Computer Systems (SS 2012)
Exercise 2: April 23, 2012

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at

March 23, 2012

The exercise is to be submitted by the denoted deadline via the submission interface of the Moodle course as a single file in zip (`.zip`) or tarred gzip (`.tgz`) format which contains the following files:

- A PDF file `Exercise`*`Number-MatNr`*`.pdf` (where *Number* is the number of the exercise and *MatNr* is your "Matrikelnummer") which consists of the following parts:
  1. A decent cover page with the title of the course, the number of the exercise, and the author of the solution (identified by name, Matrikelnummer and email address).
  2. For every source file, a listing in a *fixed width font*, e.g. `Courier`, (such that indentations are appropriately preserved) and an appropriate *font size* such that source code lines do not break.
  3. A description of all tests performed (copies of program inputs and program outputs) explicitly highlighting, if some test produces an unexpected result.
  4. Any additional explanation you would like to give. In particular, if your solution has unwanted problems or bugs, please document these explicitly (you will get more credit for such solutions).
- Each source file of your solution (no object files or executables).

Please obey the coding style recommendations posted on the course site.

Exercise 2: Delaunay Revisited

Rewrite the program of Exercise 1 in an object-oriented style[1]. In more detail:

Write a class `Point` with the following minimal interface

```
class Point
{
  double _x;
  double _y;
public:
  Point(double x, double y);
  double x() const;
  double y() const;
  void draw() const;
  void draw(Point* p1) const;
  void clear(Point* p1) const;
  int side(Point *p0, Point *p1) const;
};
```

such that `p = new Point(x,y)` creates a new point p with coordinates x, y, `p->x()` and `p->y()` return its coordinates, `p->draw()` draws that point, `p->draw(p1)` draws a line from that point to another point p_1 and `p->clear(p1)` erases that line. A call `p->side(p0,p1)` returns the side $(0, \pm 1)$ on which p is with respect to the lines through points p_0, p_1.

Write a class `Triangle` with the following minimal interface

```
class Triangle
{
  Point* _p0;
  Point* _p1;
  Point* _p2;
public:
  Triangle(Point* p0, Point* p1, Point *p2);
  Point* p0() const;
  Point* p1() const;
  Point* p2() const;
  bool inside(Point *p) const;
};
```

such that `t = new Triangle(p0,p1,p2)` creates a new triangle t with corners p_0, p_1, p_2, `t->p0()`, `t->p1()`, and `t->p2()` return its corners, and `t->inside(p)` is true, if and only if point p is inside t.

Class `Heap` is as in Exercise 1.

As for class `TriangleSet`, take a class with the following minimal interface:

```
#include <list>
```

[1] If you have not solved that exercise, you may ask a colleague for a solution.

```
class TriangleSet
{
  list<Triangle*> triangles;
  list<Triangle*>::iterator next;
public:
  TriangleSet() { }
  void add(Triangle* t)
  { triangles.push_back(t); }
  void remove(Triangle* t)
  { triangles.remove(t); }
  void gotoStart()
  { next = triangles.begin(); }
  bool hasNext()
  { return next != triangles.end(); }
  Triangle* getNext()
  { Triangle* t = *next; next++; return t; }
};
```

where `T = new TriangleSet()` creates a new set T, `T->add(t)` adds triangle t to T, and `T->remove(t)` removes t from T. You can iterate over all elements of T by first calling `S->gotoStart()`. After that you can repeatedly call `T->hasNext()` to determine, if T has another triangle. If yes, a call of `T->getNext()` will return that triangle.

As for class `EdgeStack` take the following class

```
#include <list>
class EdgeStack
{
  list<Point*> point0;
  list<Point*> point1;
  list<Triangle*> triangle;
public:
  void add(Point* p0, Point* p1, Triangle *t)
  { point0.push_back(p0); point1.push_back(p1); triangle.push_back(t); }
  bool isEmpty()
  { return point0.empty(); }
  Point* getFirstPoint()
  { Point* p = point0.back(); point0.pop_back(); return p; }
  Point* getSecondPoint()
  { Point* p = point1.back(); point1.pop_back(); return p; }
  Triangle* getTriangle()
  { Triangle* t = triangle.back(); triangle.pop_back(); return t; }
};
```

where `E = new EdgeStack()` sets E to the empty set. A call `E->add(a,b,t)` adds $\langle a, b, t\rangle$ to E. A call `E->isEmpty()` determines whether E is empty. If not, subsequent calls of `E->getFirstPoint()`, `E->getSecondPoint()`, `E->getTriangle()` return a triple $\langle a, b, t\rangle$ of E and remove it from E.

The main functionality of the program is again provided by a class `Delaunay` such that a call of a function `T=Delaunay::triangulate(number,points)` returns the triangulation (triangle set) T of the *number* points in array *points* (without drawing T). Afterwards, a call of a function `Delaunay::draw(T,x,y,w,h)` draws T in a rectangle with left upper corner x, y, width w, and height h (for the drawing, the coordinates of the points of T are to be transformed in such a way, that this rectangle becomes the smallest enclosing rectangle of the point set). Please note that the heap management functions `Heap::init()` and `Heap::clear()` should be called outside `triangulate()` (in particular, the heap must not be cleared before T is disposed).

Test the program as in Exercise 1, but draw the computed triangulation at least twice in non-overlapping regions of the screen with different sizes (also draw a frame around these regions). The deliverables of this exercise are the same as in Exercise 1.