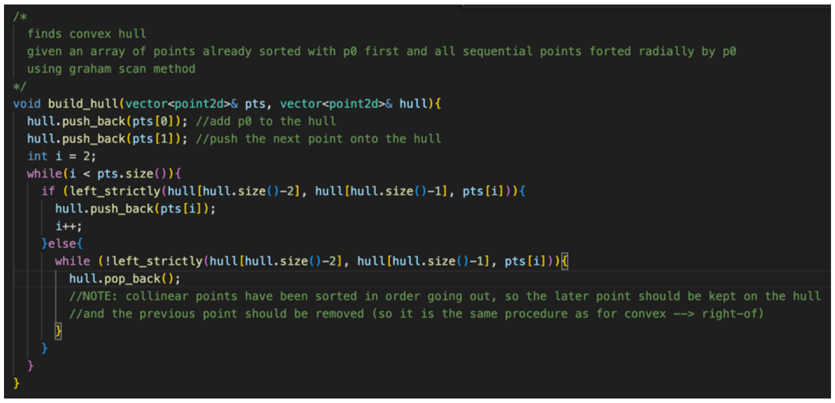
(1) Handling degenerate and collinear inputs:

   a. points on the hull that are collinear

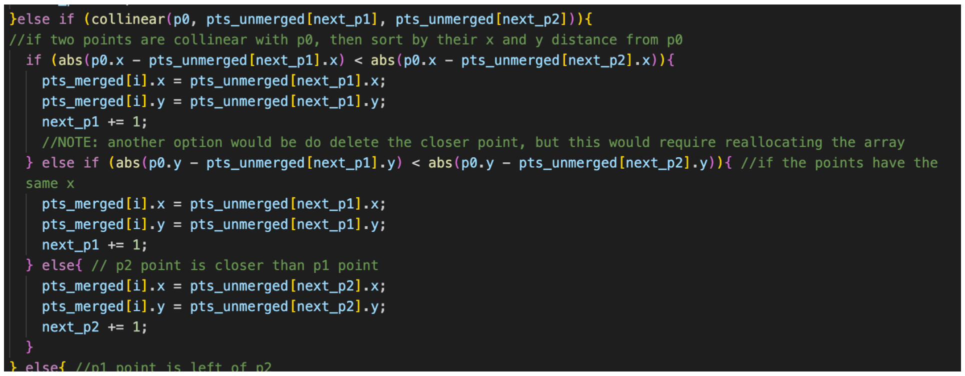
      i. for any set of 3+ collinear points that are all three adjacent in radially sorted order with respect to P0, I pop off the center point(s) and keep the first and last points. I do this by comparing with left_strictly() instead of left_on(), so that any set of 3 collinear points is treated the same as any set of three points with a right turn (ie the center one is removed)



```
/*
  finds convex hull
  given an array of points already sorted with p0 first and all sequential points forted radially by p0
  using graham scan method
*/
void build_hull(vector<point2d>& pts, vector<point2d>& hull){
  hull.push_back(pts[0]); //add p0 to the hull
  hull.push_back(pts[1]); //push the next point onto the hull
  int i = 2;
  while(i < pts.size()){
    if (left_strictly(hull[hull.size()-2], hull[hull.size()-1], pts[i])){
      hull.push_back(pts[i]);
      i++;
    }else{
      while (!left_strictly(hull[hull.size()-2], hull[hull.size()-1], pts[i])){
        hull.pop_back();
        //NOTE: collinear points have been sorted in order going out, so the later point should be kept on the hull
        //and the previous point should be removed (so it is the same procedure as for convex --> right-of)
      }
    }
  }
}
```

   b. points that are collinear with P0:

