

Polygon Data Structures in C and C++

A Practical Header / Implementation Split with Transformations

1 Overview

This document presents clean, reusable polygon data structures in both C++ and C, suitable for direct inclusion in your projects. The focus is on:

- A simple and idiomatic `Polygon` class in C++.
- A dynamic-array-based `Polygon` structure in C.
- Core geometric operations:
 - Area (shoelace formula).
 - Point-in-polygon test (ray casting).
 - Perimeter computation.
- Basic transformations:
 - Translation.
 - Rotation about an arbitrary origin.
 - Scaling about an arbitrary origin.
- Clear header and implementation splits:
 - `Polygon.hpp` / `Polygon.cpp` for C++.
 - `polygon.h` / `polygon.c` for C.

All code listings use the `minted` package for syntax highlighting.

2 C++ Polygon API

2.1 Header File: Polygon.hpp

```
1  #ifndef POLYGON_HPP
2  #define POLYGON_HPP
3
4  #include <cstdint>
5  #include <vector>
6
7  // Basic 2D point
8  struct Point {
9      double x{};
10     double y{};
11 };
12
13 // Simple polygon represented as an ordered list of vertices.
14 // The polygon is implicitly closed: the last vertex connects back to the first.
15 class Polygon {
16 public:
17     Polygon() = default;
18
19     explicit Polygon(std::vector<Point> points);
20
21     // Vertex management
22     void addVertex(const Point& p);
23     std::size_t vertexCount() const;
24
25     // Geometric properties
26     double area() const;           // Shoelace formula (unsigned)
27     double perimeter() const;     // Sum of edge lengths
28     bool containsPoint(const Point& p) const; // Ray-casting
29
30     // Transformations
31     void translate(double dx, double dy);
32     void rotate(double angleRadians,
33                 double originX,
34                 double originY);
35     void scale(double sx,
36                double sy,
37                double originX,
38                double originY);
39
40     // Read-only access to vertices
41     const std::vector<Point>& getVertices() const;
42
43 private:
44     std::vector<Point> vertices;
45 };
46
47 #endif // POLYGON_HPP
```

2.2 Implementation File: Polygon.cpp

```
1  #include "Polygon.hpp"
2  #include <cmath>
3
4  Polygon::Polygon(std::vector<Point> points)
5      : vertices(std::move(points)) {}
6
7  void Polygon::addVertex(const Point& p) {
8      vertices.push_back(p);
9  }
10
11  std::size_t Polygon::vertexCount() const {
12      return vertices.size();
13  }
14
15  // Shoelace formula for polygon area.
16  // Returns an unsigned area (always >= 0).
17  double Polygon::area() const {
18      const std::size_t n = vertices.size();
19      if (n < 3) {
20          return 0.0;
21      }
22
23      double sum = 0.0;
24      for (std::size_t i = 0; i < n; ++i) {
25          const Point& a = vertices[i];
26          const Point& b = vertices[(i + 1) % n]; // wrap to first
27          sum += a.x * b.y - b.x * a.y;
28      }
29      return 0.5 * std::abs(sum);
30  }
31
32  // Perimeter: sum of Euclidean distances between consecutive vertices.
33  double Polygon::perimeter() const {
34      const std::size_t n = vertices.size();
35      if (n < 2) {
36          return 0.0;
37      }
38
39      double length = 0.0;
40      for (std::size_t i = 0; i < n; ++i) {
41          const Point& a = vertices[i];
42          const Point& b = vertices[(i + 1) % n];
43          const double dx = b.x - a.x;
44          const double dy = b.y - a.y;
45          length += std::sqrt(dx * dx + dy * dy);
46      }
47      return length;
48  }
49
50  // Ray-casting algorithm for point-in-polygon.
51  // Returns true if the point is strictly inside or on the boundary (approximately).
```

```

52  bool Polygon::containsPoint(const Point& p) const {
53      const std::size_t n = vertices.size();
54      if (n < 3) {
55          return false;
56      }
57
58      bool inside = false;
59      for (std::size_t i = 0, j = n - 1; i < n; j = i++) {
60          const Point& pi = vertices[i];
61          const Point& pj = vertices[j];
62
63          const bool intersect =
64              ((pi.y > p.y) != (pj.y > p.y)) &&
65              (p.x < (pj.x - pi.x) * (p.y - pi.y) / (pj.y - pi.y) + pi.x);
66
67          if (intersect) {
68              inside = !inside;
69          }
70      }
71      return inside;
72  }
73
74  // Translation: shift every vertex by (dx, dy).
75  void Polygon::translate(double dx, double dy) {
76      for (Point& v : vertices) {
77          v.x += dx;
78          v.y += dy;
79      }
80  }
81
82  // Rotation about an arbitrary origin (originX, originY).
83  // angleRadians is measured counter-clockwise.
84  void Polygon::rotate(double angleRadians,
85                      double originX,
86                      double originY) {
87      const double c = std::cos(angleRadians);
88      const double s = std::sin(angleRadians);
89
90      for (Point& v : vertices) {
91          const double xShifted = v.x - originX;
92          const double yShifted = v.y - originY;
93
94          const double xr = xShifted * c - yShifted * s;
95          const double yr = xShifted * s + yShifted * c;
96
97          v.x = xr + originX;
98          v.y = yr + originY;
99      }
100  }
101
102  // Scaling about an arbitrary origin.
103  // (sx, sy) are scale factors in x and y respectively.
104  void Polygon::scale(double sx,

