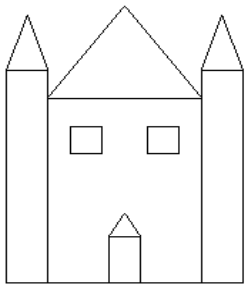
Procedural decomposition examples

A note about graphics: The graphics library we use in CS106B/X supports simple pen-based drawing in a Cartesian coordinate system scaled on inches. To familiarize yourself with it, check out section 5.3 of the reader and peruse the **`graphics.h`** and **`extgraph.h`** header files in the reader appendix or browse the "Documentation" link of our class web site. We will ask you to write some simple graphics code this quarter but it's not a focus for us, so I'd advise picking up the details on an as-needed basis. For assignments that use fancy graphics, we'll typically provide the graphics for you since writing that code can be boring and tedious and we'd rather not bog you down.

This handout is adopted from our old CS106A (in C) course, where we used this as an example for learning how to decompose a problem. Decomposition in the procedural paradigm means something slightly different than the decomposition you practiced in the object-oriented world. An object-oriented program is divided in classes, which then separate their operations into methods. The approach in a procedural program is to divide the problem into its component tasks, implemented as separate functions, each of which is potentially further subdivided into smaller functions. For example, the **`usher.cpp`** program presented below is defined in terms of boxes, circles, grids, and triangles. Identifying the general tools is part of the strategy of **bottom-up implementation**. Breaking the program down step by step is the process of **top-down design.** Both strategies are essential to your success as a programmer.

This example is drawn from Exercise 8 of Chapter 7 of *The Art and Science of C*, which asks you to draw a diagram of the Edgar Allen Poe's (very scary) House of Usher.



The point of this exercise was further practice with the graphics library and an exercise in top-down design and stepwise refinement. Look the picture, try to decompose the picture into its components, look for opportunities for unification and come up with a reasonable set of functions that can accomplish the job.

