ENPM702: Assignment #3

v2.0

Due Date: 07/26/2025

Points: 30 pts

Robot Control System with Functions

Contents

1 Changelog

- v2.0:
  - Updated the due date.
  - Updated section 7.1.2: Version #2 has been modified to take an `std::pair` as argument.
  - Updated section 7.1.3: Added more examples and included edge cases.
  - Updated section 7.3.1 (`find_extremes`): Added more examples.
  - Updated the `main()` function (section 7.4) and output format (section 8.
    - I was hoping you would fix the awful code in the main function but I decided to do it so you can focus on the main topic of the assignment.
- v1.0: Original version.

2 Guidelines

This assignment is to be completed **individually**, and all instructions must be followed carefully. Not adhering to these guidelines may result in a zero for the assignment.

- Do not use solutions or components created by others.
- Keep your work private; do not share your files or code.
- Any GitHub repository used for development must be private.
- Submit your completed work as a zipped file via Canvas.
- Feel free to discuss general concepts with classmates, but sharing specific implementation details is strictly forbidden.
- The use of AI-generated code (e.g., ChatGPT, Copilot) is not permitted.

For this assignment, you must organize your code using proper function declarations and definitions. All function declarations must be placed in header files (*.hpp) and definitions in source files (*.cpp). You must demonstrate proper use of function overloading, default parameters, and const-correctness.

3 Overview

You will develop a C++ program that implements a **Robot Control System** for an autonomous cleaning robot. This assignment focuses on applying function fundamentals including declarations vs definitions, parameter passing mechanisms, function overloading, default parameters, and proper return value handling.

The robot performs various tasks:

- **Navigation** - Calculate distances and validate movement paths
- **Cleaning** - Track cleaning efficiency and battery usage
- **Sensor Processing** - Analyze environmental data for decision making
- **Status Reporting** - Generate comprehensive system reports

4 Learning Objectives

By completing this assignment, you will demonstrate proficiency in:

- **Function Organization**: Proper separation of declarations and definitions using header files.
- **Parameter Passing**: Pass-by-value, pass-by-reference, and pass-by-const-reference.
- **Function Overloading**: Multiple functions with same name but different parameters.
- **Default Parameters**: Functions with optional parameters and proper declaration practices.
- **Const-correctness**: Proper use of `const` in function parameters and return types.
- **Documentation**: Professional function documentation using Doxygen.
- **Return Types and Values**: Proper return type usage and value handling.

5 Scenario

You are developing a control system for an autonomous cleaning robot operating in an office building. The robot must navigate between rooms, monitor its cleaning efficiency, process sensor data, and generate detailed reports. Your system must implement various mathematical calculations, data validation, and status tracking using proper function design principles.

6 Technical Requirements

6.1 Project Setup

Retrieve the starter project rwa3_enpm702_summer_2025 from GitHub. The `main()` function is already provided. You will create additional *.hpp and *.cpp files as rewuired.

- Configure CMakeLists.txt to compile with C++17, enabling all warnings (`-Wall`) and strict compliance (`-pedantic-errors`
- Use `include_directories(include)` to specify the header files location.
- Generate the executable from multiple *.cpp files (as shown in Lecture 5).

7 Implementation Requirements

7.1 Navigation Module

7.1.1 Distance Calculations (Function Overloading)

Implement two overloaded functions named **`calculate_distance`**:

Version #1: Accept four double parameters representing x1, y1, x2, y2 coordinates

- Calculate and return the Euclidean distance between two 2D points
- Use the standard distance formula: $\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$

Version #2: Accept two vector parameters, each containing {x, y} coordinates (consider using **`std::pair`**, **`std::array`**, or **`std::vector`**):

- Extract coordinates from the vectors and calculate distance.
- Should call Version #1 internally to avoid code duplication.
- Use **`const`**-reference parameters for efficiency.

