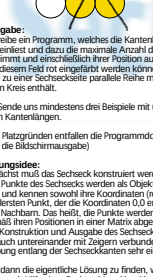
Musteraufgabe:

In einem sechseckigen Feld von dicht gepackten Kreisen sollen möglichst viele Kreise gefärbt werden, allerdings so, daß auf jeder zu einer Sechsseitseite parallelen Reihe höchstens ein Kreis eingefärbt ist.

Die Abbildung zeigt ein Feld mit Kantenlänge 3 und einem rot eingefärbten Kreis. Die gelben Kreise geben die Reihen an, auf der die rote Kreis ausstrahlt. In diesen Reihen können also keine weiteren Kreise gefärbt werden.



Aufgabe:
Schreibe ein Programm, welches die Kantenlänge des Feldes einliest und dazu die maximale Anzahl der Kreise bestimmt und einschließlich ihrer Position ausgibt, die auf diesem Feld rot eingefärbt werden können, ohne daß eine zu einer Sechsseitseite parallele Reihe mehr als einen roten Kreis enthält.

► Sende uns mindestens drei Beispiele mit unterschiedlichen Kantenlängen.

(aus Platzgründen entfallen die Programmdokumentation und die Bildschirmausgabe)

Lösungsidee:
Zunächst muß das Sechseck konstruiert werden; die einzelnen Punkte des Sechsecks werden als Objekte implementiert und kennen sowohl ihre Koordinaten (relativ zum mittleren Punkt, der die Koordinaten 0,0 erhält) als auch ihre Nachbarn. Das heißt, die Punkte werden sowohl gemäß ihren Positionen in einer Matrix abgelegt, was bei der Konstruktion und Ausgabe des Sechsecks nützlich ist, als auch untereinander mit Zeigern verbunden, was die Färbung entlang der Sechseckkanten sehr einfach macht.

Um dann die eigentliche Lösung zu finden, werden die Steine mit Hilfe eines Backtracking-Algorithmus auf freie Felder plaziert. Das Programm beginnt mit der Plazierung eines Steines auf dem Feld 0,0. Nach der Plazierung werden die Felder in der gleichen Reihe und den Diagonalen markiert (gefärbt). Nun wird versucht, auf gleiche Weise weitere Steine zu setzen. Voraussetzung dafür ist, daß mindestens ein unmarkiertes Feld vorhanden ist. Ist dies nicht der Fall, werden der zuletzt gesetzte Stein und die durch ihn verursachten Markierungen entfernt und der Stein, wenn möglich, anders plaziert.
Das Programm numeriert die Felder bei ihrer Erzeugung durch und verwendet diese Nummern im weiteren Verlauf bei der Reihenfolge, in der Felder besetzt werden. Prinzipiell ist die die Reihenfolge, in der die Felder besetzt werden, aber beliebig, sie hat keinen Einfluß auf die maximal setzbare Anzahl Steine.
Die Steine werden auf die Felder in der Reihenfolge wachsender Nummern gesetzt. Dies reicht aus, alle übrigen möglichen Reihenfolgen (absteigend, alternierend) werden nicht ausprobiert.
Das Programm bricht ab, wenn es alle möglichen Positionierungen geprüft oder 2 * Anzahl Kantenlänge - 1 Steine plaziert hat. (Das ist die maximal mögliche Zahl.)

(Alternative Lösungsidee: Die Felder werden nicht explizit markiert, sondern man prüft beim Setzen eines Steins, ob in den entsprechenden Zeilen und Diagonalen bereits Steine gesetzt sind.)

Halbformale Programmbeschreibung:
Lies die Kantenlänge ein
Erzeuge ein mittleren Punkt des Sechsecks
    {rekursiver Aufruf: Wenn ein Punkt erzeugt
        wird, werden auch alle neuen
        Nachbarpunkte, die noch im Sechseck
        liegen, erzeugt}

wiederhole:
    Ist die Anzahl der gesetzten Steine größer als
    die bisher beste Lösung?
        Wenn ja, übernimm dies als neue Lösung
        und gib sie aus
        { nicht unbedingt erforderlich }
    Für alle Felder:
    { in der vorgegebenen Reihenfolge }
        Könnte auf dieses Feld ein Stein gesetzt
        werden?
            Wenn ja:
                Setze den Stein, markiere die Zeile
                und Diagonalen
                Rufe die Schleife { wiederhole }
                rekursiv auf
                Entferne den Stein und die von ihm
                erzeugten Markierungen
bis alle Kombinationen getestet oder das
Maximum erreicht wurde.

