COP 4338 Assignment 4: Space Mission Control System

📑 Table of Contents

📢 Important Note!

Please ensure that your implementation works on the course-designated ocelot server: ocelot-bbhatkal.aul.fiu.edu . Claims such as **"it works on my machine"** will not be considered valid, as all testing and grading will be conducted in that environment.

⚠️ **Important:** You are provided with an autograder to support consistent testing and maintain the highest level of grading transparency. The same autograder will be used by the instructor for final evaluation.

If your implementation passes all test cases using the autograder, you are likely to receive full credit for your submission.

⚠️ **There is only one submission - RESUBMISSION IS NOT ALLOWED. You must test your implementation thoroughly with the provided testing framework on the Ocelot server before the final submission.**

---

1. 📧 TA Contact Information

For any questions or concerns related to assignment grading, please reach out to the TA listed below.

- Ensure that all communication is polite and respectful
- You must contact the TA **first** for any grading issues. If the matter remains unresolved, feel free to reach out to me

Name: Sanskar Lohani — sloha002@fiu.edu
Sections: COP 4338-RVG (88613), COP 4338-U01 (84664)

---

2. 📖 Assignment Overview

This assignment provides hands-on experience with **defensive programming in C**,and **advanced C programming concepts**. You will implement **6 core functions** for a **space mission control system** that demonstrates **dynamic memory allocation**, **structures and enumerations**, **file I/O operations**, **memory management techniques**, and **defensive programming principles** essential for systems programming.

The system simulates real-world mission management scenarios commonly found in aerospace systems, embedded control systems, and enterprise mission-critical applications. **The instructor-provided framework handles all input/output, formatting, and testing** - your job is to focus purely on implementing the core memory management and data structure logic.

This assignment lays the foundation for understanding how real-world systems defend themselves using defensive programming techniques, utilize structures as data transfer containers, perform file operations, and manage dynamic data and memory. The dynamic memory allocation and structure manipulation techniques you're implementing here are the backbone of every operating system, database engine, and mission-critical application running today, where reliability and memory efficiency are paramount.

☑️**Course Modules Covered:**

- **Module-1**
- **Module-2**
- **Module-3**
- **Module-4**
- **Module-5**

---

3. 🎯 Learning Objectives

By completing this assignment, you will be able to:

1. **Master dynamic memory management** using `malloc()`, `calloc()`, `realloc()`, and `free()` for runtime memory allocation and deallocation.
2. **Design and manipulate complex data structures** including nested structures and dynamically allocated arrays for managing multi-level data.
3. **Implement robust file I/O operations** with `fscanf()` and `fprintf()` for data persistence and professional output generation.
4. **Apply defensive programming principles** with comprehensive input validation, null pointer checks, and error handling mechanisms.
5. **Prevent memory leaks** through proper cleanup and multi-level memory deallocation techniques.
6. **Utilize enumerations for type safety** to define clear, maintainable categories and ensure type-safe control flows.

---

4. ⚙️ Assignment Tasks

What You Must Implement ?

You will implement **6 functions** that collectively create a **space mission control system**. The instructor-provided framework handles all formatting, testing, and integration.

Core System Capabilities:

- Manages **dynamic mission arrays** with automatic capacity expansion
- Handles **mission creation** with crew allocation and communication systems
- Stores mission data with **proper memory management** for nested structures
- Processes **communication logs** with timestamp validation and priority levels
- Provides **file I/O capabilities** for mission data persistence and reporting
- Implements **comprehensive cleanup** to prevent memory leaks

6 Required Functions:

1. `create_mission_control()` - Dynamic system initialization with memory allocation
2. `create_mission_with_crew()` - Complex structure creation with nested array allocation
3. `add_communication()` - Dynamic array expansion and data insertion
4. `load_missions_from_file()` - File I/O with fscanf and data validation
5. `save_mission_report()` - Formatted output generation with fprintf
6. `free_mission_control()` - Multi-level memory deallocation and cleanup

📋 Detailed Function Specifications

Function 1: `create_mission_control(int initial_capacity)`

Purpose: Initialize the mission control system using dynamic memory allocation.

Requirements:

- Master dynamic memory allocation using `malloc()`
- Understand structure initialization and implement proper error handling
- Apply defensive programming to handle memory allocation failures gracefully
- Use the provided **skeleton code** as a reference

Integration Role: This is the foundational function that sets up the system framework for all subsequent operations.

Function 2: `create_mission_with_crew(MissionControl* system, int mission_id, const char* name, const char* launch_date)`

Purpose: Create a new mission entry with crew details, incorporating comprehensive validation and dynamic array management.

Requirements:

- Apply multi-parameter input validation with type-safe handling

- Implement dynamic array expansion using `realloc()`
- Allocate and initialize nested structures using `calloc()`
- Utilize provided validation functions for date format checking
- Use the provided **skeleton code** as a guide for your implementation

Integration Role: Enables the system to dynamically create and manage mission entries with proper memory allocation for crew and communication systems.

