Swinburne University of Technology

School of Science, Computing and Engineering Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	2, Basic I/O & Vector Operations
Lecturer:	Dr. Markus Lumpe

A journey of a thousand miles begins with a single step.

Lao Tsu

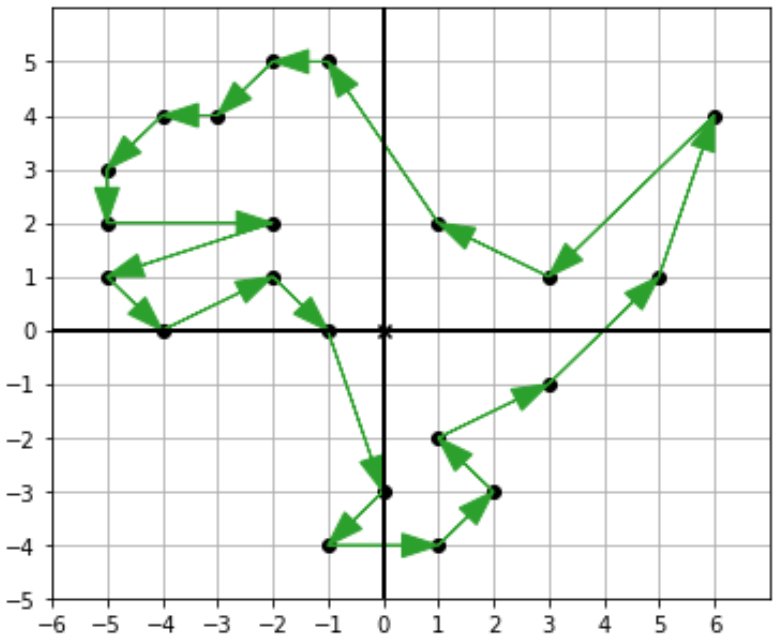


Figure 1: Vectors in the Plane Giving Rise to a Polygon.

Basic I/O and Vector Operations C++

In this tutorial, we develop a small console application that reads 2D vector data from a text file and uses this data to capture the vertices of a polygon. A 2D vector is a point in the plane relative to the origin. We can think of a 2D vector as a straight arrow in the plane as shown in Figure 2.

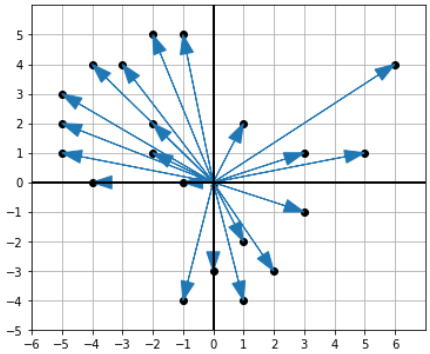


Figure 2: 2D Vectors in the Plane.

If we connect the points in Figure 2 as in Figure 3, we obtain a dinosaur polygon (or shape).

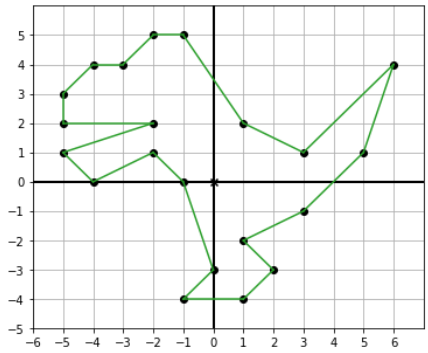


Figure 3: Dinosaur Polygon.

Many attributes and operations can be defined for polygons. In this tutorial, we focus on two:

- the perimeter of a polygon and
- a uniform scale operation for a polygon.

Additionally, we define a `readData()` method to read 2D vector values from a text file and populate polygon data.

Our application requires three components:

- a `main()` function,
- the `Vector2D` class discussed in the lectures, and
- a `Polygon` class that stores data and provides the required member functions.

The `main()` function receives the pathname of a text file containing 2D vector data as an argument. We assume the data in the text file is well-formed, that is, it contains correct 2D vector values and defines no more than `MAX_VERTICES = 30` entries.

The structure of the `main()` function follows the basic principles of applications using data I/O. We

1. check that `main()` has received the right number of arguments,
2. declare an object variable of type `std::istream` and open the corresponding input file stream,
3. check that we have successfully opened the input file,
4. read data,
5. close input file, and
6. execute the required operations.