```

```

105         double sy,
106         double originX,
107         double originY) {
108     for (Point& v : vertices) {
109         const double xShifted = v.x - originX;
110         const double yShifted = v.y - originY;
111
112         const double xs = xShifted * sx;
113         const double ys = yShifted * sy;
114
115         v.x = xs + originX;
116         v.y = ys + originY;
117     }
118 }
119
120 const std::vector<Point>& Polygon::getVertices() const {
121     return vertices;
122 }

```

2.3 Example Usage (C++)

```

1  #include "Polygon.hpp"
2  #include <iostream>
3
4  int main() {
5      // Simple triangle
6      std::vector<Point> pts = {
7          {0.0, 0.0},
8          {1.0, 0.0},
9          {0.0, 1.0}
10     };
11
12     Polygon poly(pts);
13
14     std::cout << "Area: " << poly.area() << "\n";
15     std::cout << "Perimeter: " << poly.perimeter() << "\n";
16
17     Point p{0.25, 0.25};
18     std::cout << "Contains (0.25, 0.25)? "
19         << (poly.containsPoint(p) ? "yes" : "no") << "\n";
20
21     // Translate by (1, 1)
22     poly.translate(1.0, 1.0);
23
24     // Rotate 90 degrees CCW around origin (0, 0)
25     poly.rotate(3.141592653589793 / 2.0, 0.0, 0.0);
26
27     // Uniformly scale by 2 around origin (0, 0)
28     poly.scale(2.0, 2.0, 0.0, 0.0);
29
30     return 0;
31 }

```

3 C Polygon API

3.1 Header File: polygon.h

```
1  #ifndef POLYGON_H
2  #define POLYGON_H
3
4  #include <stddef.h> // size_t
5
6  // Basic 2D point
7  typedef struct {
8      double x;
9      double y;
10 } Point;
11
12 // Polygon represented as a dynamic array of vertices.
13 typedef struct {
14     Point* vertices;
15     size_t count;
16     size_t capacity;
17 } Polygon;
18
19 // Lifecycle
20 int polygon_init(Polygon* poly, size_t initial_capacity);
21 void polygon_free(Polygon* poly);
22
23 // Vertex management
24 int polygon_add_vertex(Polygon* poly, Point p);
25
26 // Geometric properties
27 double polygon_area(const Polygon* poly);
28 double polygon_perimeter(const Polygon* poly);
29 int polygon_contains_point(const Polygon* poly, Point p);
30
31 // Transformations
32 void polygon_translate(Polygon* poly, double dx, double dy);
33 void polygon_rotate(Polygon* poly,
34                     double angleRadians,
35                     double originX,
36                     double originY);
37 void polygon_scale(Polygon* poly,
38                   double sx,
39                   double sy,
40                   double originX,
41                   double originY);
42
43 #endif // POLYGON_H
```

