```
double CArea::Compute(long n, const double* x, const double* y, const CUIntArray* Outlier /*NULL*/)
{
	Init();

	m_X = x;
	m_Y = y;

	for (long i = 0; i < n; i++)
	{
		if (Outlier)
		{
			if (!Outlier->GetAt(i))
				m_Points.Add(i);
		}
		else
			m_Points.Add(i);
	}

	// Determine perimeter points of cluster
	GetPerimeter();

	return ComputeArea();
}

double CArea::Compute(const CUIntArray* specific, const double* x, const double* y, const CUIntArray*
Outlier /*NULL*/)
{
	Init();

	m_X = x;
	m_Y = y;

	long sz = specific->GetSize();
	for (long i = 0; i < sz; i++)
	{
		UINT Case = specific->GetAt(i);
		if (Outlier)
		{
			if (!Outlier->GetAt(Case))
```

```
                        m_Points.Add(Case);
                }
                else
                        m_Points.Add(Case);
        }

        // Determine perimeter points of cluster
        GetPerimeter();

        return ComputeArea();
}

////////////////////////////////////////////////////////////////////////////////
//
//      CArea:  Area calculation functions
//
////////////////////////////////////////////////////////////////////////////////

double CArea::ComputeArea()
{
        long nPerimPts = m_Perimeter.GetSize();
        double TotalArea = 0.0;

        // Determine origin
        double Origin = m_Y[m_Perimeter[0]];
        for (long i = 1; i < nPerimPts; i++)
                if (m_Y[m_Perimeter[i]] < Origin)
                        Origin = m_Y[m_Perimeter[i]];

        // Compute Area of cluster
        long pos1, pos2;
        double Area, yMin, yMax;
        for (i = 0; i < nPerimPts; i++)
        {
                pos1 = m_Perimeter[i];
                if (i + 1 == nPerimPts)
                        pos2 = m_Perimeter[0];
                else
                        pos2 = m_Perimeter[i + 1];
```

```
            yMin = min(m_Y[pos1], m_Y[pos2]);
            yMax = max(m_Y[pos1], m_Y[pos2]);

            Area = ComputeDistance(m_X[pos1], Origin, m_X[pos2], Origin) *
                     ComputeDistance(m_X[pos1], Origin, m_X[pos1], yMin) +
                    (ComputeDistance(m_X[pos1], yMin, m_X[pos2], yMin) *
                     ComputeDistance(m_X[pos2], yMin, m_X[pos2], yMax)) / 2 ;

            // If points are headed East, add distance.  If West, subtract distance.
            if (m_X[pos1] < m_X[pos2])
                  TotalArea += Area;
            else
                  TotalArea -= Area;
      }

      return TotalArea;
}

double CArea::ComputeDistance(double x1, double y1, double x2, double y2)
{
      // Compute linear euclidian distance
      double Distance = 0.0;
      double Radius = 3956.0; // Approximate radius of the earth in miles
      double pi = acos(-1.0);
      double radians = pi / 180.0;

      x1 *= radians;
      x2 *= radians;
      y1 *= radians;
      y2 *= radians;

      if (x1 != x2 || y1 != y2)
      {
            Distance = pow(sin((y2 - y1) / 2.0), 2.0) + cos(y1) * cos(y2) * pow(sin((x2 - x1) / 2.0), 2.0);
            Distance = (2 * asin(min(1, pow(Distance, 0.5)))) * Radius;
      }

      return Distance;
}
```

```
////////////////////////////////////////////////////////////////////////////
//
//    CArea:  Perimeter functions
//
////////////////////////////////////////////////////////////////////////////

double CArea::ComputeSlope(long Point1, long Point2, long Direction)
{
      double slope = 0.0;

      if (m_Y[Point1] == m_Y[Point2])
            slope = 0.0;
      else if (m_X[Point1] == m_X[Point2])
            slope = (m_Y[Point2] < m_Y[Point1] || Direction) ? MinSlope : MaxSlope;
      else
            slope = (m_Y[Point2] - m_Y[Point1]) / (m_X[Point2] - m_X[Point1]);

      return slope;
}

