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`

```cpp
#ifndef POLYGON_HPP
#define POLYGON_HPP

#include <cstddef>
#include <vector>

// Basic 2D point
struct Point {
    double x{};
    double y{};
};

// Simple polygon represented as an ordered list of vertices.
// The polygon is implicitly closed: the last vertex connects back to the first.
class Polygon {
public:
    Polygon() = default;

    explicit Polygon(std::vector<Point> points);

    // Vertex management
    void addVertex(const Point& p);
    std::size_t vertexCount() const;

    // Geometric properties
    double area() const;          // Shoelace formula (unsigned)
    double perimeter() const;     // Sum of edge lengths
    bool containsPoint(const Point& p) const; // Ray-casting

    // Transformations
    void translate(double dx, double dy);
    void rotate(double angleRadians,
                double originX,
                double originY);
    void scale(double sx,
               double sy,
               double originX,
               double originY);

    // Read-only access to vertices
    const std::vector<Point>& getVertices() const;

private:
    std::vector<Point> vertices;
};

#endif // POLYGON_HPP
```

2.2 Implementation File: `Polygon.cpp`

```cpp
#include "Polygon.hpp"
#include <cmath>

Polygon::Polygon(std::vector<Point> points)
    : vertices(std::move(points)) {}

void Polygon::addVertex(const Point& p) {
    vertices.push_back(p);
}

std::size_t Polygon::vertexCount() const {
    return vertices.size();
}

// Shoelace formula for polygon area.
// Returns an unsigned area (always >= 0).
double Polygon::area() const {
    const std::size_t n = vertices.size();
    if (n < 3) {
        return 0.0;
    }

    double sum = 0.0;
    for (std::size_t i = 0; i < n; ++i) {
        const Point& a = vertices[i];
        const Point& b = vertices[(i + 1) % n]; // wrap to first
        sum += a.x * b.y - b.x * a.y;
    }
    return 0.5 * std::abs(sum);
}

// Perimeter: sum of Euclidean distances between consecutive vertices.
double Polygon::perimeter() const {
    const std::size_t n = vertices.size();
    if (n < 2) {
        return 0.0;
    }

    double length = 0.0;
    for (std::size_t i = 0; i < n; ++i) {
        const Point& a = vertices[i];
        const Point& b = vertices[(i + 1) % n];
        const double dx = b.x - a.x;
        const double dy = b.y - a.y;
        length += std::sqrt(dx * dx + dy * dy);
    }
    return length;
}

// Ray-casting algorithm for point-in-polygon.
// Returns true if the point is strictly inside or on the boundary (approximately).
```

```cpp
bool Polygon::containsPoint(const Point& p) const {
    const std::size_t n = vertices.size();
    if (n < 3) {
        return false;
    }

    bool inside = false;
    for (std::size_t i = 0, j = n - 1; i < n; j = i++) {
        const Point& pi = vertices[i];
        const Point& pj = vertices[j];

        const bool intersect =
            ((pi.y > p.y) != (pj.y > p.y)) &&
            (p.x < (pj.x - pi.x) * (p.y - pi.y) / (pj.y - pi.y) + pi.x);

        if (intersect) {
            inside = !inside;
        }
    }
    return inside;
}

// Translation: shift every vertex by (dx, dy).
void Polygon::translate(double dx, double dy) {
    for (Point& v : vertices) {
        v.x += dx;
        v.y += dy;
    }
}

// Rotation about an arbitrary origin (originX, originY).
// angleRadians is measured counter-clockwise.
void Polygon::rotate(double angleRadians,
                     double originX,
                     double originY) {
    const double c = std::cos(angleRadians);
    const double s = std::sin(angleRadians);

    for (Point& v : vertices) {
        const double xShifted = v.x - originX;
        const double yShifted = v.y - originY;

        const double xr = xShifted * c - yShifted * s;
        const double yr = xShifted * s + yShifted * c;

        v.x = xr + originX;
        v.y = yr + originY;
    }
}

// Scaling about an arbitrary origin.
// (sx, sy) are scale factors in x and y respectively.
void Polygon::scale(double sx,
```

```
                    double sy,
                    double originX,
                    double originY) {
    for (Point& v : vertices) {
        const double xShifted = v.x - originX;
        const double yShifted = v.y - originY;

        const double xs = xShifted * sx;
        const double ys = yShifted * sy;

        v.x = xs + originX;
        v.y = ys + originY;
    }
}

const std::vector<Point>& Polygon::getVertices() const {
    return vertices;
}
```