7.1.2 Coordinate Validation

Implement two overloaded functions named **`is_valid_coordinate`**:

Version #1: Single coordinate validation:

- Accept a coordinate value and boundary parameters.
- Include default parameters: minimum bound (0.0) and maximum bound (100.0).
- Return true if coordinate is within bounds, false otherwise.
  - valid = (min_bound ≤ coordinate ≤ max_bound)
- **Examples**:
  - **`is_valid_coordinate(50.0)`** → true (uses defaults: 0.0 ≤ 50.0 ≤ 100.0)
  - **`is_valid_coordinate(-5.0)`** → false (-5.0 < 0.0)
  - **`is_valid_coordinate(150.0)`** → false (150.0 > 100.0)
  - **`is_valid_coordinate(75.0, 10.0, 80.0)`** → true (10.0 ≤ 75.0 ≤ 80.0)
  - **`is_valid_coordinate(5.0, 10.0, 80.0)`** → false (5.0 < 10.0)

Version #2: 2D point validation:

- Accept a pair containing x and y coordinates plus boundary parameters.
- Function signature:

```
bool is_valid_coordinate(const std::pair<double, double>& point, double min_bound = 0.0, double max_bound = 100.0);
```

- Use the same default parameters as Version #1.
- Return true if both coordinates are valid.
  - valid = (min_bound ≤ point.first ≤ max_bound) AND (min_bound ≤ point.second ≤ max_bound)
- Should call Version #1 internally to validate each coordinate.
- **Examples**:
  - `is_valid_coordinate({25.0, 75.0})` → true (both 25.0 and 75.0 are in [0.0, 100.0])
  - `is_valid_coordinate({-10.0, 50.0})` → false (x = -10.0 < 0.0)
  - `is_valid_coordinate({50.0, 120.0})` → false (y = 120.0 > 100.0)
  - `is_valid_coordinate({30.0, 40.0}, 20.0, 60.0)` → true (both coordinates in [20.0, 60.0])
  - `is_valid_coordinate({10.0, 40.0}, 20.0, 60.0)` → false (x = 10.0 < 20.0)

7.1.3 Distance Calculation

Implement `calculate_total_distance`:

- Accept a vector of coordinate pairs representing waypoints.
- Calculate the cumulative distance between consecutive points.
- Return the total distance as a double.
- For waypoints $P_0, P_1, P_2, \ldots, P_n$:

$$\text{Total Distance} = \sum_{i=0}^{n-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \tag{1}$$

- Use the `calculate_distance` function internally to avoid code duplication.
- Return 0.0 if the vector has fewer than 2 waypoints.

Example Usage:

- **Example 1 - Multiple waypoints:**
  - Input: `std::vector<std::pair<double, double>> waypoints{{0.0, 0.0}, {3.0, 4.0}, {6.0, 8.0}};`
  - Calculation:
    - Distance from (0.0, 0.0) to (3.0, 4.0): $\sqrt{3^2 + 4^2} = 5.0$
    - Distance from (3.0, 4.0) to (6.0, 8.0): $\sqrt{3^2 + 4^2} = 5.0$
  - Result: `calculate_total_distance(waypoints)` returns 10.0
- **Example 2 - Complex path:**
  - Input: `std::vector<std::pair<double, double>> path{{0.0, 0.0}, {5.0, 0.0}, {5.0, 12.0}, {10.0, 15.0}, {12.5, 15.0}};`
  - Calculation:
    - (0.0, 0.0) to (5.0, 0.0): 5.0 units
    - (5.0, 0.0) to (5.0, 12.0): 12.0 units
    - (5.0, 12.0) to (10.0, 15.0): $\sqrt{25 + 9} \approx 5.83$ units
    - (10.0, 15.0) to (12.5, 15.0): 2.5 units
  - Result: Total distance ≈ 25.3 units
- **Example 3 - Edge cases:**
  - Empty vector: returns 0.0
  - Single waypoint `{{5.0, 5.0}}`: returns 0.0
  - Two identical waypoints `{{3.0, 4.0}, {3.0, 4.0}}`: returns 0.0

7.2 Cleaning Module

7.2.1 Efficiency Calculations

Implement `calculate_efficiency`:

- Accept area cleaned and total area as parameters.
- Calculate and return efficiency as a percentage (0.0 to 100.0).

$$\text{Efficiency (\%)} = \frac{\text{Area Cleaned}}{\text{Total Area}} \times 100 \tag{2}$$

- Handle edge case where total area is zero.

7.2.2 Status Update

Implement `update_status`:

- Accept references to battery level and total area cleaned (to be modified).
- Accept session battery consumption and session area cleaned (by value).
- Update the totals appropriately (subtract battery consumption, add area cleaned).
- Use pass-by-reference to modify the original variables.

