COP 4338 Assignment 3: Restaurant Management 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 **advanced C programming concepts** and **systems programming fundamentals**. You will implement **6 core functions** for a **restaurant menu and order management system** that demonstrates **structured programming principles**, **multi-dimensional array manipulation**, **pointer arithmetic**, **function pointers**, **const correctness**, and **string manipulation** techniques essential for systems programming.

The system simulates real-world data management scenarios commonly found in embedded systems, database management, and enterprise applications. **The instructor-provided framework handles all input/output, formatting, and testing** - your job is to focus purely on implementing the core algorithmic and pointer concepts.

☑️**Course Modules Covered:**

- **Module-1**
- **Module-2**
- **Module-3**

---

3. 🎯 Learning Objectives

By completing this assignment, you will be able to:

1. **Master pointer arithmetic** for efficient array traversal and string manipulation without using indexing.
2. **Apply call-by-reference techniques** with pointer parameters to modify and return multiple values safely.
3. **Use function pointers** to design flexible, modular, and reusable algorithms.
4. **Implement safe string and memory operations** using standard library functions with proper boundary checks and null-pointer safety.
5. **Design robust input validation and error-handling mechanisms** for reliable program behavior.
6. **Structure modular functions and maintainable program architectures** using proper parameter passing and const-correctness principles.

---

4. ⚙️ Assignment Task

What You Must Implement ?

You will implement **6 functions** that collectively create a **restaurant menu and order management system**. The instructor-provided framework handles all formatting, testing, and integration.

Core System Capabilities:

- Manages up to **10 menu items** with names, categories, and prices
- Handles **order processing** with quantity-based calculations
- Stores item names (max 20 characters), categories (max 15 characters), and prices ($1.00 - 50.00$)
- Processes **multiple sorting strategies** using function pointers
- Calculates **order statistics** including subtotal, tax, and final total
- Provides **search capabilities** using pointer-based linear search
- Uses **safe string manipulation** with standard library functions

6 Required Functions:

1. `copyString()` - String copying using pure pointer arithmetic
2. `findMenuItem()` - Linear search implementation with string comparison
3. `calculateSum()` - Array summation using pointer arithmetic
4. `addMenuItem()` - Array modification with call-by-reference
5. `processOrder()` - Multiple return values via pointer parameters
6. `sortMenu()` - Sort menu using function pointer for comparison strategy

📋 Detailed Function Specifications

Function 1: `copyString(char* dest, const char* src)`

Purpose: Implement fundamental string copying using pure pointer arithmetic without any string library functions.

Requirements:

- Use **only pointer arithmetic** (no array indexing like `src[i]`)
- Copy characters one by one until null terminator (`'\0'`)
- Ensure destination is properly null-terminated
- Handle NULL pointer validation

Integration Role: Used throughout the system for data manipulation.

Function 2: `findMenuItem(char names[][MAX_NAME_LENGTH], int count, const char* searchName)`

Purpose: Implement linear search algorithm to locate menu items by name using safe string comparison.

Requirements:

- Return index (0 to count-1) if item found, `ITEM_NOT_FOUND` (-1) if not found
- Use `strncmp()` for bounded string comparison safety
- Implement student-defined loop safety protection
- Handle edge cases (empty arrays, NULL parameters)

Integration Role: Enables the system to locate existing menu items for operations like ordering and modification.

Function 3: `calculateSum(float* prices, int count)`

Purpose: Calculate total sum of price array using pointer arithmetic for array traversal.

Requirements:

- Use **only pointer arithmetic** for traversal (no `prices[i]` notation)
- Accumulate sum using pointer dereferencing (`*ptr`)
- Implement student-defined loop safety protection
- Return 0.0 for invalid input or empty arrays

Integration Role: Provides statistical analysis capabilities for menu pricing and order calculations.

Function 4: `addMenuItem(char names[][MAX_NAME_LENGTH], char categories[][MAX_CATEGORY_LENGTH], float* prices, int* count, const char* name, const char* category, float price)`

Purpose: Add new menu items to arrays using call-by-reference for count modification and safe string operations.