2.3 Example Usage (C++)

```
#include "Polygon.hpp"
#include <iostream>

int main() {
    // Simple triangle
    std::vector<Point> pts = {
        {0.0, 0.0},
        {1.0, 0.0},
        {0.0, 1.0}
    };

    Polygon poly(pts);

    std::cout << "Area: " << poly.area() << "\n";
    std::cout << "Perimeter: " << poly.perimeter() << "\n";

    Point p{0.25, 0.25};
    std::cout << "Contains (0.25, 0.25)? "
              << (poly.containsPoint(p) ? "yes" : "no") << "\n";

    // Translate by (1, 1)
    poly.translate(1.0, 1.0);

    // Rotate 90 degrees CCW around origin (0, 0)
    poly.rotate(3.141592653589793 / 2.0, 0.0, 0.0);

    // Uniformly scale by 2 around origin (0, 0)
    poly.scale(2.0, 2.0, 0.0, 0.0);

    return 0;
}
```

3 C Polygon API

3.1 Header File: polygon.h

```c
#ifndef POLYGON_H
#define POLYGON_H

#include <stddef.h> // size_t

// Basic 2D point
typedef struct {
    double x;
    double y;
} Point;

// Polygon represented as a dynamic array of vertices.
typedef struct {
    Point*  vertices;
    size_t  count;
    size_t  capacity;
} Polygon;

// Lifecycle
int  polygon_init(Polygon* poly, size_t initial_capacity);
void polygon_free(Polygon* poly);

// Vertex management
int polygon_add_vertex(Polygon* poly, Point p);

// Geometric properties
double polygon_area(const Polygon* poly);
double polygon_perimeter(const Polygon* poly);
int    polygon_contains_point(const Polygon* poly, Point p);

// Transformations
void polygon_translate(Polygon* poly, double dx, double dy);
void polygon_rotate(Polygon* poly,
                    double angleRadians,
                    double originX,
                    double originY);
void polygon_scale(Polygon* poly,
                   double sx,
                   double sy,
                   double originX,
                   double originY);

#endif // POLYGON_H
```