long CArea::GetNextPoint(long CurPoint, long Direction, BOOL &LastXSloped)
{
      long NextPoint = -1;
      double ExtremeSlope = 0.0;
      BOOL ExtremeSet = FALSE;
      double CurSlope;
      long sz = m_Points.GetSize();

      // Loop to find greatest slope from current point
      for (long i = 0; i < sz; i++)
      {
            long TestPoint = m_Points[i];
            if (TestPoint == CurPoint)
                  continue;

            if (m_X[TestPoint] == m_X[CurPoint] && m_Y[TestPoint] == m_Y[CurPoint])
                  continue;

            if (!Direction)
            {
```

```
            if (m_X[TestPoint] < m_X[CurPoint])
                continue;
        }
        else
        {
            if (m_X[TestPoint] > m_X[CurPoint])
                continue;
        }

        CurSlope = ComputeSlope(CurPoint, TestPoint, Direction);
        if (!ExtremeSet || CurSlope > ExtremeSlope)
        {
            if (!LastXSloped || (CurSlope != 0 && CurSlope != MaxSlope && CurSlope != MinSlope))
            {
                ExtremeSet = TRUE;
                ExtremeSlope = CurSlope;
                NextPoint = TestPoint;
            }
        }
    }

    LastXSloped = FALSE;

    // If next point is found, check for horizontal and vertical
    if (NextPoint > -1)
    {
        CurSlope = ComputeSlope(CurPoint, NextPoint, Direction);

        if (!CurSlope)
        {
            // Vertical line
            LastXSloped = TRUE;
            for (long i = 0; i < sz; i++)
            {
                if (m_Points[i] == CurPoint || m_Points[i] == NextPoint)
                    continue;

                if (m_Y[NextPoint] == m_Y[m_Points[i]])
                {
                    if (m_X[NextPoint] > m_X[CurPoint])
```

```
                        {
                              if (m_X[m_Points[i]] > m_X[NextPoint])
                                    NextPoint = m_Points[i];
                        }
                        else
                        {
                              if (m_X[m_Points[i]] < m_X[NextPoint])
                                    NextPoint = m_Points[i];
                        }
                  }
            }
      }
      else if (CurSlope == MinSlope || CurSlope == MaxSlope)
      {
            // Horizontal lines
            LastXSloped = TRUE;
            for (long i = 0; i < sz; i++)
            {
                  if (m_Points[i] == CurPoint || m_Points[i] == NextPoint)
                        continue;

                  if (m_X[NextPoint] == m_X[m_Points[i]])
                  {
                        if (m_Y[NextPoint] > m_Y[CurPoint])
                        {
                              if (m_Y[m_Points[i]] > m_Y[NextPoint])
                                    NextPoint = m_Points[i];
                        }
                        else
                        {
                              if (m_Y[m_Points[i]] < m_Y[NextPoint])
                                    NextPoint = m_Points[i];
                        }
                  }
            }
      }
}

return NextPoint;
}
```

```
void CArea::GetPerimeter()
{
	BOOL LastXSloped = FALSE;
	long sz = m_Points.GetSize();

	long WestMost = m_Points[0];
	for (long i = 1; i < sz; i++)
	{
		if (m_X[m_Points[i]] < m_X[WestMost])
			WestMost = m_Points[i];
		else if (m_X[m_Points[i]] == m_X[WestMost] && m_Y[m_Points[i]] < m_Y[WestMost])
			WestMost = m_Points[i];
	}

	m_Perimeter.Add(WestMost);
	long NextPoint;
	long CurPoint = WestMost;

	long Direction = 0;
	for (i = 0; i < sz * 2; i++)
	{
		NextPoint = GetNextPoint(CurPoint, Direction, LastXSloped);

		if (NextPoint < 0)
		{
			++Direction;	// just a duplicate point, not end of search (possible???)
