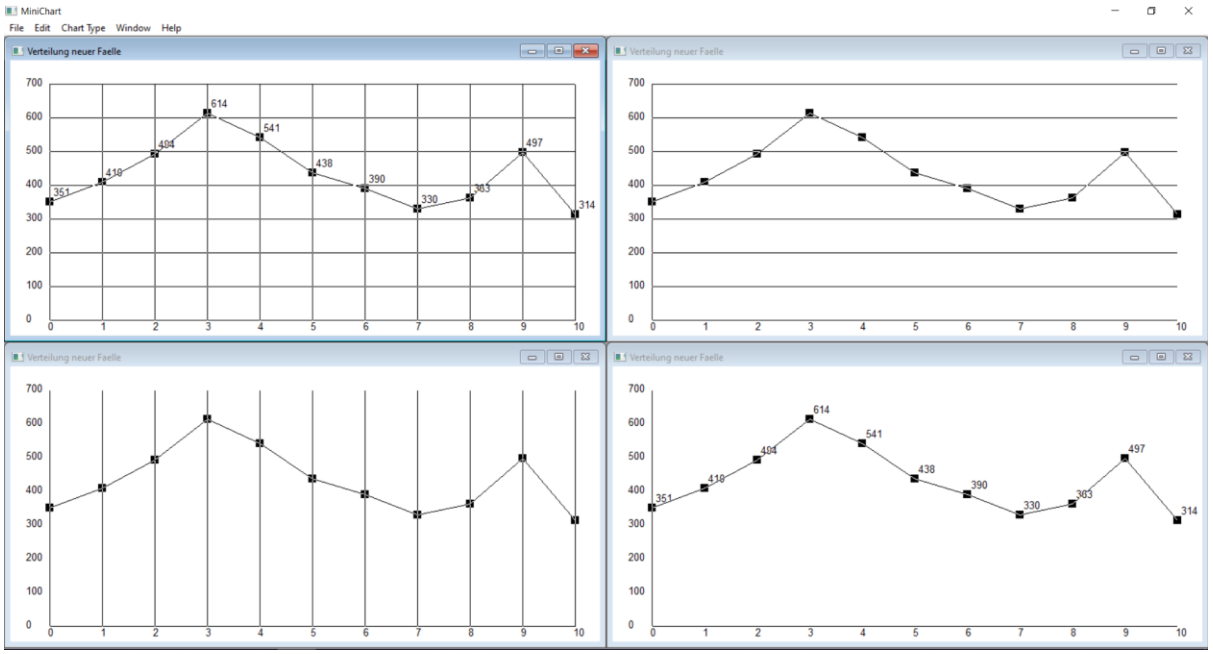
```
  dummy1.y := origin.y;

  for i := 0 to data^.Size - 1 do
  begin
    Str(i, intStr);
    DrawString(dummy1, intStr, 10);
    dummy1.x := dummy1.x + horizontalGap;
  end;

  dummy2.x := origin.x;
  dummy2.y := origin.y - verticalGap * maxVal;

  dummy1.x := origin.x;
  dummy1.y := origin.y;

  if (cType = GridLines) or (cType = HorizontalLines) then
    for i := 1 to data^.Size do
    begin
      dummy1.x := dummy1.x + horizontalGap;
      dummy2.x := dummy2.x + horizontalGap;
      DrawLine(dummy1, dummy2, 1);
    end;

  // draw values
  prev.x := 0;
  prev.y := 0;
  curr.x := origin.x;
  for i := 1 to data^.Size do
  begin
    currVal := MLInteger(data^.GetAt(i))^.AsInteger;
    curr.y := origin.y - Round((currVal / (maxVal * 100)) * maxVal *
verticalGap);

    dummy1.x := curr.x - 5;
    dummy1.y := curr.y - 5;
    DrawFilledRectangle(dummy1, 10, 10);

    if i <> 1 then
      DrawLine(prev, curr, 1);

    if (cType = Simple) or (cType = GridLines) then
    begin
      Str(currVal, intStr);
      dummy1.x := curr.x + 5;
      dummy1.y := curr.y - 20;
      DrawString(dummy1, intStr, 10);
    end;
    prev := curr;
```