The main() function is defined as follows:

```
#include "Polygon.h"

#include <iostream>
#include <fstream>
#include <cmath>

int main( int argc, char* argv[] )
{
  if ( argc < 2 )
  {
    std::cerr << "Arguments missing." << std::endl;
    std::cerr << "Usage: VectorOperations <filename>" << std::endl;

    return 1; // return failure, not enough arguments
  }

  // create text input stream connected to the file named in argv[1]
  std::ifstream lInput( argv[1], std::ifstream::in );

  if ( !lInput.good() )       // operation can fail
  {
    std::cerr << "Cannot open input file: " << argv[1] << std::endl;

    return 2;  // program failed (cannot open input)
  }

  Polygon lPolygon;

  lPolygon.readData( lInput );

  // close input file
  lInput.close();

  std::cout << "Data read:" << std::endl;

  for ( size_t i = 0; i < lPolygon.getNumberOfVertices(); i++ )
  {
    std::cout << "Vertex #" << i << ": " << lPolygon.getVertex( i ) << std::endl;
  }

  std::cout << "The perimeter of lPolygon is " << lPolygon.getPerimeter() << std::endl;
  std::cout << "Scale polygon by 3.2:" << std::endl;

  Polygon lScaled = lPolygon.scale( 3.2f );

  std::cout << "The perimeter of lScaled is " << lScaled.getPerimeter() << std::endl;

  float lFactor =
         std::roundf( lScaled.getPerimeter() * 100.0f / lPolygon.getPerimeter() ) / 100.0f;

  std::cout << "Scale factor: " << lFactor << std::endl;

  return 0;    // return success
}
```

The class `Polygon` is defined as follows:

```
#pragma once

#include "Vector2D.h"

constexpr size_t MAX_VERTICES = 30;

class Polygon
{
private:
  Vector2D fVertices[MAX_VERTICES];
  size_t fNumberOfVertices;

public:

  Polygon() noexcept;

  void readData( std::istream& aIStream );

  size_t getNumberOfVertices() const noexcept;
  const Vector2D& getVertex( size_t aIndex ) const;
  float getPerimeter() const noexcept;

  Polygon scale( float aScalar ) const noexcept;
};
```

The class `Polygon` depends on the class `Vector2D`. We use an array of `Vector2D` values as an internal representation for polygons. The array has a static dimension with a compile-time specified value. Please note, this is just one way to store the data. We will explore other approaches in the future.

In addition to the array `fVertices`, we need a counter to record the number of entries in the array `fVertices`. The count is a `size_t` value called `fNumberOfVertices`. We use `size_t` whenever we use unsigned integers (i.e., values greater or equal to zero).

The member functions of Polygon are defined as follows:

- The constructor `Polygon()` has to initialize all member variables with sensible values. We only need to specify the value for `fNumberOfVertices` here. The array `fVertices` is automatically initialized (via the default constructor, the constructor that takes no arguments, or the one where all arguments have a default value). Please note, that arrays in C++ are initialized automatically, if and only if the base type of the array is a class type. We exploit this language feature here. The base type of array `fVertices` is class `Vector2D`.
- The getter `getNumberOfVertices()` returns the value of `fNumberOfVertices`. The method is marked `const` to signify that this method is *read-only* and does not change any member variables.
- The getter `getVertex()` returns the 2D vector stored at `aIndex` in the array `fVertices`. It is a *read-only* method. In addition, we return a *reference* to a 2D vector rather than a value copy. This is an important idiom in C++. The location of the actual 2D Vector is at position `aIndex` in array `fVertices`. The getter `getVertex()` returns a reference to that location rather than copying the value into the caller space. This yields a very efficient data access method. The return value is marked `const` to indicate further that the caller can only read the value. Write operations are not permitted.
- The method `readData()` inputs a 2D vector from the input stream `aIStream`. We use formatted input to read the data. As a control structure, we use a **`while`**

statement. Whenever we can successfully read data for the next vector, we must increment `fNumberOfVertices`. The while loop stops when no more data can be fetched from the input, that is, the input reaches `end-of-file`.