			continue;
		}
		else if (m_X[CurPoint] == m_X[NextPoint] && m_Y[CurPoint] > m_Y[NextPoint])
		{
			++Direction;
		}

		if (NextPoint == WestMost)
			break;

		CurPoint = NextPoint;
		m_Perimeter.Add(NextPoint);
	}
```

```
}


/////////////////////////////////////////////////////////////////////////////
//
//    CArea:  Attribute functions
//
/////////////////////////////////////////////////////////////////////////////

void CArea::CopyPerimeter(CUIntArray& P)
{
      long i, sz;
      sz = m_Perimeter.GetSize();
      P.SetSize(sz);
      for (i = 0; i < sz; i++)
            P[i] = m_Perimeter[i];
}


// Perim.h : definition of the CArea class

#ifndef PERIM__H
#define PERIM__H

#include "Global.h"

class CClustGrid;

class CPerim : public CObject
{
// Construction
public:
      CPerim();

public:
      void Compute(CClustGrid* pGrid, CUIntArray& nPerim);

protected:
      void ConvexHull(CPointArray& pXY, CUIntArray& nPerim);
```

```
	void PointsOnGrid(CClustGrid* pGrid, CPointArray& pXY);
	void RecordsOnPerimiter(CClustGrid* pGrid, CPointArray& pXY, CUIntArray& nPerimXY, CUIntArray&
nPerimPts);
	void BendPerimiter(CClustGrid* pGrid, CPointArray& pXY, CUIntArray& nPerimXY);

	double Dot(Point& p1, Point& p2);
	double PointToLine(Point p, Line line, Point& pline, double& pctline);
	double Distance(Point& pt1, Point& pt2);
};

#endif


// Perim.cpp : implementation of the CPerim class
//


#include "stdafx.h"
#include "Perimeter.h"

#include "2dch.h"
#include "GrdClust.h"

#include <math.h>

/////////////////////////////////////////////////
CPerim::CPerim()
{
}

void CPerim::Compute(CClustGrid* pGrid, CUIntArray& nPerimPts)
{
	// Declare
	CPointArray pXY;
	CUIntArray nPerimXY;

	// Set array points based on grid.  Points to be considered for convex hull.
	PointsOnGrid(pGrid, pXY);

	// Determine convex hull
```

```
	ConvexHull(pXY, nPerimXY);

	// Bend perimiter lines
	//BendPerimiter(pGrid, pXY, nPerimXY);

	// Add all points that are on the perimiter to UINT array.
	RecordsOnPerimiter(pGrid, pXY, nPerimXY, nPerimPts);

	return;
}

void CPerim::PointsOnGrid(CClustGrid* pGrid, CPointArray& pXY)
{
	Point pt;
	unsigned int nCols = pGrid->GetNCols();
	unsigned int nRows = pGrid->GetNRows();

	// Add all occcupied cell coordinates to Point array
	for (unsigned int i = 0; i < nCols; i++)
		for (unsigned int j = 0; j < nRows; j++)
			if (pGrid->GetValue(pGrid->GetIndex(i,j)) != 0)
			{
				pt.x = i;
				pt.y = j;
				pXY.Add(pt);
			}

	return;
}

void CPerim::ConvexHull(CPointArray& pXY, CUIntArray& nPerim)
{
	// Calculate convex hull (perimiter)
	C2dch* pCH = new C2dch;
	pCH->Compute(&pXY, nPerim);
	delete pCH;
	return;
}
```

```
void CPerim::RecordsOnPerimiter(CClustGrid* pGrid, CPointArray& pXY, CUIntArray& nPerimXY, CUIntArray&
nPerimPts)
{
	// Add all perimeter record numbers to array.