Mathematical Operations:

$$\text{New Battery Level} = \text{Current Battery} - \text{Session Consumption} \tag{3}$$

$$\text{New Total Area} = \text{Current Total} + \text{Session Area} \tag{4}$$

Example Usage:

- Initial state: battery = 85.0%, total_area = 120.0 m^2.
- Session data: consumed 15.0% battery, cleaned 25.0 m^2.
- Call: `update_status(battery, total_area, session_consumption, session_area)`.
- Result: battery = 70.0%, total_area = 145.0 m^2.

Safety Considerations:

- Ensure battery level doesn't go below 0.0%.
- Area cleaned should always be non-negative.

7.2.3 Battery Management (Function Overloading)

Implement two overloaded functions named `estimate_battery_life`:

Version #1: Default consumption calculation:

- Accept current battery percentage.
- Include a default parameter for consumption rate (2.0% per minute).
- Return estimated minutes of operation remaining.

Version #2: Mode-based calculation:

- Accept current battery percentage and cleaning mode string.
- Define consumption rates: "eco" (1.5%/min), "normal" (2.0%/min), "turbo" (3.5%/min).
- Return estimated minutes based on the selected mode.
- Handle invalid mode strings with a default rate.

Mathematical Operations:

$$\text{Battery Life (minutes)} = \frac{\text{Current Battery \%}}{\text{Consumption Rate (\%/min)}} \tag{5}$$

Example Usage Version 1:

- Current battery: 75.0%.
- Call: `estimate_battery_life(75.0)`.
- Result: 37.5 minutes (using default 2.0%/min rate).

Example Usage Version 2:

- Current battery: 60.0%.

- Eco mode call: `estimate_battery_life(60.0, "eco")`.
- Result: 40.0 minutes ($\frac{60.0}{1.5}$).
- Turbo mode call: `estimate_battery_life(60.0, "turbo")`.
- Result: 17.1 minutes ($\frac{60.0}{3.5}$).

7.3 Utils Module

7.3.1 Data Processing Functions

Implement `calculate_average`:

- Accept a vector of numeric values.
- Calculate and return the arithmetic mean.
- Return 0.0 if the vector is empty.

Mathematical Operations:

$$\text{Average} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{6}$$

Example Usage:

- Input vector: `cleaning_efficiency_data={85.5, 92.0, 78.3, 88.7, 91.2}`.
- Call: `calculate_average(cleaning_efficiency_data)`.
- Result: 87.14 (sum = 435.7, count = 5).
- Empty vector call: `calculate_average(empty_vector)`.
- Result: 0.0.

Implement `find_extremes`:

- Accept a vector of values (const reference) and two reference parameters for storing results.
- Find the minimum and maximum values in the vector.
- Store the minimum value in the `min_val` reference parameter.
- Store the maximum value in the `max_val` reference parameter.
- Return `true` if successful, `false` if the vector is empty.
- If the vector is empty, leave the reference parameters unchanged.

Example Usage:

- **Example 1 - Normal case:**
  - Input: `std::vector<double> battery_readings{12.5, 8.3, 15.7, 4.9, 11.2};`
  - Before: `double min_val{};` (`min_val = 0.0`) and `double max_val{};` (`max_val = 0.0`).
  - Call: `bool result{find_extremes(battery_readings, min_val, max_val)};`
  - Result: `result = true`, `min_val = 4.9`, `max_val = 15.7`
- **Example 2 - Empty vector:**
  - Input: `std::vector<double> battery_readings{};`
  - Before: `double min_val{};` (`min_val = 0.0`) and `double max_val{};` (`max_val = 0.0`).
  - Call: `bool result{find_extremes(battery_readings, min_val, max_val)};`
  - Result: `result = false`, `min_val = 0.0` (unchanged), `max_val = 0.0` (unchanged)
- **Example 3 - Single element:**
  - Input: `std::vector<double> battery_readings{42.7};`
  - Before: `double min_val{};` (`min_val = 0.0`) and `double max_val{};` (`max_val = 0.0`).
  - Call: `bool result{find_extremes(battery_readings, min_val, max_val)};`
  - Result: `result = true`, `min_val = 42.7`, `max_val = 42.7`

7.3.2 Display Functions

Implement `display_status`:

- Accept current position (x, y), battery level, and efficiency as parameters.
- Display formatted robot status information to console.
- Use appropriate formatting for numerical values.