Function 3: `add_communication(MissionControl* system, int mission_id, const char* timestamp, const char* message, Priority priority)`

Purpose: Add communication logs to missions with dynamic array expansion and priority-based validation.

Requirements:

- Locate target mission using mission_id with validation
- Implement dynamic array expansion using `realloc()` when capacity is reached
- Validate timestamp format and priority enumeration values
- Use safe string operations with boundary checks
- Use the provided **skeleton code** as a reference

Integration Role: Provides communication logging capabilities with automatic capacity management for mission tracking.

Function 4: `load_missions_from_file(MissionControl* system, const char* filename)`

Purpose: Read and parse mission data from external files using `fscanf()` with comprehensive validation.

Requirements:

- Open and validate file accessibility with proper error handling
- Parse structured mission data using `fscanf()` format strings
- Validate all loaded data (mission IDs, dates, crew counts)
- Handle file I/O errors and malformed data gracefully
- Use the provided **skeleton code** as a guide

Integration Role: Enables data persistence and bulk mission loading from configuration files.

Function 5: `save_mission_report(MissionControl* system, const char* filename)`

Purpose: Generate professionally formatted mission reports using `fprintf()` with statistical analysis.

Requirements:

- Create/overwrite output files using proper file modes
- Format output using `fprintf()` with alignment and spacing

- Calculate and display mission statistics (total missions, crew counts, communication logs)
- Implement proper error handling for file operations
- Use the provided **skeleton code** as a reference

Integration Role: Provides reporting capabilities for mission status documentation and analysis.

Function 6: `free_mission_control(MissionControl* system)`

Purpose: Perform complete multi-level memory cleanup to prevent memory leaks.

Requirements:

- Implement proper deallocation order (nested structures before parent structures)
- Free all dynamically allocated arrays (missions, crew, communications)
- Handle NULL pointers gracefully throughout cleanup process
- Set freed pointers to NULL to prevent dangling references
- Use the provided **skeleton code** as a guide

Integration Role: Ensures proper resource cleanup and prevents memory leaks when system shutdown is required.

---

5. 🏗️ Project Framework

1) Provided Framework Infrastructure

You will receive a complete development framework including:

1. **Framework files**: `driver.c`, `mission_control.h`, `Makefile`, skeleton implementations, and sample executable `A4_sample`
2. **Your implementation files**: `mission_control.c`, `communication.c`, `file_io.c`, `memory_mgmt.c`
3. **Testing tools**: `autograder_space_mission_system.sh`, `batchgrader_space_mission_system.sh` `TESTCASES.txt`, `EXPECTED_OUTPUT.txt`

⚠️ **Critical Testing Information:** **The instructor will use different and more rigorous test cases for final grading that are not provided to students. However, if you pass all provided test cases with 100% accuracy, you are highly likely to pass the instructor's final grading test cases.**

Comprehensive Development Workflow

> ⚠️ **Important**: **Please refer to the document A4_Testing_Guide.pdf for the detailed testing instructions.**

Step 1: Understand Expected Output Format

⚠️ The file `TESTCASES.txt` must be present in the directory where `A4_sample` is located.

```
# Generate expected output from instructor sample
./A4_sample > EXPECTED_OUTPUT.txt

# View the expected output format to understand correct behavior
cat EXPECTED_OUTPUT.txt
```

Step 2: Implement with Incremental Testing

- **Test Each Function Individually:**
  - Implement one function at a time
  - Test thoroughly before moving to the next
  - Use the autograder after each function implementation

Step 3: Use Autograders for Final Validation

- ⚠️ **It is very importat to test your implelentation with both the autograders before the final submission**
- ⚠️ **Important**: **Please note that instructor will use the same autograders for the final grading with different test cases.**

🤖 Autograder System

The autograder compares your program's output with pre-provided reference output to determine correctness.

Internal Process:

1. **Clean rebuild**: `make clean && make`
2. **Runs your implementation**: `./space_mission > STUDENT_OUTPUT.txt`
3. **Compares outputs**: Extracts each data structure section and compares with expected results
4. **Calculates scores**: Awards points based on percentage of matching output lines
5. **Provides detailed feedback**: Shows exactly which lines don't match

⚠️ *IMPORTANT: Final Grading Notice*

🚨 **The instructor will use the SAME autograder script for final grading, but may include additional edge case testing.**

Your implementation must:

- ✅ **Pass the provided autograder with 100/100 points**
- ✅ **Handle all edge cases** correctly (empty arrays, null pointers, boundary conditions)
- ✅ **Produce deterministic output** for identical inputs
- ✅ **Compile cleanly** with the provided Makefile

---

6. 🔧 Development & Testing

Recommended Function Implementation Order

Follow this sequence to build your implementation incrementally:

1. **Start with** `create_mission_control()` - Foundation memory allocation (10 points)
2. **Implement** `create_mission_with_crew()` - Core mission creation logic (25 points)
3. **Continue with** `add_communication()` - Dynamic array operations (20 points)
4. **Implement** `load_missions_from_file()` - File I/O operations (15 points)
5. **Add** `save_mission_report()` - Report generation (10 points)
6. **Finish with** `free_mission_control()` - Memory cleanup (10 points)