	unsigned int sz = nPerimXY.GetSize();
	nPerimPts.SetSize(sz);
	for (unsigned int i = 0; i < sz; i++)
	{
		unsigned int x = (unsigned int)pXY.GetAt(nPerimXY[i]).x;
		unsigned int y = (unsigned int)pXY.GetAt(nPerimXY[i]).y;
		nPerimPts[i] = pGrid->GetValue(pGrid->GetIndex(x, y));
	}
	return;
}

void CPerim::BendPerimiter(CClustGrid* pGrid, CPointArray& pXY, CUIntArray& nPerimXY)
{
	unsigned long nPoints = (unsigned long)pXY.GetSize();
	unsigned long nPerimPts = (unsigned long)nPerimXY.GetSize();
	unsigned long nNotPerimPts = nPoints - nPerimPts;

	// Assign points to perimeter or inside perimeter arrays for optimizing perimiter
	Point pt1, pt2;
	CUIntArray nNotPerimXY;
	nNotPerimXY.SetSize(nNotPerimPts);

	for (unsigned long i = 0; i < nPerimPts - 1; i++)
	{
		pt1 = pXY[nPerimXY[i]];
		pt2 = pXY[nPerimXY[i + 1]];
		double LineLen = Distance(pt1, pt2);

		unsigned long BegLook = (unsigned long) ((LineLen * 0.1) + 0.5);
		unsigned long EndLook = (unsigned long) ((LineLen * 0.5) + 0.5);


	}

	return;
}
```

```
double CPerim::PointToLine(Point p, Line line, Point& pline, double& pctline)
{
      //-----------------------------------------------------------------------
      // Returns the distance of a point to a line.
      //   Calculates the closest x,y point on the line.
      //   Calculates the percentage up the line the closest x,y point occupies.
      //-----------------------------------------------------------------------

      double dist = -1.0;
      Point diff = { p.x - line.pt1.x, p.y - line.pt1.y};

      pctline = Dot(line.pt2, diff) / Dot(line.pt2, line.pt2);
      if (pctline < 0.0 || pctline > 1.0)
            return dist;

      diff.x -= pctline * line.pt2.x;
      diff.y -= pctline * line.pt2.y;

      pline.x = line.pt1.x + pctline * line.pt2.x;
      pline.y = line.pt1.y + pctline * line.pt2.y;
      dist = sqrt(Dot(diff,diff));
      return dist;
}

double CPerim::Dot( Point& p1,  Point& p2)
{
      return p1.x * p2.x + p1.y * p2.y;
}

double CPerim::Distance(Point& pt1, Point& pt2)
{
      double x1 = pt1.x;
      double y1 = pt1.y;
      double x2 = pt2.x;
      double y2 = pt2.y;
    double Radius = 3956.0;                          // approx earth radius in miles
    double Radians = acos(-1.0) / 180.0;  // pi / 180
      double Distance = 0.0;
```

```
	if (x1 != x2 || y1 != y2)
	{
		x1 *= Radians;
		x2 *= Radians;
		y1 *= Radians;
		y2 *= Radians;

		Distance = pow(sin((y2-y1) / 2.0), 2.0) + cos(y1) * cos(y2) * pow(sin((x2-x1) / 2.0), 2.0);
		Distance = (2 * asin(min(1, pow(Distance, .5)))) * Radius;
	}

	return Distance;
}

// Data.h : header file
//

#ifndef DATA__H
#define DATA__H

/////////////////////////////////////////////////////////////////////////////
// CData dialog

class CData : public CObject
{
// Construction
public:
	CData();
	~CData();

// Attributes
protected:
	// Data variables
	double* m_Long;
	double* m_Lat;
	CUIntArray m_nWt;		// Array of number of HH at same location
	CStringArray m_BG;		// Block groups
	CStringArray m_Type;		// Location type (Residential/Business)

	unsigned int m_nCases;		// Number of cases
```

```
	unsigned int m_nCasesWt;	// Number of cases weighted
	unsigned int m_nCurr;		// Number of block groups read

	BOOL m_bIsOpen;				// Flag to indicate open data file and Long/Lat allocations
	BOOL m_bEOF;				// Flag to indicate end of file