      i. in the initial sorting, when sorting radially with respect to P0, if two points are collinear with P0, I sort them by ascending x and why distance from P0 (see below). This way, when these three points are handled in the next step of the graham scan (shown above in a), the closer points are popped off the hull and only the furthest away point is included



```
}else if (collinear(p0, pts_unmerged[next_p1], pts_unmerged[next_p2])){
//if two points are collinear with p0, then sort by their x and y distance from p0
  if (abs(p0.x - pts_unmerged[next_p1].x) < abs(p0.x - pts_unmerged[next_p2].x)){
    pts_merged[i].x = pts_unmerged[next_p1].x;
    pts_merged[i].y = pts_unmerged[next_p1].y;
    next_p1 += 1;
    //NOTE: another option would be do delete the closer point, but this would require reallocating the array
  } else if (abs(p0.y - pts_unmerged[next_p1].y) < abs(p0.y - pts_unmerged[next_p2].y)){ //if the points have the
  same x
    pts_merged[i].x = pts_unmerged[next_p1].x;
    pts_merged[i].y = pts_unmerged[next_p1].y;
    next_p1 += 1;
  } else{ // p2 point is closer than p1 point
    pts_merged[i].x = pts_unmerged[next_p2].x;
    pts_merged[i].y = pts_unmerged[next_p2].y;
    next_p2 += 1;
  }
} else{ //p1 point is left of p2
```

   c. points that are collinear and at minimum y

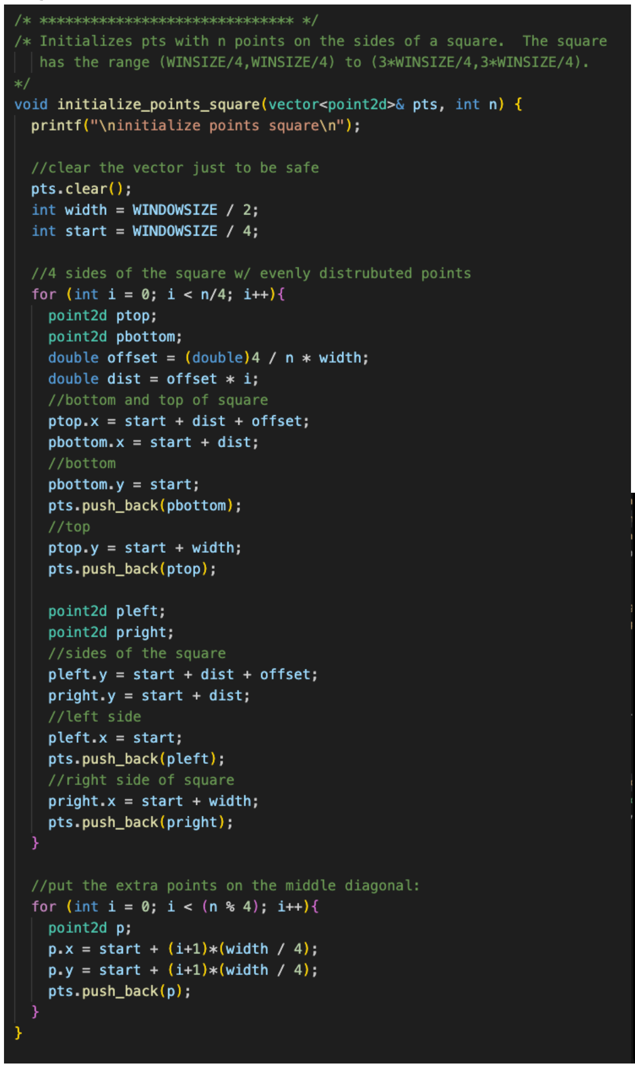
      i. the point with the minimum y value is taken as P0, but if there are multiple points with the same y value, then the point among them with the lowest x value is taken, this way, only one of the most extreme of these collinear points is taken to be on the hull

(2) pictures of the two initializers you created (2) pictures of the other initializers you used;