Programmtext: (zur Abwechslung in C++)
// Sechseck.cpp

```
#include <stdio.h>
#include <stdlib.h>

class Sechseck;

class Punkt
{
  protected:
  int     iX, iY, iTyp, iZaehler;
  Punkt   *Rechts, *Links, *RechtsOben,
    *LinksUnten, *RechtsUnten, *LinksOben;

  void SetzeTyp(int iNeuerTyp);
  void EntferneTyp(int iNeuerTyp)

  public:
  Punkt(int X,int Y);

  void BildeVerbindungen(Sechseck *Basis);
  void SetzePunkt();
  void EntfernePunkt();
  int  LieferTyp();
}
typedef Punkt * PunktZeiger;

class Sechseck
{
  protected:
  PunktZeiger* Punkte;
  PunktZeiger* PunktListe;
  int   iKantenLaenge, iBreite, iNrPunkte;

  public:
  Sechseck(int iKante);
  ~Sechseck();
  int   NrPunkte();
  Punkt* PunktNr(int iNr);
  Punkt* LieferePunkt(int iX,int iY);
  Punkt* LiefereOderErzeugePunkt(int iX,int iY);
}


Punkt::Punkt(int X,int Y)
{iX = X; iY = Y; iTyp = 0; iZaehler = 0;}

void Punkt::BildeVerbindungen (Sechseck *Basis)
{
  Rechts = Basis->LiefereOderErzeugePunkt(iX+2,iY);
  if (Rechts != NULL) Rechts->Links = this;
  Links = Basis->LiefereOderErzeugePunkt(iX-2,iY);
  if (Links != NULL)   Links->Rechts = this;
  RechtsOben = Basis->LiefereOderErzeugePunkt(iX+1,iY+1);
  if (RechtsOben != NULL) LinksUnten->RechtsOben = this;
  RechtsUnten = Basis->LiefereOderErzeugePunkt(iX+1,iY-1);
  if (LinksUnten != NULL) LinksUnten->RechtsOben = this;
  LinksOben = Basis->LiefereOderErzeugePunkt(iX-1,iY+1);
  if (LinksOben != NULL) LinksOben->RechtsUnten = this;
}

void Punkt::SetzeTyp(int iNeuerTyp)
{
  if (iTyp == iNeuerTyp) iZaehler++;
  else {
      iTyp = iNeuerTyp;
      iZaehler = 1;
    }
}

void Punkt::EntferneTyp(int iNeuerTyp)
{
  if (iZaehler < 2)
    {
      iZaehler = 1;
      iTyp = iNeuerTyp;
    } else iZaehler--;
}



int Punkt::LieferTyp()
{return iTyp}

void Punkt::SetzePunkt()
{
  Punkt *I;
  I = Rechts;
  while (I != NULL)
    { I->SetzeTyp(2); I = I->Rechts; };
  I = Links;
  while (I != NULL)
    { I->SetzeTyp(2); I = I->Links; };
  I = RechtsOben;
  while (I != NULL)
    { I->SetzeTyp(2); I = I->RechtsOben; };
  I = RechtsUnten;
  while (I != NULL)
    { I->SetzeTyp(2); I = I->RechtsUnten; };
  I = LinksOben;
  while (I != NULL)
    { I->SetzeTyp(2); I = I->LinksOben; };
  I = LinksUnten;
  while (I != NULL)
    { I->SetzeTyp(2); I = I->LinksUnten; };
  SetzeTyp(1);
}


void Punkt::EntfernePunkt()
{
  Punkt *I;
  I = Rechts;
  while (I != NULL)
    { I->EntferneTyp(0); I = I->Rechts; };
  I = Links;
  while (I != NULL)
    { I->EntferneTyp(0); I = I->Links; };
  I = RechtsOben;
  while (I != NULL)
    { I->EntferneTyp(0); I = I->RechtsOben; };
  I = RechtsUnten;
  while (I != NULL)
    { I->EntferneTyp(0); I = I->RechtsUnten; };
  I = LinksOben;
  while (I != NULL)
    { I->EntferneTyp(0); I = I->LinksOben; };
  I = LinksUnten;
  while (I != NULL)
    { I->EntferneTyp(0); I = I->LinksUnten; };