	CStdioFile* m_File;			// Data file pointer
	CStdioFile* m_TmpFile;		// Records read for split group
	CString m_Buffer;		// Read buffer
	CString m_LastRead;			// Last record read

	CString m_ListFile;			// List file name
	CStringArray m_List;		// Array of splits to process

// Functions
protected:
	// List routines
	void InitList(LPCSTR listName);
	BOOL ReadList(LPCSTR listName);

	// Data routines
	void Init();
	void InitOpen();
	long GetLongField();
	double GetDblField();
	CString GetStrField();

public:
	BOOL Open(LPCSTR, LPCSTR);
	BOOL Read();
	BOOL Close();

	// Attribute routines
	double* GetLongitude();
	double* GetLatitude();
	unsigned int GetNCases();
	unsigned int GetNCasesWt();
	unsigned int GetWt(UINT index);
	unsigned int GetNDone();
	BOOL GetEOF();
	CString GetGroup(UINT nGrp);
```

```
	CString GetBlockGroup(UINT nCase);
	CString GetType(UINT nCase);

	CString GetTempFileName();
};

#endif

// Data.cpp : implementation file

#include "stdafx.h"
#include "Data.h"

CData::CData()
{
	m_Long = new double[MAX_CASES];
	m_Lat = new double[MAX_CASES];
	m_File = new CStdioFile();
	m_TmpFile = new CStdioFile();

	m_bIsOpen = FALSE;
	InitOpen();
}

CData::~CData()
{
	if (m_Long)
		delete []m_Long;

	if (m_Lat)
		delete []m_Lat;

	if (m_File)
		delete m_File;

	if (m_TmpFile)
	{
		CFile::Remove(m_TmpFile->GetFilePath());
		delete m_TmpFile;
	}
```

```
}

/////////////////////////////////////////////////////////////////////////////
// CData:  Specific List

void CData::InitList(LPCSTR list)
{
	m_List.RemoveAll();
	m_ListFile = list;
}

BOOL CData::ReadList(LPCSTR list)
{
	InitList(list);

	// If specific split groups, open list file and read split groups.
	if (m_ListFile != "ALL" && m_ListFile != "NONE")
	{
		// Open and read
		CStdioFile* pf;
		try
		{
			pf = new CStdioFile(list, CFile::modeRead);
		}
		catch (CFileException* e)
		{
			AfxMessageBox("ERROR: Could not open the split file list that was specified.");
			e->Delete();
			return FALSE;
		}

		// Scan split groups to be processed
		CString str;
		while (pf->ReadString(str))
			m_List.Add(str.Left(BGNUMLEN));

		// Close and return
		pf->Close();
		delete pf;
	}
```

```
	return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CData:  Data File

void CData::Init()
{
	m_nCases = 0;
	m_nCasesWt = 0;
	m_nWt.RemoveAll();
	m_BG.RemoveAll();
	m_Type.RemoveAll();

	if (!m_Long || !m_Lat)
	{
		AfxMessageBox("ERROR:  Not enough memory to read files.");
		return;
	}

	for (unsigned int i = 0; i < MAX_CASES; i++)
	{
		m_Long[i] = 0.0;
		m_Lat[i] = 0.0;
	}
}

void CData::InitOpen()
{
	// Close any files left open
	Close();

	// Initialize and allocate members
	m_nCurr = 0;
	m_LastRead = "";
	m_bEOF = FALSE;

	Init();
}
```

```
BOOL CData::Open(LPCSTR dataName, LPCSTR listName)
{
	// Initialize one-time-only members
	InitOpen();

	// Read split list if specified
	if (!ReadList(listName))
		return FALSE;

	// Open data file
	if (!m_File->Open(dataName, CFile::modeRead))
	{
		AfxMessageBox("ERROR: The attempt to open the data file has failed.");
		return FALSE;
	}

	m_bIsOpen = TRUE;
	return TRUE;
}

BOOL CData::Read()
{
	Init();

	// Declare
	unsigned int index = 0;
	CString Split = "";
	BOOL bBegin = FALSE;
	BOOL bEnd = FALSE;

	// Open temporary file
	m_TmpFile->Open(GetTempFileName(), CFile::modeCreate | CFile::modeWrite);