Example Usage:

- Call: `display_status(12.5, 8.3, 85.7, 92.4)`.
- Console output (numbers may vary):

```
=== Robot Status ===
Position: (12.5, 8.3)
Battery Level: 85.7%
Cleaning Efficiency: 92.4%
===================
```

Formatting Considerations:

- Use fixed-point notation with 1 decimal place for positions.
- Display battery and efficiency as percentages with 1 decimal place.
- Include appropriate units and labels for clarity.

7.4 Main Program

Your `main()` function should demonstrate all implemented functionality. A minimal `main()` function is shown below.

```cpp
int main() {
    // Navigation Testing
    std::cout << "=== NAVIGATION TESTING ===\n";

    // Distance calculation tests
    const double distance_coords{calculate_distance(0.0, 0.0, 3.0, 4.0)};
    const auto point1 = std::make_pair(0.0, 0.0);
    const auto point2 = std::make_pair(5.0, 12.0);
    const double distance_points{calculate_distance(point1, point2)};

    // Coordinate validation tests
    is_valid_coordinate(50.0);
    is_valid_coordinate(-5.0);
    is_valid_coordinate(75.0, 10.0, 80.0);
    is_valid_coordinate({25.0, 75.0});
    is_valid_coordinate({50.0, 120.0});
    is_valid_coordinate({30.0, 40.0}, 20.0, 60.0);

    // Total distance calculation
    const std::vector<std::pair<double, double>> waypoints{
        {0.0, 0.0}, {5.0, 0.0}, {5.0, 12.0}, {10.0, 15.0}, {12.5, 15.0}
    };
    const double total_distance{calculate_total_distance(waypoints)};

    // Cleaning System Testing
    std::cout << "\n=== CLEANING SYSTEM TESTING ===\n";

    // Efficiency calculation
    const double cleaning_efficiency{calculate_efficiency(75.0, 150.0)};

    // Battery life estimation - both versions
    const double battery_life_default{estimate_battery_life(60.0)};
    const double battery_life_eco{estimate_battery_life(60.0, "eco")};
    const double battery_life_turbo{estimate_battery_life(60.0, "turbo")};

    // Status update
    double current_battery{100.0};  // NOT const - modified by update_status
    double total_cleaned_area{0.0};  // NOT const - modified by update_status
    update_status(current_battery, total_cleaned_area, 25.0, 80.0);

    // Utils Testing
    std::cout << "\n=== UTILS TESTING ===\n";

    // Average calculation
    const std::vector<double> efficiency_data{85.5, 92.0, 78.3, 88.7, 91.2};
    const double average_efficiency{calculate_average(efficiency_data)};

    // Find extremes
    double min_value{};  // NOT const - may be modified by find_extremes
    double max_value{};  // NOT const - may be modified by find_extremes
    const bool extremes_found{find_extremes(efficiency_data, min_value, max_value)};

    // Test with empty vector
    const std::vector<double> empty_data{};
    double min_empty{};  // NOT const - may be modified by find_extremes
    double max_empty{};  // NOT const - may be modified by find_extremes
    const bool extremes_empty{find_extremes(empty_data, min_empty, max_empty)};

    // Test with non-empty vector (different initial values)
    const std::vector<double> sensor_data{-3.2, 7.8, -1.5, 9.4, 2.1};
    double min_sensor{};  // NOT const - may be modified by find_extremes
    double max_sensor{};  // NOT const - may be modified by find_extremes
    const bool extremes_sensor{find_extremes(sensor_data, min_sensor, max_sensor)};

    // Display status
    display_status(12.5, 8.3, 85.7, 92.4);
}
```

The numbers used in the example above are for demonstration purpose. Ensure all your functions work with different numbers.

8 Expected Output Format

Your program should produce clear, well-formatted output demonstrating all functionality. You are free to update the main function to accomplish the following output.