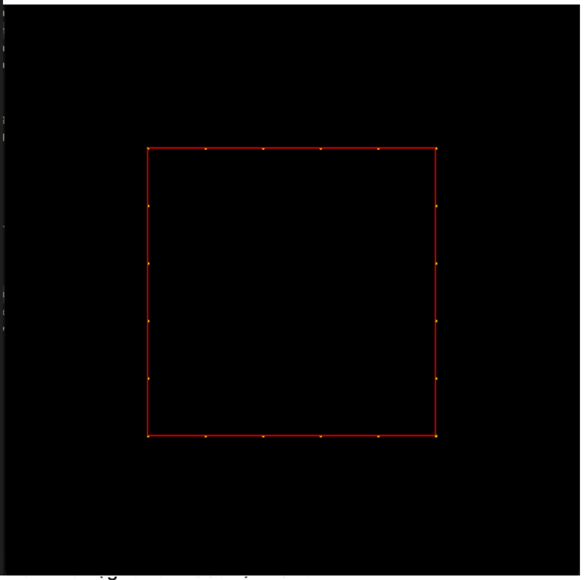
(3) pictures of the convex hulls computed for each initial configuration of points, n=20

MY INITIALIZERS (1/3)

SQUARE:

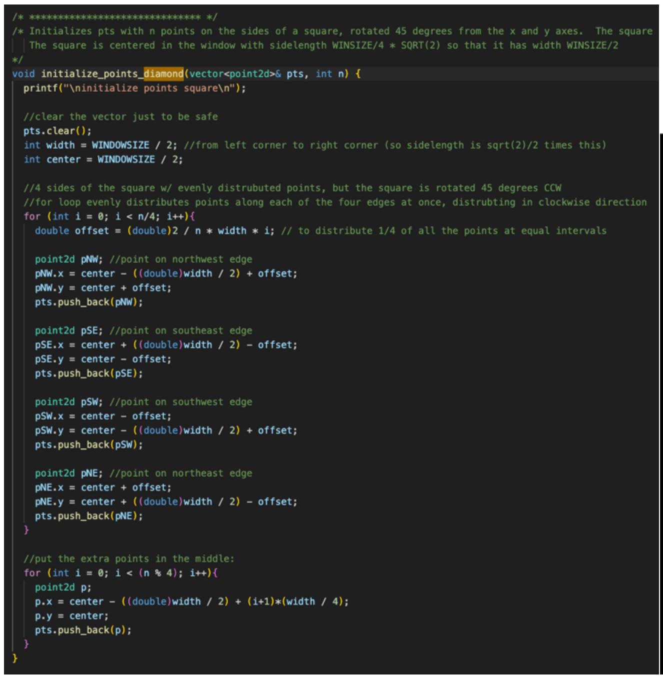


```
/* ******************************* */
/* Initializes pts with n points on the sides of a square.  The square
   has the range (WINSIZE/4,WINSIZE/4) to (3*WINSIZE/4,3*WINSIZE/4).
*/
void initialize_points_square(vector<point2d>& pts, int n) {
  printf("\ninitialize points square\n");

  //clear the vector just to be safe
  pts.clear();
  int width = WINDOWSIZE / 2;
  int start = WINDOWSIZE / 4;

  //4 sides of the square w/ evenly distrubuted points
  for (int i = 0; i < n/4; i++){
    point2d ptop;
    point2d pbottom;
    double offset = (double)4 / n * width;
    double dist = offset * i;
    //bottom and top of square
    ptop.x = start + dist + offset;
    pbottom.x = start + dist;
    //bottom
    pbottom.y = start;
    pts.push_back(pbottom);
    //top
    ptop.y = start + width;
    pts.push_back(ptop);

    point2d pleft;
    point2d pright;
    //sides of the square
    pleft.y = start + dist + offset;
    pright.y = start + dist;
    //left side
    pleft.x = start;
    pts.push_back(pleft);
    //right side of square
    pright.x = start + width;
    pts.push_back(pright);
  }

  //put the extra points on the middle diagonal:
  for (int i = 0; i < (n % 4); i++){
    point2d p;
    p.x = start + (i+1)*(width / 4);
    p.y = start + (i+1)*(width / 4);
    pts.push_back(p);
  }
}
```