- The function `getPerimeter()` computes the circumference of the polygon. The calculation requires a **`for`**-loop. The perimeter of the polygon is the sum of the lengths of the vectors along the polygon boundary.

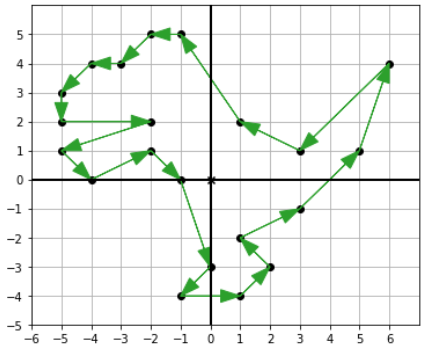


Figure 4: Vectors along the Polygon Boundary.

Starting at `i=0` the first vertex v_i, we grab the next vertex v_{i+1} and build a new vector `segment` = $v_{i+1} - v_i$. The length of the `segment` is added to the perimeter of the polygon. We need to run this loop for all elements in `fVertices`. You may use the modulus operator for the next vertex to guarantee that the value of loop variable `i` is in the interval `[0, fNumberOfVertices)`.

- The function `scale()` returns a new polygon in which the vertices (i.e., the 2D vectors) have been multiplied by the value `aScalar`. The function is *read-only*, that is, it does not alter any member variable. You should start with a copy of the polygon. Use the following technique to create a copy:

  ```
  Polygon Result = *this;
  ```

  The expression **`*this`** yields the current **`this`** object, which is used to initialize the value `Result`. Use a **`for`**-loop to multiply every 2D vector in `fVertices` by `aScalar` and assign the result to the `Result`'s `fVertices` in turn (i.e., at the same index). At the end, return the value `Result`.

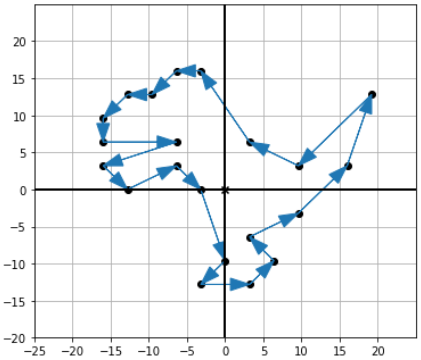


Figure 5: Polygon scaled by 3.2.

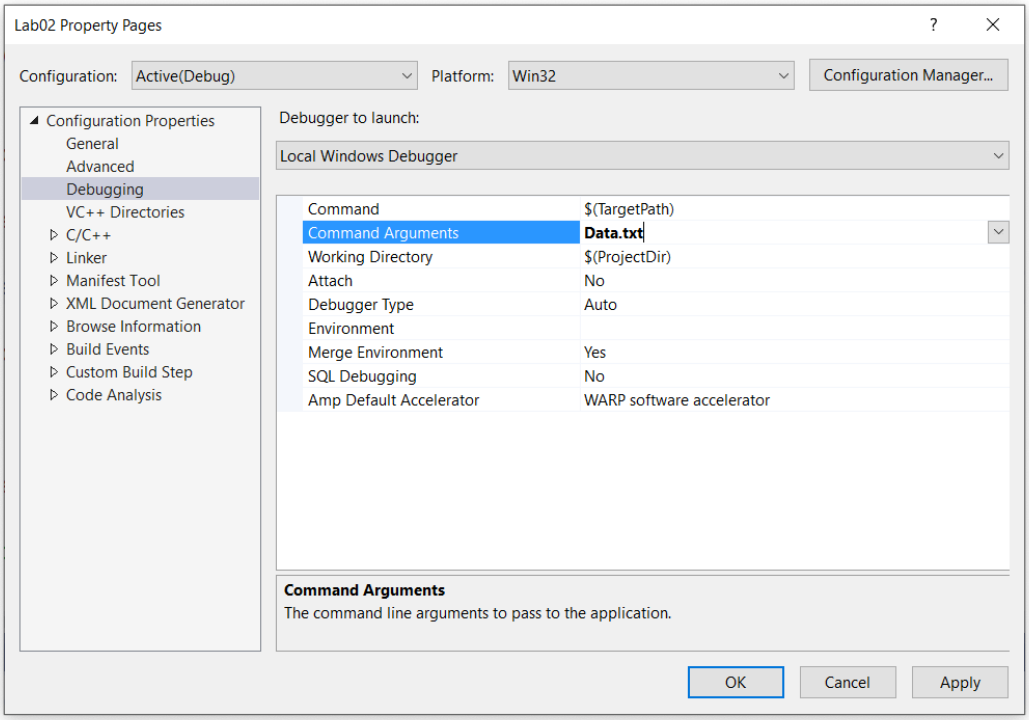
Running the application using Data.txt should produce the following output:

```
Data read:
Vertex #0: [6,4]
Vertex #1: [3,1]
Vertex #2: [1,2]
Vertex #3: [-1,5]
Vertex #4: [-2,5]
Vertex #5: [-3,4]
Vertex #6: [-4,4]
Vertex #7: [-5,3]
Vertex #8: [-5,2]
Vertex #9: [-2,2]
Vertex #10: [-5,1]
Vertex #11: [-4,0]
Vertex #12: [-2,1]
Vertex #13: [-1,0]
Vertex #14: [0,-3]
Vertex #15: [-1,-4]
Vertex #16: [1,-4]
Vertex #17: [2,-3]
Vertex #18: [1,-2]
Vertex #19: [3,-1]
Vertex #20: [5,1]
The perimeter of lPolygon is 44.75
Scale polygon by 3.2:
The perimeter of lScaled is 143.32
Scale factor: 3.2
```