3.2 Implementation File: polygon.c

```
1  #include "polygon.h"
2
3  #include <stdlib.h> // malloc, realloc, free
```

```

4  #include <math.h>    // fabs, sqrt, cos, sin
5
6  static int polygon_reserve(Polygon* poly, size_t new_capacity) {
7      if (new_capacity <= poly->capacity) {
8          return 0;
9      }
10     Point* new_data =
11         (Point*)realloc(poly->vertices, new_capacity * sizeof(Point));
12     if (!new_data) {
13         return -1;
14     }
15     poly->vertices = new_data;
16     poly->capacity = new_capacity;
17     return 0;
18 }
19
20 int polygon_init(Polygon* poly, size_t initial_capacity) {
21     if (!poly) {
22         return -1;
23     }
24     poly->count = 0;
25     poly->capacity = (initial_capacity > 0) ? initial_capacity : 4;
26
27     poly->vertices =
28         (Point*)malloc(poly->capacity * sizeof(Point));
29     if (!poly->vertices) {
30         poly->capacity = 0;
31         return -1;
32     }
33     return 0;
34 }
35
36 void polygon_free(Polygon* poly) {
37     if (!poly) {
38         return;
39     }
40     free(poly->vertices);
41     poly->vertices = NULL;
42     poly->count = 0;
43     poly->capacity = 0;
44 }
45
46 int polygon_add_vertex(Polygon* poly, Point p) {
47     if (!poly) {
48         return -1;
49     }
50     if (poly->count == poly->capacity) {
51         size_t new_cap = (poly->capacity == 0) ? 4 : poly->capacity * 2;
52         if (polygon_reserve(poly, new_cap) != 0) {
53             return -1;
54         }
55     }
56     poly->vertices[poly->count++] = p;

```

```

57     return 0;
58 }
59
60 // Shoelace formula for polygon area.
61 // Returns an unsigned area (always >= 0).
62 double polygon_area(const Polygon* poly) {
63     if (!poly || poly->count < 3) {
64         return 0.0;
65     }
66
67     double sum = 0.0;
68     size_t n = poly->count;
69     for (size_t i = 0; i < n; ++i) {
70         const Point* a = &poly->vertices[i];
71         const Point* b = &poly->vertices[(i + 1) % n];
72         sum += a->x * b->y - b->x * a->y;
73     }
74     return 0.5 * fabs(sum);
75 }
76
77 // Perimeter: sum of edge lengths.
78 double polygon_perimeter(const Polygon* poly) {
79     if (!poly || poly->count < 2) {
80         return 0.0;
81     }
82
83     double length = 0.0;
84     size_t n = poly->count;
85     for (size_t i = 0; i < n; ++i) {
86         const Point* a = &poly->vertices[i];
87         const Point* b = &poly->vertices[(i + 1) % n];
88         double dx = b->x - a->x;
89         double dy = b->y - a->y;
90         length += sqrt(dx * dx + dy * dy);
91     }
92     return length;
93 }
94
95 // Ray-casting algorithm for point-in-polygon.
96 int polygon_contains_point(const Polygon* poly, Point p) {
97     if (!poly || poly->count < 3) {
98         return 0;
99     }
100
101     size_t n = poly->count;
102     int inside = 0;
103
104     for (size_t i = 0, j = n - 1; i < n; j = i++) {
105         const Point* pi = &poly->vertices[i];
106         const Point* pj = &poly->vertices[j];
107
108         int intersect =
109             ((pi->y > p.y) != (pj->y > p.y)) &&