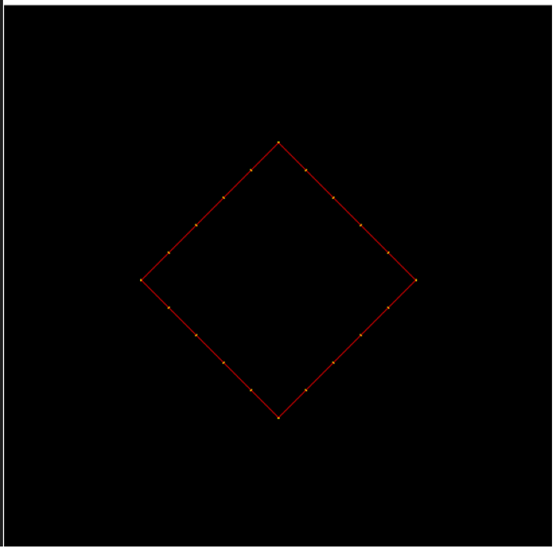


```
hull: [375,125] [375,375] [125,375] [125,125]
hull time:  [0.00u (56%) 0.00s (56%) 0.00 112.5%]
```

MY INITIALIZERS (2/3)
ROTATED SQUARE:

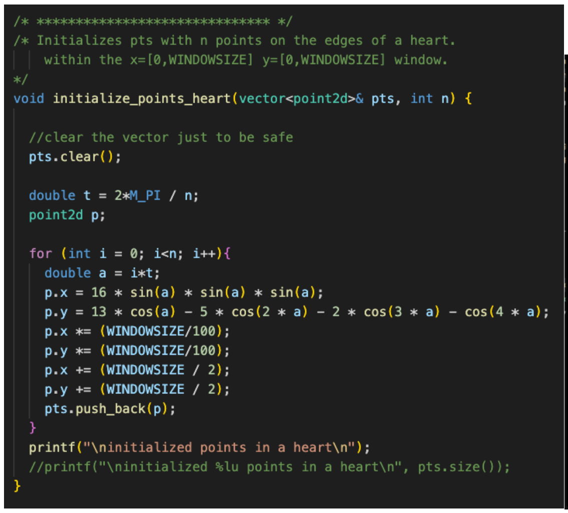


```
/* ****************************** */
/* Initializes pts with n points on the sides of a square, rotated 45 degrees from the x and y axes.  The square
   The square is centered in the window with sidelength WINSIZE/4 * SQRT(2) so that it has width WINSIZE/2
*/
void initialize_points_diamond(vector<point2d>& pts, int n) {
  printf("\ninitialize points square\n");

  //clear the vector just to be safe
  pts.clear();
  int width = WINDOWSIZE / 2; //from left corner to right corner (so sidelength is sqrt(2)/2 times this)
  int center = WINDOWSIZE / 2;

  //4 sides of the square w/ evenly distrubuted points, but the square is rotated 45 degrees CCW
  //for loop evenly distributes points along each of the four edges at once, distrubting in clockwise direction
  for (int i = 0; i < n/4; i++){
    double offset = (double)2 / n * width * i; // to distribute 1/4 of all the points at equal intervals

    point2d pNW; //point on northwest edge
    pNW.x = center - ((double)width / 2) + offset;
    pNW.y = center + offset;
    pts.push_back(pNW);

    point2d pSE; //point on southeast edge
    pSE.x = center + ((double)width / 2) - offset;
    pSE.y = center - offset;
    pts.push_back(pSE);

    point2d pSW; //point on southwest edge
    pSW.x = center - offset;
    pSW.y = center - ((double)width / 2) + offset;
    pts.push_back(pSW);

    point2d pNE; //point on northeast edge
    pNE.x = center + offset;
    pNE.y = center + ((double)width / 2) - offset;
    pts.push_back(pNE);
  }

  //put the extra points in the middle:
  for (int i = 0; i < (n % 4); i++){
    point2d p;
    p.x = center - ((double)width / 2) + (i+1)*(width / 4);
    p.y = center;
    pts.push_back(p);
  }
}
```