Multiplying a vector by a scalar yields a vector whose length is the product of the scalar and the length of the original vector. That is, multiplying a vector by 2, yields a vector twice as long. This also applies to the perimeter of a polygon. Hence the computed scale factor matches the scalar up to some floating-point rounding error.

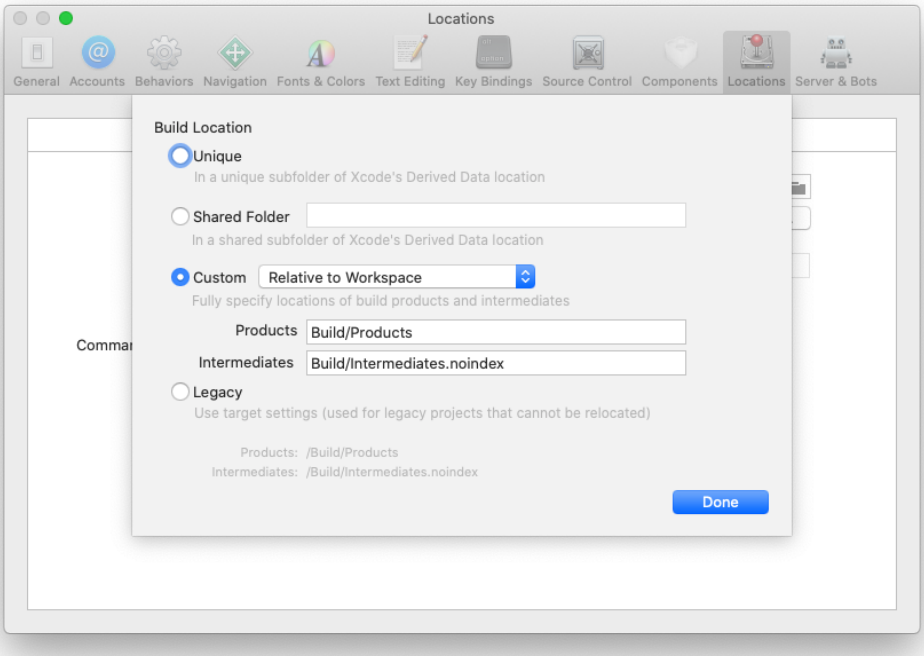