Use our solution below as an illustrative example of simple, complete C++ program that demonstrates good design and style. Note the many helper routines and how parameters allow those helpers to be as general as possible. A good design should make it relatively straightforward to make modifications — how would you change the program to make the towers taller or add panes to the windows or change the towers to rounded instead of peaked tops? A clean design should also allow you to quickly zero in on the troubled area when hunting bugs — if the door was being drawn in the wrong location, where would you look to try to fix it?

```
/*
 * File: usher.cpp
 * ---------------
 * Program to draw a representation of the house of Usher.
 */
#include "genlib.h"
#include "graphics.h"

/*
 * Constants
 * ---------
 * The following constants control the sizes of elements in the display.
 */

const double HouseWidth = 1.5;
const double HouseHeight = 2.0;
const double HouseArch = 1.0;

const double TowerWidth = 0.4;
const double TowerHeight = 2.3;
const double TowerArch = 0.6;

const double TotalHouseHeight = (HouseHeight + HouseArch);

const double DoorWidth = 0.3;
const double DoorHeight =.5;
const double DoorArch = .25;

const double WindowLevel = 1.4;
const double WindowSize = 0.3;


/* Function prototypes */

void DrawBox(double x, double y, double width, double height);
void DrawTriangle(double x, double y, double base, double height);
void DrawPeakedBox(double x, double y,
                           double width, double height, double arch);
void DrawWindow(double x, double y);
void DrawDoor(double x, double y);
void DrawTower(double x, double y);
void DrawMainHouse(double x, double y);
void DrawHouseOfUsher(double x, double y);
```

```
/* Main program calculates where to place lower left corner so that
 * entire house is centered within the graphics window.
 */
int main()
{
    double  centerX, centerY;

    InitGraphics();
    centerX = GetWindowWidth()/2;
    centerY = GetWindowHeight()/2;
    DrawHouseOfUsher(centerX - HouseWidth/2 - TowerWidth,
                     centerY - TotalHouseHeight/2);
    return 0;
}




/*
 * Function: DrawHouseOfUsher
 * Usage: DrawHouseOfUsher(x, y);
 * ------------------------------
 * Draws the complete House of Usher with lower left corner at (x, y).
 * The left tower is drawn based at (x,y), the main house is drawn right
 * next to it, and the right tower is placed on the other side.
 */
void DrawHouseOfUsher(double x, double y)
{
    DrawTower(x, y);
    DrawMainHouse(x + TowerWidth, y);
    DrawTower(x + TowerWidth + HouseWidth, y);
}




/*
 * Function: DrawMainHouse
 * Usage: DrawMainHouse(x, y);
 * ---------------------------
 * Draws the main part of the house, including the door and windows.  The
 * door is centered and the two windows are balanced on either side.  The
 * lower left corner of the house will be placed at (x, y).
 */
void DrawMainHouse(double x, double y)
{
    DrawPeakedBox(x, y, HouseWidth, HouseHeight, HouseArch);
    DrawDoor(x + (HouseWidth- DoorWidth)/2, y);
    double windowX = x + HouseWidth/4 - WindowSize/2;
    DrawWindow(windowX, y +WindowLevel);
    windowX += HouseWidth/2;
    DrawWindow(windowX, y + WindowLevel);
}
```

```
/*
 * Function: DrawTower
 * Usage: DrawTower(x, y);
 * -----------------------
 * Draws a side tower with lower left corner at (x, y).  Uses
 * the TowerHeight, TowerWidth, and TowerArch constants to set sizes.
 */
void DrawTower(double x, double y)
{
    DrawPeakedBox(x, y, TowerWidth, TowerHeight, TowerArch);
}



/*
 * Function: DrawDoor
 * Usage: DrawDoor(x, y);
 * ----------------------
 * Draws the door at (x, y).  Uses the DoorHeight, DoorWidth, and
 * DoorArch constants to set sizes of the various door components.
 */
void DrawDoor(double x, double y)
{
   DrawPeakedBox(x, y, DoorWidth, DoorHeight, DoorArch);
}




/*
 * Function: DrawWindow
 * Usage: DrawWindow(x, y);
 * ------------------------
 * Draws a single window with the lower left corner at (x, y).  Uses
 * the WindowSize constant for the window width and height.
 */
void DrawWindow(double x, double y)
{
    DrawBox(x, y, WindowSize, WindowSize);
}




/*
 * Function: DrawPeakedBox
 * Usage: DrawPeakedBox(x, y, width, height, arch);
 * ------------------------------------------------
 * Draws a rectangle with a triangular top.  The arguments are as
 * in DrawBox, with an additional arch parameter indicating the
 * height of the triangle.  This function is a common element in
 * several parts of the picture.
 */
void DrawPeakedBox(double x, double y,
                           double width, double height, double arch)
{
     DrawBox(x, y, width, height);
     DrawTriangle(x, y + height, width, arch);
}
```

```
/*
 * Function: DrawBox
 * Usage: DrawBox(x, y, width, height);
 * ------------------------------------
 * This function draws a rectangle of the given width and height with
 * its lower left corner at (x, y).
 */

void DrawBox(double x, double y, double width, double height)
{
    MovePen(x, y);
    DrawLine(width, 0);
    DrawLine(0, height);
    DrawLine(-width, 0);
    DrawLine(0, -height);
}




/*
 * Function: DrawTriangle
 * Usage: DrawTriangle(x, y, base, height);
 * ----------------------------------------
 * This function draws an isosceles triangle (i.e., having two equal sides)
 * with a horizontal base.  The coordinate of the left endpoint of the base
 * is (x, y), and the triangle has the indicated base length and height.
 * If height is positive, the triangle points upward.  If height is
 * negative, the triangle points downward.
 */
void DrawTriangle(double x, double y, double base, double height)
{
     MovePen(x, y);
     DrawLine(base, 0);
     DrawLine(-base/2, height);
     DrawLine(-base/2, -height);
}
```

The regular polygon: an exercise in generalization

To construct shapes from the graphics primitives, you might create utility functions: one to draw a rectangle, another for triangles, one for a diamond, and so on. When you think about, you might want to unify these into a more general shape-drawing function that can draw a shape with any number of equilateral sides. Given a radius and applying some simple geometry, we can create a function to draw such a polygon, triangles, squares, pentagons, and so on. Now notice a diamond is nothing more than a square on its side (i.e. rotated 45 degrees). By adding a starting angle of rotation, we can further generalize our function for those shapes, too. Demonstrating generalization to the extreme, here is a function that can draw any equilateral polygon with any starting angle of rotation:

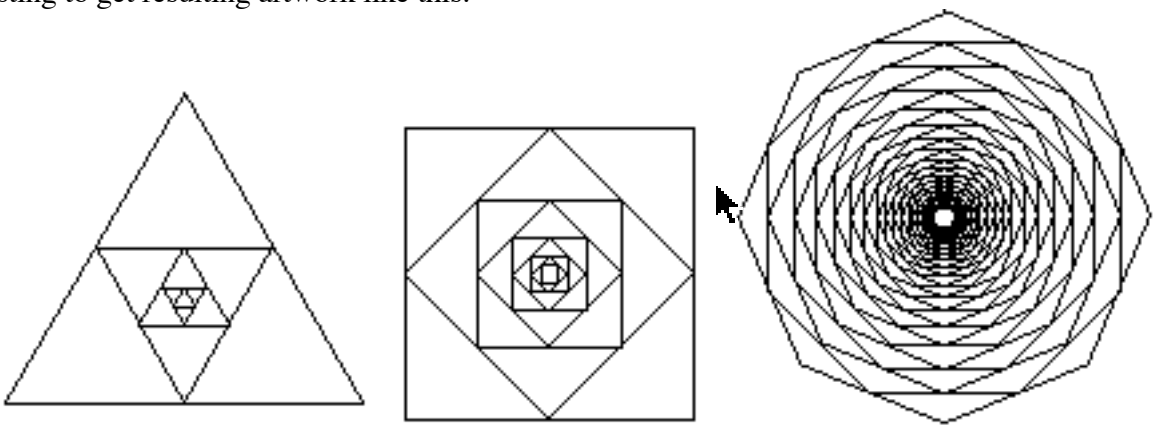
```
/*
 * Function: DrawPolygon
 * Usage: DrawPolygon(cx, cy, numSides, radius, startAngle);
 * ---------------------------------------------------------
 * Draws a n-sided equilateral polygon centered around the point (cx, cy)
 * and of the size such that it would be exactly inscribed in a circle
 * of the specified radius.  The starting angle sets where the polygon
 * begins. (i.e. a "corner" is placed at exactly that angle)  From there,
 * the function will draw segments of the polygon in counter-clockwise
 * direction until all sides have been drawn.
 */
void DrawPolygon(double centerX, double centerY, int numSides,
                     double radius, double startAngleInRads)
{
   double angleIncr = (2*PI)/numSides;
   double angle = startAngleInRads;
   MovePen(centerX + radius*sin(angle),centerY + radius*cos(angle));
   for (int i = 0; i < numSides; i++) {
        DrawChord(radius, angle, angle + angleIncr);
        angle += angleIncr;
   }
}


/*
 * Function: DrawChord
 * Usage: DrawChord(radius, startAngle, stopAngle);
 * ------------------------------------------------
 * Helper function for DrawPolyon.  Given the pen is position at angle
 * startAngle on a circle of specified radius, this function will draw a
 * chord from that position to the  position of the stopAngle on the same
 * circle, therefore drawing a chord on the circle from start to stop.
 */
void DrawChord(double radius, double start, double stop)
{
     double startX = radius*sin(start);
     double stopX  = radius*sin(stop);
     double startY = radius*cos(start);
     double stopY  = radius*cos(stop);
     DrawLine(stopX - startX, stopY - startY);
}
```

Drawing stop signs and yield signs and dodecagons and benzene rings and even circles (choose `numSides` to be huge…) is a snap! We can also construct any number of nice wrapper functions that provide a simpler interface for the common shapes. Functions like `DrawDiamond` or `DrawTriangle` or `DrawOctagon` would be simple covers that pass through the correct arguments to the multi-

purpose **DrawPolygon** function. Therefore, a client's use of our functions would be cleaner and more readable.

Lastly, just for fun, we can create a function to draw spirograph-like nested polygons by constructing a function that repeatedly calls **DrawPolygon** in a loop while slightly shrinking and rotating on each nesting to get resulting artwork like this:



n = 3 *n = 4* *n = 8*

```
/*
 * Function: DrawNestedPolygon
 * Usage: DrawNestedPolygon(cx, cy, numSides, radius);
 * ---------------------------------------------------
 * Draws a set of nested n-sided equilateral polygons centered around
 * (cx, cy).  The outermost polygon is of size according to radius and
 * has starting angle of 0.  The nextmost polygon is rotated half the distance
 * of one side and its radius decreased so that its vertices hit the
 * halfway points of the outer polygon edges.  It continues nesting like
 * this until it gets to the center and the radius becomes too small to
 * draw any more shapes.
 */
void DrawNestedPolygon(double cx, double cy, int numSides, double radius)
{
     double angle = 0.0;
     double angleIncr = PI/numSides;
     while (radius > 0.05) {
          DrawPolygon(cx,cy,numSides,radius,angle);
          radius = cos(angleIncr)*radius;
          angle += angleIncr;
     }
}
```

How many IBM repairmen does it take to change a flat tire? Five. One to jack up the car, and four to swap tires until they find the one that is flat.