3.2 Implementation File: polygon.c

```c
#include "polygon.h"

#include <stdlib.h> // malloc, realloc, free
```

```c
#include <math.h>   // fabs, sqrt, cos, sin

static int polygon_reserve(Polygon* poly, size_t new_capacity) {
    if (new_capacity <= poly->capacity) {
        return 0;
    }
    Point* new_data =
        (Point*)realloc(poly->vertices, new_capacity * sizeof(Point));
    if (!new_data) {
        return -1;
    }
    poly->vertices = new_data;
    poly->capacity = new_capacity;
    return 0;
}

int polygon_init(Polygon* poly, size_t initial_capacity) {
    if (!poly) {
        return -1;
    }
    poly->count = 0;
    poly->capacity = (initial_capacity > 0) ? initial_capacity : 4;

    poly->vertices =
        (Point*)malloc(poly->capacity * sizeof(Point));
    if (!poly->vertices) {
        poly->capacity = 0;
        return -1;
    }
    return 0;
}

void polygon_free(Polygon* poly) {
    if (!poly) {
        return;
    }
    free(poly->vertices);
    poly->vertices = NULL;
    poly->count = 0;
    poly->capacity = 0;
}

int polygon_add_vertex(Polygon* poly, Point p) {
    if (!poly) {
        return -1;
    }
    if (poly->count == poly->capacity) {
        size_t new_cap = (poly->capacity == 0) ? 4 : poly->capacity * 2;
        if (polygon_reserve(poly, new_cap) != 0) {
            return -1;
        }
    }
    poly->vertices[poly->count++] = p;
```

```c
    return 0;
}

// Shoelace formula for polygon area.
// Returns an unsigned area (always >= 0).
double polygon_area(const Polygon* poly) {
    if (!poly || poly->count < 3) {
        return 0.0;
    }

    double sum = 0.0;
    size_t n = poly->count;
    for (size_t i = 0; i < n; ++i) {
        const Point* a = &poly->vertices[i];
        const Point* b = &poly->vertices[(i + 1) % n];
        sum += a->x * b->y - b->x * a->y;
    }
    return 0.5 * fabs(sum);
}

// Perimeter: sum of edge lengths.
double polygon_perimeter(const Polygon* poly) {
    if (!poly || poly->count < 2) {
        return 0.0;
    }

    double length = 0.0;
    size_t n = poly->count;
    for (size_t i = 0; i < n; ++i) {
        const Point* a = &poly->vertices[i];
        const Point* b = &poly->vertices[(i + 1) % n];
        double dx = b->x - a->x;
        double dy = b->y - a->y;
        length += sqrt(dx * dx + dy * dy);
    }
    return length;
}

// Ray-casting algorithm for point-in-polygon.
int polygon_contains_point(const Polygon* poly, Point p) {
    if (!poly || poly->count < 3) {
        return 0;
    }

    size_t n = poly->count;
    int inside = 0;

    for (size_t i = 0, j = n - 1; i < n; j = i++) {
        const Point* pi = &poly->vertices[i];
        const Point* pj = &poly->vertices[j];

        int intersect =
            ((pi->y > p.y) != (pj->y > p.y)) &&
```

```
            (p.x < (pj->x - pi->x) * (p.y - pi->y)
                   / (pj->y - pi->y) + pi->x);

        if (intersect) {
            inside = !inside;
        }
    }
    return inside;
}

// Translation: shift every vertex by (dx, dy).
void polygon_translate(Polygon* poly, double dx, double dy) {
    if (!poly) {
        return;
    }
    for (size_t i = 0; i < poly->count; ++i) {
        poly->vertices[i].x += dx;
        poly->vertices[i].y += dy;
    }
}

// Rotation about an arbitrary origin (originX, originY).
void polygon_rotate(Polygon* poly,
                    double angleRadians,
                    double originX,
                    double originY) {
    if (!poly) {
        return;
    }

    double c = cos(angleRadians);
    double s = sin(angleRadians);

    for (size_t i = 0; i < poly->count; ++i) {
        double xShifted = poly->vertices[i].x - originX;
        double yShifted = poly->vertices[i].y - originY;

        double xr = xShifted * c - yShifted * s;
        double yr = xShifted * s + yShifted * c;

        poly->vertices[i].x = xr + originX;
        poly->vertices[i].y = yr + originY;
    }
}

// Scaling about an arbitrary origin.
void polygon_scale(Polygon* poly,
                   double sx,
                   double sy,
                   double originX,
                   double originY) {
    if (!poly) {
        return;
```

```
    }

    for (size_t i = 0; i < poly->count; ++i) {
        double xShifted = poly->vertices[i].x - originX;
        double yShifted = poly->vertices[i].y - originY;

        double xs = xShifted * sx;
        double ys = yShifted * sy;

        poly->vertices[i].x = xs + originX;
        poly->vertices[i].y = ys + originY;
    }
}
```

3.3 Example Usage (C)

```c
#include "polygon.h"
#include <stdio.h>

int main(void) {
    Polygon poly;
    if (polygon_init(&poly, 0) != 0) {
        fprintf(stderr, "Failed to initialize polygon\n");
        return 1;
    }

    // Simple square
    polygon_add_vertex(&poly, (Point){0.0, 0.0});
    polygon_add_vertex(&poly, (Point){1.0, 0.0});
    polygon_add_vertex(&poly, (Point){1.0, 1.0});
    polygon_add_vertex(&poly, (Point){0.0, 1.0});

    printf("Area:      %f\n", polygon_area(&poly));
    printf("Perimeter: %f\n", polygon_perimeter(&poly));

    Point p = {0.5, 0.5};
    printf("Contains (0.5, 0.5)? %s\n",
           polygon_contains_point(&poly, p) ? "yes" : "no");

    // Transformations
    polygon_translate(&poly, 2.0, 3.0);
    polygon_rotate(&poly, 3.141592653589793 / 4.0, 0.0, 0.0);
    polygon_scale(&poly, 2.0, 2.0, 0.0, 0.0);

    polygon_free(&poly);
    return 0;
}
```

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.