Solution Properties

- Visual Studio

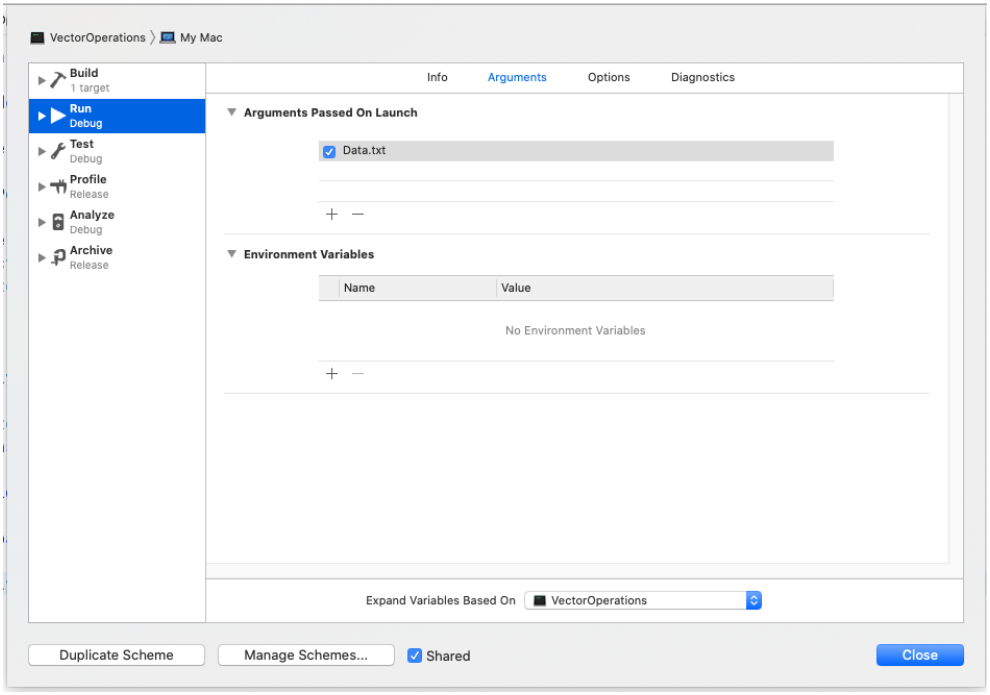


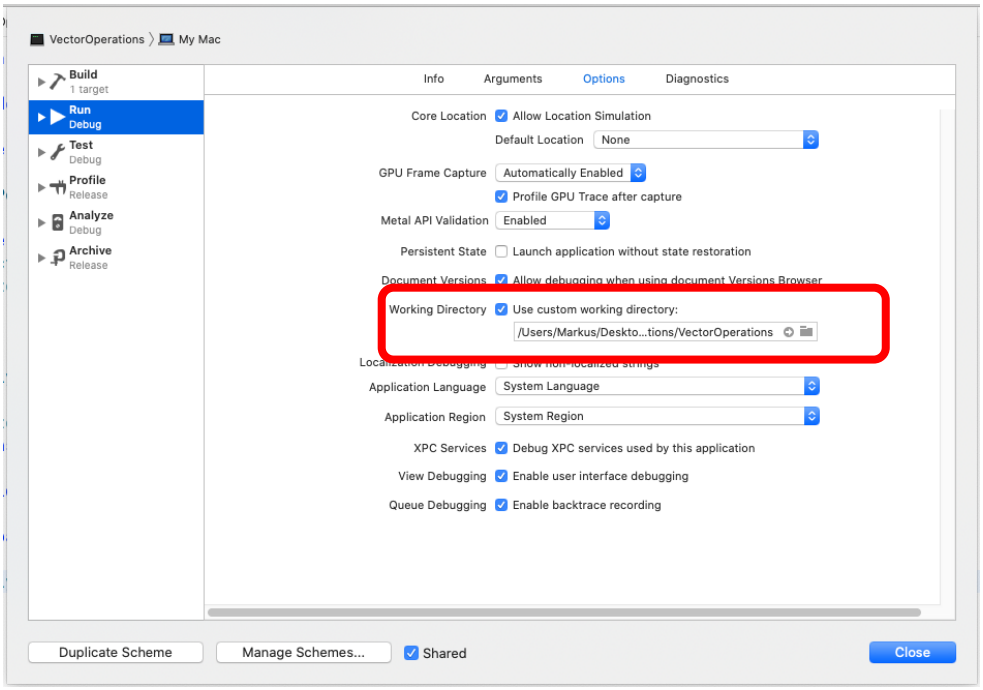
- MaxOS/XCode

  Preferences/Derived Data – Advanced Setting



Xcode users must use **edit scheme** and set a custom working directory to the location of Data.txt (in Options).





This exercise requires approximately 70 lines of low-density C++ code.