  SetzeTyp(0);
}


Sechseck::Sechseck(int iKante)
{
  int iA;
  iKantenLaenge = iKante;
  iBreite = 2 * iKante - 1;
  iBreite* = 2;
  iNrPunkte = 1;
  for (iA = 0; iA < iKante; iA++)
      iNrPunkte + = 6 * iA;
  PunktListe = new PunktZeiger[iNrPunkte];
  iNrPunkte = 1;
  Punkte = new PunktZeiger[iBreite * iBreite];
  for (iA = 0; iA < iBreite * iBreite; iA++)
      Punkte[iA] = NULL;
  PunktListe[0] = new Punkt(0,0);
  (Punkte[iKante*iBreite+iKante] =
      PunktListe[0])->BildeVerbindungen(this);
}

Sechseck::~Sechseck()
{
  int iA;
  for (iA = 0; iA < iNrPunkte; iA++)
      delete PunktListe[iA];
}

Punkt* Sechseck::LieferePunkt(int iX, int iY)
{
  if ((abs(iX) >= 2*iKantenLaenge) || (abs(iY) >=
      iKantenLaenge)) return NULL;
  if (abs(iX) >= ((2*iKantenLaenge) - abs(iY))) return NULL;
  return Punkte[(iY+iKantenLaenge) * iBreite + iX +
      iKantenLaenge];
}

Punkt* Sechseck::LiefereOderErzeugePunkt(int iX, int iY)
{
  int iA,iB;
  if (abs(iX) >= 2*iKantenLaenge) || (abs(iY) >=
      iKantenLaenge)) return NULL;
  if (abs(iX) >= ((2*iKantenLaenge) - abs(iY))) return NULL;
  iA = iX+iKantenLaenge;
  iB = iY+iKantenLaenge;
  if (Punkte[iB*iBreite+iA] == NULL)
    {
      (Punkte[iB*iBreite+iA] =
        new Punkt(iX,iY))->BildeVerbindungen(this);
      PunktListe[iNrPunkte++] = Punkte[iB*iBreite+iA];
    };
  return Punkte[iB*iBreite+iA];
}

int Sechseck::NrPunkte()
{return iNrPunkte}

Punkt* Sechseck::PunktNr(int iNr)
{return PunktListe[iNr]}

Sechseck *Form;
int   iBest, iMaximum, iKantenLaenge, iNrPunkte;

void ZeigeSechseck()
{
  int iX, iY;
  for (iY = -iKantenLaenge+1; iY < iKantenLaenge; iY++)
    {
      for (iX = -((iKantenLaenge-1)*2);
        iX<2*iKantenLaenge-1; iX++)
        if (Form->LieferePunkt(iX,iY) == NULL) printf(" ");
        else
          switch (Form->LieferePunkt(iX,iY)->LieferTyp()) {
            case 0: printf(" "); break;
            case 1: printf("O"); break;
            case 2: printf("+"); break;
          };
      printf("\n");
    }
}

int Suche(int iPunkte,int iStart)
{
  int   iA;
  Punkt i*;
  if (iPunkte>iBest)
    {
      iBest = iPunkte;
      printf(" %i ",iPunkte, iBest);
      ZeigeSechseck();
      if (iBest == iMaximum) return 1;
    };
  for (iA = iStart; iA < iNrPunkte; iA++)
    if ((P = Form->PunktNr(iA))->LieferTyp() == 0)
      {
        P->SetzePunkt();
        if (Suche(iPunkte+1,iA+1)) return 1;
        P->EntfernePunkt();
      };
  return 0;
}

int main()
{
  printf("Kantenlaenge: ");
  scanf("%i",&iKantenLaenge);
  iBest = 0;
  iMaximum = 2 * iKantenLaenge - 1;
  Form = new Sechseck(iKantenLaenge);
  iNrPunkte = Form->NrPunkte();
  Suche(0,0);
  delete Form;
  return 0;
}
```

Design, Kuhn, Kammann & Kuhn, Köln Illustration: Gunnar Collet, Christina Reibsack, Köln Druck: DPS Druck und Verlag, Köln