The runtime complexity of my code can be analyzed as follows:

1. **calcOblongCorner function:** The calcOblongCorner function has a time complexity of O(1) since it executes a number of operations regardless of the input size.
2. **arbitraryMinMax function:** This function operates in the manner. Its time complexity remains constant denoted as O(1). It accomplishes this by generating a number from a range.
3. **doRectanglesOverlap function:** Similar to the functions this particular one also has a time complexity that remains constant denoted as O(1). This means that it executes a predetermined number of operations without regard to the magnitude of the input.
4. **main function:** The main function has a loop that runs a number of times determined by the user's input. Inside this loop there is another loop that checks if any of the created rectangles overlap with each other. In the worst case scenario when all rectangles overlap with each other it would take a time complexity of O(n^2) where n represents the number of rectangles.
5. **Writing to file:** The process of saving the information, for each rectangle, to the file requires a time complexity of O(n) with 'n' representing the number of rectangles.

After analysis, I determined that my code's worst-case time complexity is O(n^2) due to the nested loop in the function. In contrast, the best-case scenario occurs when none of the rectangles overlap, resulting in a linear time complexity of O(n log n), where n represents the number of rectangles.

Approach
I've defined a struct named 'Oblong' to represent rectangular objects with specific attributes like width ('span'), height ('elevation'), center coordinates ('a' and 'b'), and a corner angle ('corner'). I've implemented a function called 'calcOblongCorner' to calculate the corner angle of an Oblong using trigonometric calculations. Additionally, I've created a function named 'arbitraryMinMax' for generating random double values within a specified range and 'doRectanglesOverlap' to check if two rectangles overlap. Within the main function, I prompt the user to input the number of rectangles they want to generate and seed the random number generator for ensuring randomness. I use a vector named 'oblong' to store the generated Oblong objects. Inside a loop, I generate the specified number of rectangles, ensuring that they don't overlap with existing ones. If an overlap is detected, I regenerate the rectangle. I also create a CSV file named "anglesForRectangle.csv" to store the properties of the non-overlapping rectangles. If the file is successfully opened, I write the rectangle data to it. Finally, I calculate the total occlusion angle over all rectangles by summing the corner angles of each rectangle and display the result to the user. This code combines random number generation, geometry calculations, and file handling to analyze and store information about non-overlapping rectangles.