	// Set Split List if reading all Split
	if (m_ListFile == "NONE")
	{
		m_List.Add("None");

		// Read data
```

```
        while(!m_bEOF)
        {
                if (m_File->ReadString(m_Buffer))
                {
                        // Assign values to  member variables
                        Split = (GetStrField()).Left(BGNUMLEN);
                        m_Long[index] = GetDblField();
                        m_Lat[index++] = GetDblField();
                        m_nWt.Add(GetLongField());
                        m_BG.Add(GetStrField());
                        m_Type.Add(GetStrField());

                        // Add original buffer to array
                        m_TmpFile->WriteString(m_Buffer);
                        m_TmpFile->WriteString(" \n");

                        // Increment number of cases/weighted cases
                        m_nCasesWt += m_nWt[m_nCases++];
                        if (m_nCases == MAX_CASES)
                                m_bEOF = TRUE;
                }
                else
                        m_bEOF = TRUE;
        }
}
else
{
        m_Buffer = m_LastRead;
        Split = (GetStrField()).Left(BGNUMLEN);

        if (m_ListFile == "ALL")
                if (m_nCurr)
                        m_List.Add(Split);
                else
                        m_List.Add("SET FIRST");

        // If the last record read was part of the next split group, set first record
        if (Split == m_List[m_nCurr])
        {
                bBegin = TRUE;
```

```
        m_Long[index] = GetDblField();
        m_Lat[index++] = GetDblField();
        m_nWt.Add(GetLongField());
        m_BG.Add(GetStrField());
        m_Type.Add(GetStrField());

        m_TmpFile->WriteString(m_LastRead);
        m_TmpFile->WriteString(" \n");

        m_nCasesWt += m_nWt[m_nCases++];
}

// If the last record read was greater than the next split group, skip read
if (Split > m_List[m_nCurr])
        bEnd = TRUE;

while(!bEnd)
{
        // Read data
        if (!m_File->ReadString(m_Buffer))
                m_bEOF = TRUE;

        if (m_bEOF)
                break;

        m_LastRead = m_Buffer;

        // Get split and set first split for ALL
        Split = (GetStrField()).Left(BGNUMLEN);
        if (m_List[m_nCurr] == "SET FIRST")
                m_List[m_nCurr] = Split;

        // If correct split group, set flags
        if (Split == m_List[m_nCurr])
                bBegin = TRUE;
        else if (Split > m_List[m_nCurr])
        {
                bBegin = FALSE;
                bEnd = TRUE;
```

```
                }

                // Fill in data arrays
                if (bBegin)
                {
                        m_Long[index] = GetDblField();
                        m_Lat[index++] = GetDblField();
                        m_nWt.Add(GetLongField());
                        m_BG.Add(GetStrField());
                        m_Type.Add(GetStrField());

                        m_TmpFile->WriteString(m_LastRead);
                        m_TmpFile->WriteString(" \n");

                        m_nCasesWt += m_nWt[m_nCases++];
                        if (m_nCases == MAX_CASES)
                        {
                                m_nCasesWt -= m_nWt[m_nCases - 1];
                                m_nCases--;
                                m_bEOF = TRUE;