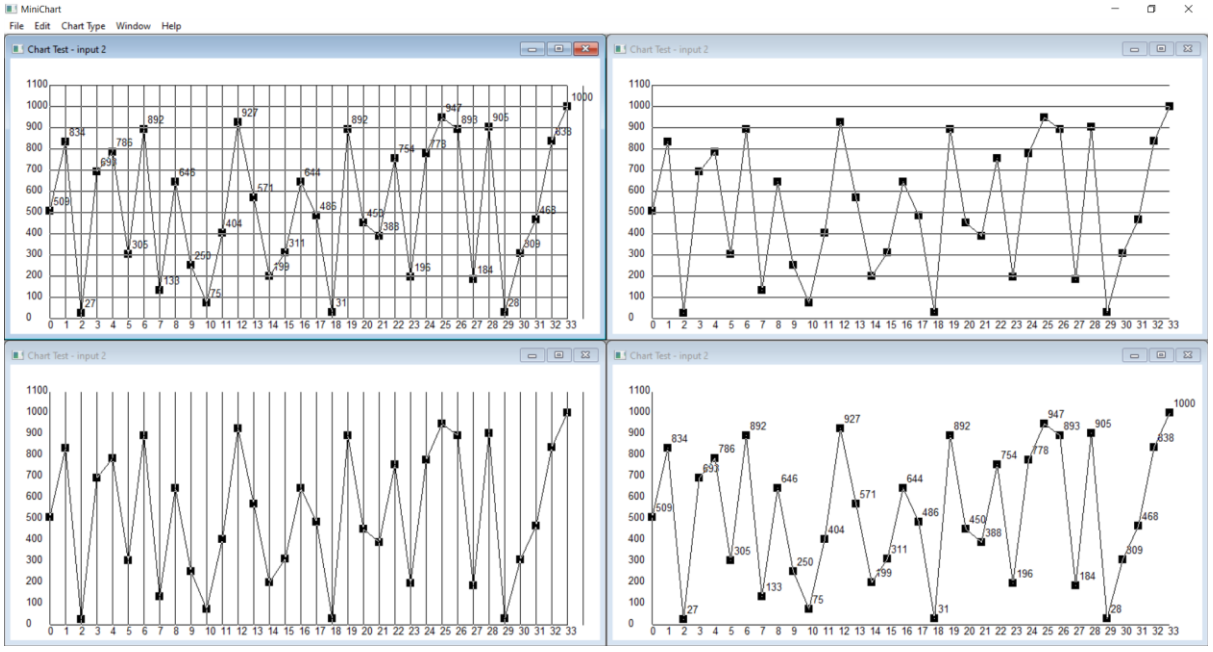
```
    curr.x := curr.x + horizontalGap;
  end;
end; (*ChartWindowObj.DrawChart*)

(*=== ChartApplication ===*)

function NewChartApplication(filename: string): ChartApplication;
var
  a: ChartApplication;
begin
  New(a, Init('MiniChart', filename));
  NewChartApplication := a;
end; (*NewChartApplication*)

constructor ChartApplicationObj.Init(name, filename: STRING);
begin
  inherited Init(name);
  Register('ChartApplication', 'MLApplication');
  ExtractDataFromFile(filename);
end; (*ChartApplicationObj.Init*)

destructor ChartApplicationObj.Done;
begin
  inherited Done;
end; (*ChartApplicationObj.Done*)

procedure ChartApplicationObj.OpenNewWindow;
begin
  NewChartWindow(title, data)^.Open;
end; (*ChartApplicationObj.OpenNewWindow*)

procedure ChartApplicationObj.BuildMenus;
begin
  (*chart types menu:*)
  simpleLinesCommand  := NewMenuCommand('Chart Type', 'Simple',  's');
  verticalLinesCommand := NewMenuCommand('Chart Type',  'Vertical', 'v');
  horizontalLinesCommand  := NewMenuCommand('Chart
Type',  'Horizontal',  'h');
  gridLinesCommand  := NewMenuCommand('Chart Type',  'Grid',  'g');
end; (*ChartApplicationObj.BuildMenus*)

procedure ChartApplicationObj.ExtractDataFromFile(filename: STRING);
var
  inFile: TEXT;
  line: string;
  value: integer;
begin
  assign(inFile, filename);
```

```
  reset(inFile);

  data := NewMLVector;

  if not eof(inFile) then
    readln(inFile, title);

  while not eof(inFile) do
  begin
    readln(inFile, line);
    Val(line, value);
    data^.Add(NewMLInt(value));
  end;
end; (*ChartApplicationObj.ExtractDataFromFile*)

(*=== main program ===*)

var
  a: ChartApplication;
  filename: string;

begin (*Chart1*)
  write('Enter input filename: '); ReadLn(filename);
  a := NewChartApplication(filename);
  a^.Run;
  Dispose(a, Done);
  WriteMetaInfo;
end. (*Chart1*)
```

Tests:







(input files are in the zip)