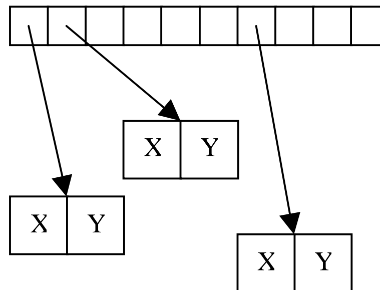
Vorbemerkung

Sämtliche Quellcodes erstelle ich mit Visual C++, ich versuche aber, mit ANSI-C++ auszukommen.
Die Variablen folgen der Ungarischen Notation von Charles Simonyi (Microsoft).

Aufgabe M2.1

Die Koordinaten verwalte ich im Hauptspeicher, indem ich ein Array benutze, dessen Elemente Zeiger auf eine Struktur sind, die jeweils einen Punkt (mit X- und Y-Koordinate) aufnehmen kann.
Grafisch kann man sich diesen Zusammenhang wie folgt veranschaulichen:



Wenn ein Zeiger `NULL` ist, so existieren für ihn keine zugehörigen Koordinaten. Im Bild ist dies bei den Elementen 2 bis 5 und 7 bis 9 der Fall (die Zählung beginnt links bei 0).

Normalerweise würde ich in C für die Speicherung der Koordinaten ein `struct` verwenden. Dies macht aber den Vergleich zweier Koordinaten recht umständlich und fehleranfällig, so dass ich mich für eine Klasse entschied. Diese enthält zwei Konstruktoren zur einfachen Erschaffung eines Punktes und eine boolesche Funktion `Equals`, die obiges Vergleichsproblem elegant löst.

Im folgenden Quelltext habe ich die Zeilen, die der Deklaration der Datenstruktur dienen, blau markiert, die Funktion `Clear` ist dagegen rot.

```
////////////////////////////////////////////////////////////
// Softwarebauelemente I, Aufgabe M2.1
//
// author:             Stephan Brumme
// last changes:       October 18, 2000


// we need const NULL
#include <stdio.h>


// simple class for storing a single point
// two constructors to ease creation
// contains Equals method which compares the coordinates of two points
class CPoint
{
public:
	// default constructor, zero the member variables
	CPoint()
	{	m_nX = 0; m_nY = 0;	}

	// easier creation of a point
	CPoint(int X, int Y)
	{	m_nX = X; m_nY = Y;	}

	// public member variables
	int m_nX, m_nY;

	// compares two points
	bool Equals(CPoint Point)
	{	return ((m_nX == Point.m_nX) && (m_nY == Point.m_nY));	}
};
```

```
// class for storing a set of points in an array
// each entry is implemented as a pointer to CPoint (see above)
class CSetOfPoints
{
public:
        // default constructor
        CSetOfPoints();

        // clear the set
        void Clear();
        // insert a point to the set, returns true on success
        bool Insert(CPoint Point);
        // remove a point from the set, returns true on success
        bool Scratch(CPoint Point);
        // determine whether a point is already in the set
        bool Contains(CPoint Point);

        // max. number of stored points
        enum { MAX_POINTS = 10 };

protected:
        // set all pointers to NULL
        void ZeroMem();
        // get array position of a specified point, -1 on error
        int  GetPos(CPoint Point);

        // storage
        CPoint* arPPoints[MAX_POINTS];
};


// default constructor
CSetOfPoints::CSetOfPoints()
{
        // memory is randomly set on startup
        // initialize all array entry by setting them to NULL
        ZeroMem();
}


// clear the set
void CSetOfPoints::Clear()
{
        // delete each entry that is non-NULL
        for (int nLoop=0; nLoop<MAX_POINTS; nLoop++)
                if (arPPoints[nLoop] != NULL)
                        delete arPPoints[nLoop];

        // set all array entries to NULL
        ZeroMem();
}


// insert a point to the set, returns true on success
bool CSetOfPoints::Insert(CPoint Point)
{
        // if point is already part of the set we don't need to insert it
        if (Contains(Point))
                return true;

        // find a free entry
        for (int nLoop=0; nLoop<MAX_POINTS; nLoop++)
                // free entry found ?
                if (arPPoints[nLoop] == NULL)
                {
                        // create a new CPoint on heap
                        arPPoints[nLoop] = new CPoint(Point.m_nX, Point.m_nY);
                        // we are done
                        return true;
                }

        // no free entry found !
        return false;
}
```

```
// remove a point from the set, returns true on success
bool CSetOfPoints::Scratch(CPoint Point)
{
	// get position of this point
	int nPos = GetPos(Point);
	// if GetPos returned -1 the point is not part of the set
	if (nPos == -1)
		return false;

	// free the memory
	delete arPPoints[nPos];
	// set pointer to NULL
	arPPoints[nPos] = NULL;

	// we are successfully done
	return true;
}


// determine whether a point is already in the set
bool CSetOfPoints::Contains(CPoint Point)
{
	// position only equals -1 if the point is not part of the set
	return (GetPos(Point) != -1);
}


// set all pointers to NULL
void CSetOfPoints::ZeroMem()
{
	for (int nLoop=0; nLoop<MAX_POINTS; nLoop++)
		arPPoints[nLoop] = NULL;
}


// get array position of a specified point, -1 on error
int CSetOfPoints::GetPos(CPoint Point)
{
	// look at each entry
	for (int nLoop=0; nLoop<MAX_POINTS; nLoop++)
		// is this entry valid ?
		if (arPPoints[nLoop] != NULL)
			// compare the values
			if (arPPoints[nLoop]->Equals(Point))
				// point found, return current position
				return nLoop;

	// point not found
	return -1;
}


// main function
void main()
{
	// construct a set
	CSetOfPoints MySet;

	// do some operations to verify functionality
	MySet.Insert(CPoint(2,4));
	MySet.Insert(CPoint(3,5));
	MySet.Insert(CPoint(7,9));

	MySet.Scratch(CPoint(3,5));
	MySet.Insert(CPoint(3,6));

	bool bContains;
	bContains = MySet.Contains(CPoint(2,5));
	bContains = MySet.Contains(CPoint(2,4));

	MySet.Clear();
}
```