```



```

110         (p.x < (pj->x - pi->x) * (p.y - pi->y)
111             / (pj->y - pi->y) + pi->x);
112
113         if (intersect) {
114             inside = !inside;
115         }
116     }
117     return inside;
118 }
119
120 // Translation: shift every vertex by (dx, dy).
121 void polygon_translate(Polygon* poly, double dx, double dy) {
122     if (!poly) {
123         return;
124     }
125     for (size_t i = 0; i < poly->count; ++i) {
126         poly->vertices[i].x += dx;
127         poly->vertices[i].y += dy;
128     }
129 }
130
131 // Rotation about an arbitrary origin (originX, originY).
132 void polygon_rotate(Polygon* poly,
133                     double angleRadians,
134                     double originX,
135                     double originY) {
136     if (!poly) {
137         return;
138     }
139
140     double c = cos(angleRadians);
141     double s = sin(angleRadians);
142
143     for (size_t i = 0; i < poly->count; ++i) {
144         double xShifted = poly->vertices[i].x - originX;
145         double yShifted = poly->vertices[i].y - originY;
146
147         double xr = xShifted * c - yShifted * s;
148         double yr = xShifted * s + yShifted * c;
149
150         poly->vertices[i].x = xr + originX;
151         poly->vertices[i].y = yr + originY;
152     }
153 }
154
155 // Scaling about an arbitrary origin.
156 void polygon_scale(Polygon* poly,
157                   double sx,
158                   double sy,
159                   double originX,
160                   double originY) {
161     if (!poly) {
162         return;

```

```

163     }
164
165     for (size_t i = 0; i < poly->count; ++i) {
166         double xShifted = poly->vertices[i].x - originX;
167         double yShifted = poly->vertices[i].y - originY;
168
169         double xs = xShifted * sx;
170         double ys = yShifted * sy;
171
172         poly->vertices[i].x = xs + originX;
173         poly->vertices[i].y = ys + originY;
174     }
175 }

```

3.3 Example Usage (C)

```

1  #include "polygon.h"
2  #include <stdio.h>
3
4  int main(void) {
5      Polygon poly;
6      if (polygon_init(&poly, 0) != 0) {
7          fprintf(stderr, "Failed to initialize polygon\n");
8          return 1;
9      }
10
11     // Simple square
12     polygon_add_vertex(&poly, (Point){0.0, 0.0});
13     polygon_add_vertex(&poly, (Point){1.0, 0.0});
14     polygon_add_vertex(&poly, (Point){1.0, 1.0});
15     polygon_add_vertex(&poly, (Point){0.0, 1.0});
16
17     printf("Area:      %f\n", polygon_area(&poly));
18     printf("Perimeter: %f\n", polygon_perimeter(&poly));
19
20     Point p = {0.5, 0.5};
21     printf("Contains (0.5, 0.5)? %s\n",
22           polygon_contains_point(&poly, p) ? "yes" : "no");
23
24     // Transformations
25     polygon_translate(&poly, 2.0, 3.0);
26     polygon_rotate(&poly, 3.141592653589793 / 4.0, 0.0, 0.0);
27     polygon_scale(&poly, 2.0, 2.0, 0.0, 0.0);
28
29     polygon_free(&poly);
30     return 0;
31 }

```

4 Notes and Extensions

- The shoelace formula implementations treat the polygon as implicitly closed by connecting vertex `i` to `(i+1) % n`.
- Orientation (clockwise vs. counter-clockwise) can be inferred from the sign of the non-`fabs` shoelace sum if needed.
- The transformation functions are intentionally minimal and side-effectful (they modify the polygon in place); if you prefer a more functional style, you can implement non-mutating versions that return new polygons.
- For more advanced features (holes, self-intersections, robust predicates, etc.), consider dedicated geometry libraries, and keep these minimal types as a learning or prototyping tool.