Complete Program Output:

```
=== NAVIGATION TESTING ===
Distance (coordinates): 5.0 units
Distance (points): 13.0 units
Coordinate validation: 50.0 in [0.0, 100.0] - Valid
Coordinate validation: -5.0 not in [0.0, 100.0] - Invalid
Coordinate validation: 75.0 in [10.0, 80.0] - Valid
2D Point validation: {25.0, 75.0} in [0.0, 100.0] - Valid
2D Point validation: {50.0, 120.0} not in [0.0, 100.0] - Invalid
2D Point validation: {30.0, 40.0} in [20.0, 60.0] - Valid
Total waypoint distance: 25.3 units

=== CLEANING SYSTEM TESTING ===
Cleaning efficiency: 50.0%
Battery life (default mode): 30.0 minutes
Battery life (eco mode): 40.0 minutes
Battery life (turbo mode): 17.1 minutes
Status update - Before: Battery=100.0%, Area=0.0 m2
Status update - After: Battery=75.0%, Area=80.0 m2

=== UTILS TESTING ===
Data analysis for values: {85.5, 92.0, 78.3, 88.7, 91.2}
Average: 87.1
Min/Max found: true - Min=78.3, Max=92.0
Empty vector test: false - Min=0.0 (unchanged), Max=0.0 (unchanged)
Sensor data test: true - Min=-3.2, Max=9.4

=== Robot Status ===
Position: (12.5, 8.3)
Battery Level: 85.7%
Cleaning Efficiency: 92.4%
==================

All functionality tested successfully.
```

Output Requirements:

- Clear section headers separating different functionality tests.
- Numerical results displayed with appropriate precision (1 decimal place).
- Before/after states shown for functions that modify parameters.
- Formatted status display with consistent units and labels.
- Success/validation messages confirming proper operation.

9 Documentation Requirements

All functions must be documented using Doxygen format:

- Brief description of what the function does.
- Parameter descriptions with types and purposes.
- Return value description.
- Any special notes about behavior or constraints.

Place all documentation in the header files with function declarations.
There is no need to generate the documentation (HTML files) for this assignment.

10 Submission Requirements

- Complete the implementation across all required files.
- Ensure proper separation of declarations (*.hpp) and definitions (*.cpp).
- Configure CMakeLists.txt for proper compilation.
- Document all functions using Doxygen format.
- Demonstrate all required functionality in `main()`.
- Zip your complete project as rwa3_firstname_lastname.zip and upload to Canvas.

11 Grading Rubric (30 pts)

Ensure you refer to the C++ Core Guidelines during development.

- **Function Organization and Headers** (8 pts)
  - **Full Points:** Perfect separation of declarations/definitions; proper header organization; correct CMakeLists.txt configuration; comprehensive Doxygen documentation.
  - **Partial Points:** Good organization with minor issues; basic documentation present.
  - **No Points:** Poor file organization; missing headers; inadequate documentation.
- **Function Overloading** (7 pts)
  - **Full Points:** Correct implementation of all overloaded functions; proper parameter differentiation.
  - **Partial Points:** Most overloads work correctly with minor issues.
  - **No Points:** Incorrect or missing function overloading; compilation errors.
- **Parameter Passing and Const-correctness** (7 pts)
  - **Full Points:** Proper use of pass-by-value, pass-by-reference, and pass-by-const-reference; excellent const-correctness throughout.
  - **Partial Points:** Good parameter passing with some const-correctness issues.
  - **No Points:** Incorrect parameter passing; poor or missing const usage.
- **Default Parameters** (4 pts)
  - **Full Points:** Correct default parameter implementation; proper declaration placement; good usage examples.
  - **Partial Points:** Basic default parameters with minor placement issues.
  - **No Points:** Incorrect or missing default parameter implementation.
- **Code Quality and Output** (4 pts)
  - **Full Points:** Professional output formatting; clean code organization; comprehensive testing in `main()`; proper return type usage.
  - **Partial Points:** Good formatting with minor presentation issues.
  - **No Points:** Poor formatting; inadequate testing; disorganized code.

11.1 Additional Notes

- All functions must compile without errors using `-Wall -pedantic-errors` flags.
- Focus on demonstrating modern C++17 best practices and const-correctness.
- Use meaningful variable names and clear code organization.
- Provide comprehensive testing of all functionality in `main()`.
- The assignment should demonstrate mastery of core function design principles in a manageable scope.