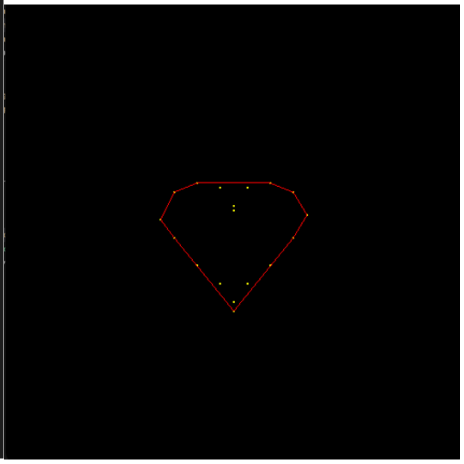


```
hull: [250,125] [375,250] [250,375] [125,250]
hull time:  [0.00u (42%) 0.00s (75%) 0.00 116.7%]
```

MY INITIALIZERS (3/3)
HEART W/ N POINTS:



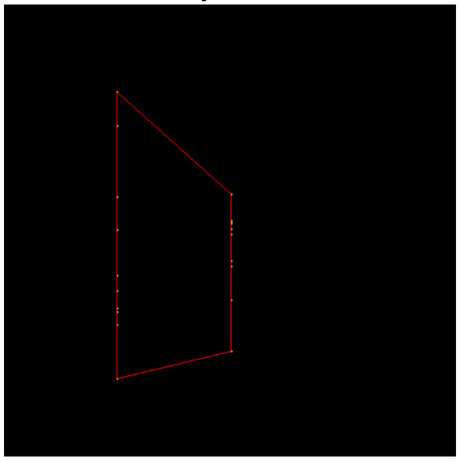
```
/* ****************************** */
/* Initializes pts with n points on the edges of a heart.
   within the x=[0,WINDOWSIZE] y=[0,WINDOWSIZE] window.
*/
void initialize_points_heart(vector<point2d>& pts, int n) {

  //clear the vector just to be safe
  pts.clear();

  double t = 2*M_PI / n;
  point2d p;

  for (int i = 0; i<n; i++){
    double a = i*t;
    p.x = 16 * sin(a) * sin(a) * sin(a);
    p.y = 13 * cos(a) - 5 * cos(2 * a) - 2 * cos(3 * a) - cos(4 * a);
    p.x *= (WINDOWSIZE/100);
    p.y *= (WINDOWSIZE/100);
    p.x += (WINDOWSIZE / 2);
    p.y += (WINDOWSIZE / 2);
    pts.push_back(p);
  }
  printf("\ninitialized points in a heart\n");
  //printf("\ninitialized %lu points in a heart\n", pts.size());
}
```



```
hull: [250,165] [315,245] [330,270] [315,295] [290,305]
[210,305] [185,295] [170,265] [185,245]
hull time:  [0.00u (64%) 0.00s (45%) 0.00 109.1%]
```

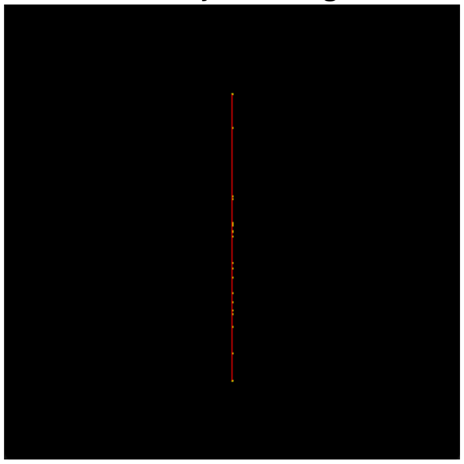
CLASSMATES' INITIALIZERS:

Victoria and Ziyou's two vertical lines:



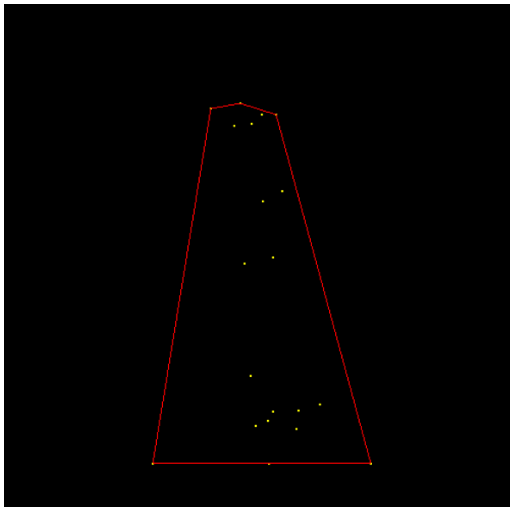
```
hull: [125, 88] [250,118] [250,290] [125,402]
hull time:  [0.00u (69%) 0.00s (62%) 0.00 130.8%]
```

Victoria and Ziyou's single vertical line



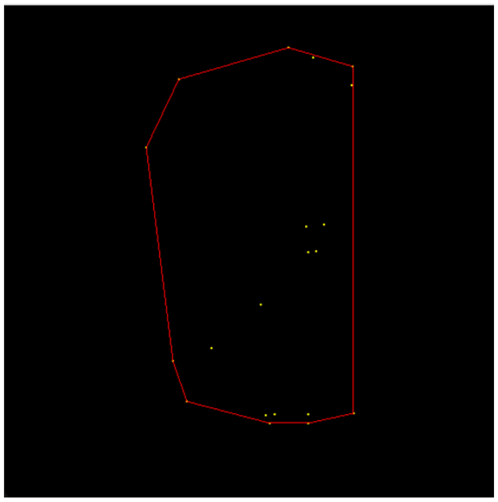
```
hull: [250, 88] [250,402]
hull time:  [0.00u (58%) 0.00s (75%) 0.00 133.3%]
```

Tom's "1" initializer:



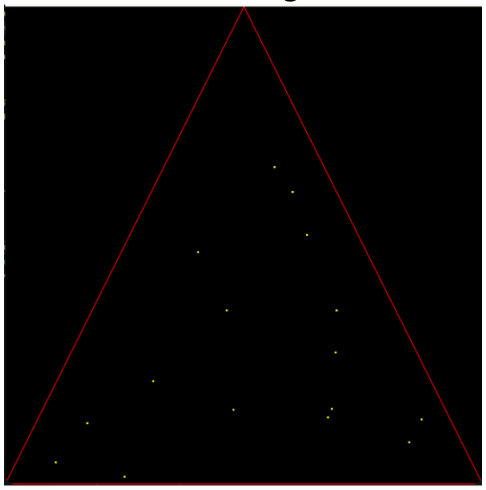
```
hull: [361, 50] [268,392] [233,403] [204,398] [147, 50]
hull time:  [0.00u (67%) 0.00s (44%) 0.00 111.1%]
```

Tom's "2" initializer:



```
hull: [308, 77] [354, 87] [353,438] [288,457] [177,425]
[144,356] [171,140] [185, 99] [269, 77]
hull time:  [0.00u (67%) 0.00s (58%) 0.00 125.0%]
```

Abhi and Max's triangle initializer:



```
hull: [500,  0] [250,500] [  0,  0]
hull time:  [0.00u (55%) 0.00s (82%) 0.00 136.4%]
```

(4) if your code does not work in all cases, explain.
    a. NA
(5) any extra features.
    a. I sped up the algorithm by first removing all points within the lines between the x and y extreme points
(6) Time you spent in: Thinking; Programming; Testing; Documenting; Total;

Getting set-up/ reading through assignment	35 min
Programming	11.5 hours total: 8.5 hours to get functional graham scan, including implementing my own merge sort function

	3 hours to add delete_points function (for speed enhancement) and to write initializers
testing	90 min
documenting	1 hour
total	14.5 hours

(7) Brief reflection prompts (you don't need to address all): how challenging did you find this project? what did you learn by doing this project? What did you wish you did differently? If you worked as a team, how did that go? What would you like to explore further?)

I wish I had used a c++ sort instead of implementing my own, although I was grateful for the coding practice and merge sort review.

I would like to better understand memory management in c++ and best practices for using void functions to edit objects at pointers vs. returning objects...

I would also like to become more fluent with pointers