                        }
                }
        }
    }
    m_nWt.FreeExtra();
    m_BG.FreeExtra();
    m_Type.FreeExtra();

    // Increment number of groups read
    m_nCurr++;

    // Set end of Split list if list is specified and has been exhausted
    if (!m_bEOF && m_ListFile != "ALL" && m_nCurr == (unsigned)m_List.GetSize())
        m_bEOF = TRUE;

    m_TmpFile->Close();
    return m_bEOF;
}

BOOL CData::Close()
```

```
{
	// Close the data file and reset
	if (m_File && m_bIsOpen)
	{
		m_File->Close();
		m_bIsOpen = FALSE;
		return TRUE;
	}
	return FALSE;
}

CString CData::GetStrField()
{
	CString str;

	m_Buffer.TrimLeft();
	int pos = m_Buffer.FindOneOf(" \t,");
	if (pos > -1)
	{
		if (pos > 0)
			str = m_Buffer.Left(pos);
		m_Buffer = m_Buffer.Mid(pos + 1);
	}
	else
	{
		str = m_Buffer;
		m_Buffer = "";
	}

	return str;
}

long CData::GetLongField()
{
	if (m_Buffer == "")
		return -1;

	return atol(GetStrField());
}
```

```
double CData::GetDblField()
{
      if (m_Buffer == "")
            return -1.0;

      return atof(GetStrField());
}

////////////////////////////////////////////////////////////////////////////
//
// CData:  Attribute funcitons
//
////////////////////////////////////////////////////////////////////////////

double* CData::GetLongitude()
{
      return m_Long;
}

double* CData::GetLatitude()
{
      return m_Lat;
}

unsigned int CData::GetNCases()
{
      return m_nCases;
}

unsigned int CData::GetNCasesWt()
{
      return m_nCasesWt;
}

unsigned int CData::GetWt(UINT index)
{
      return m_nWt[index];
}

unsigned int CData::GetNDone()
```

```
{
	return m_nCurr;
}

BOOL CData::GetEOF()
{
	return m_bEOF;
}

CString CData::GetGroup(UINT nGrp)
{
	return m_List[nGrp - 1];
}

CString CData::GetTempFileName()
{
	CString fName = m_File->GetFilePath();
	fName += ".TEMP";
	return fName;
}

CString CData::GetBlockGroup(UINT nCase)
{
	return m_BG[nCase];
}

CString CData::GetType(UINT nCase)
{
	return m_Type[nCase];
}


#ifndef GRDCLUST__H
#define GRDCLUST__H

#include "Global.h"
class CData;

class CClustGrid
```

```
{
public:
      CClustGrid();
      ~CClustGrid();

      CClustGrid(double MinLong, double MinLat, double MaxLong, double MaxLat, long feet, long nPoints,
double *Longs, double *Lats);
      void Initialize(double MinLong, double MinLat, double MaxLong, double MaxLat, long feet, long nPoints,
double *Longs, double *Lats, CULongArray* pPtMask = NULL);

      // Get attribute funcitons
      long GetDistance() {return m_dFeet;}
      long GetValue(long pos) {return m_Grid[pos];}
      long GetSize() {return m_nCols * m_nRows;}

      unsigned int GetNCols() {return m_nCols;}
      unsigned int GetNRows() {return m_nRows;}
      unsigned int GetPRow(long pos) {return m_pRow[pos];}
      unsigned int GetPCol(long pos) {return m_pCol[pos];}
      unsigned int GetIndex(unsigned int x,unsigned int y) {return x + y * m_nCols;}

      long GetDistance(long Cell1, long Cell2);
      long GetRLDistance(long Cell1, long Cell2);

      // Set attribute funcitons
      void SetValue(long pos, long value) {m_Grid[pos] = value;}

private:
      long m_dFeet;
      double m_Height;
      double m_Width;

      long *m_Grid;
      unsigned int m_nRows;
      unsigned int m_nCols;

      long m_nPoints;
      unsigned int *m_pRow;
      unsigned int *m_pCol;
};
```