Memory Management Best Practices

Critical Memory Safety Guidelines:

- Always check if `malloc()`, `calloc()`, or `realloc()` return NULL before using the pointer
- Free all dynamically allocated memory in reverse order of allocation
- Set freed pointers to NULL to avoid dangling pointer references
- **Use `valgrind` to detect memory leaks:** `valgrind --leak-check=full ./space_mission`
- Never access freed memory or memory beyond allocated bounds

Common Memory Errors to Avoid:

- Memory leaks (forgetting to free allocated memory)
- Double free errors (freeing the same memory twice)
- Use-after-free errors (accessing memory after it has been freed)
- Buffer overflows (writing beyond allocated memory boundaries)
- Dereferencing NULL pointers without validation

---

7. 📤 Assignment Submission Guidelines

- All assignments **must be submitted via Canvas** by the specified deadline as a **single compressed (.ZIP) file**. Submissions by any other means **WILL NOT** be accepted and graded!
- The filename must follow the format: `A4_FirstName_LastName.zip` (e.g., **A4_Harry_Potter.zip**)
- ⚠️ **Important:** Include the Firstname and Lastname of the **team's submitting member**, or your own name if **submitting solo**.
- ⚠️ **Important:** For **group submissions**, each group must designate one **corresponding member**. This member will be responsible for submitting the assignment and will serve as the primary point of contact for all communications related to that assignment.
- ⚠️ **Important Policy on Teamwork and Grading**

All members of a team will receive the **same grade** for collaborative assignments. It is the responsibility of the team to ensure that work is **distributed equitably** among members. Any disagreements or conflicts should be addressed and resolved within the team in a **professional manner**. **If a resolution cannot be reached internally, the instructor may be requested to intervene**.

- ⚠️ **Important:** Your implementation **must be compiled and tested on the course designated Ocelot server** (`ocelot-bbhatkal.aul.fiu.edu`) **to ensure correctness and compatibility**. **Submissions that fail to compile or run as expected on Ocelot will receive a grade of ZERO.**
- Whether submitting as a **group or individually**, every submission must include a **README** file containing the following information:
    1. Full name(s) of all group member(s)
    2. **Prefix the name of the corresponding member with an asterisk** `*`
    3. Panther ID (PID) of each member
    4. FIU email address of each member
    5. Include any relevant notes about compilation, execution, or the programming environment needed to test your assignment. **Please keep it simple—avoid complex setup or testing requirements.**
- ⚠️ **Important - Late Submission Policy:** For each day the submission is delayed beyond the **Due date** and until the **Until date** (as listed on Canvas), a **10% penalty** will be applied to the total score. **Submissions after the Until date will not be accepted or considered for grading.**
    - **Due Date**: ANNOUNCED ON CANVAS
    - **Until Date**: ANNOUNCED ON CANVAS
- Please refer to the course syllabus for any other details.

---

8. 📦 Deliverable Folder (`ZIP file`) Structure - ⚠️ exact structure required - no additional files/subfolders

> ⚠️ **DO NOT submit any other files other than these required files. The batch autograder may treat any additional items in the ZIP file as invalid, which will result in a grade of zero: mission_control.c, communication.c, file_io.c, memory_mgmt.c, README.txt**
>
> 📁 All other required framework files and test data will be supplied by the instructor to the autograder during final grading.

- ⚠️ You are required to submit your work as **a single ZIP archive file**.
- ⚠️ **Filename Format**: your `A4_Firstname_Lastname.zip` (**exact format required**).
- **You will be held responsible for receiving a ZERO grade if the submission guidelines are not followed**.

```
A4_FirstName_LastName.zip
├── mission_control.c        # Functions 1 & 2 implementation
├── communication.c          # Function 3 implementation
├── file_io.c                # Functions 4 & 5 implementation
├── memory_mgmt.c            # Function 6 implementation
└── README.txt               # Team details (see submission guidelines)
```

---

9. 💯 Grading Rubric

Criteria	Marks
⚠️ **Program fails to run or crashes during testing**	**0**
⚠️ **Partial Implementation** *(Evaluated based on completeness and at the instructor's discretion)*	**0 - 90**

Detailed Breakdown of 100 Points

Autograding Component	Points
Compilation & Build	10
`create_mission_control` implementation	10
`create_mission_with_crew` implementation	25
`add_communication` implementation	20
`load_missions_from_file` implementation	15
`save_mission_report` implementation	10
`free_mission_control` implementation	10
AUTOGRADER TOTAL	**90**
Manual Grading: Clear code structure, Meaningful comments, well-written README, Overall readability, Following the submission guidelines	10
FINAL TOTAL	**100**

⚠️ **Note**: The autograder awards 90 points automatically based on correctness. The remaining 10 points are manually graded for code quality, comments, structure, and README completeness.

Good luck! 💪