Requirements:

- Validate all input parameters (NULL checks, empty strings, price range)
- Check array capacity using `MAX_MENU_ITEMS` constant
- Use `strncpy()` with proper length limits and manual null termination
- Update count via pointer dereferencing (`(*count)++`)
- Return `OPERATION_SUCCESS` or `OPERATION_FAILURE`

Integration Role: Enables dynamic menu expansion, allowing the restaurant system to grow its menu offerings.

Function 5: `processOrder(float* prices, int* quantities, int menuSize, float* subtotal, float* tax, float* total)`

Purpose: Calculate order totals and return multiple calculated values through pointer parameters.

Requirements:

- Process parallel arrays: prices and quantities
- Calculate subtotal (sum of price × quantity for items with quantity > 0)
- Calculate tax using `TAX_RATE` constant
- Calculate final total (subtotal + tax)
- Return all values through pointer parameters (`*subtotal = value`)
- Implement student-defined loop safety protection

Integration Role: Handles the core business logic of order processing, computing costs for customer transactions.

Function 6: `sortMenu(char names[][MAX_NAME_LENGTH], char categories[][MAX_CATEGORY_LENGTH], float* prices, int count, CompareFunction compare)`

Purpose: Implement flexible sorting using function pointers for strategy pattern, with advanced function pointer detection.

Requirements:

- Create function pointer array: `{compareByPrice, compareByName, compareByCategory}`
- Detect which comparison function was passed using pointer comparison
- Dynamically select appropriate data (prices, names, or categories) for comparison
- Sort all three arrays simultaneously to maintain data consistency
- Use `strncpy()` for safe string swapping during sort operations

- Implement student-defined loop safety protection
- Support any sorting algorithm of your choice (bubble, selection, insertion, etc.)

Integration Role: Provides flexible menu organization capabilities, allowing different sorting criteria for different business needs.

---

5. 🏗️ Project Framework

This assignment includes a comprehensive development framework comprising **skeleton code files** (*templates for your implementation*), **utility functions**, a **driver program**, a **Makefile**, and an **autograders** to support and guide your implementation process.

1) Provided Framework Infrastructure (⚠️ *Do Not Modify !*)

- `restaurant.h`: Complete header file with constants, macros, and function prototypes.**(study it thoroughly!)** .
- `driver.c`: The main driver program responsible for coordinating all tests and managing the overall execution flow.
- `Makefile`: Used to build your application and generate the final executable `restaurant`. You can run it using the following command:

```
# 📖 To see all available make commands and usage information
make help

# 🏗️ To build your application
make

# 🧹 To clean the build environment (remove old binaries, object files, etc.)
make clean

# 🔄 To rebuild everything from scratch (clean + build)
make rebuild
```

- `autograder_restaurant_management_system.sh`: Automated testing script **(identical to the final grading mechanism)**.
- `batchgrader_restaurant_management_system.sh`: Batch processing script for multiple submissions. It looks for all the required files in the ZIP folder and flags error if cannot be found.

  ⚠️ **It is very importat to test your implelentation with both the autograders before the final submission**

2) Individual Function Template Files (FOR YOUR IMPLEMENTATION):

You must implement **6 functions** in separate C files:

💡 Please **remove the** `_skeleton` **suffix** from the provided filenames before starting your implementation, testing, and submission.

- `copyString_skeleton.c` - String copy function
- `findMenuItem_skeleton.c` - Search function
- `calculateSum_skeleton.c` - Sum calculation function
- `addMenuItem_skeleton.c` - Add menu item function

- `processOrder_skeleton.c` - Order processing function
- `sortMenu_skeleton.c` - Sorting function with function pointers

3) Testing Infrastructure

- `TESTCASES.txt`: # Test cases (**copy simple/moderate/rigorous testcases here**).
- `EXPECTED_OUTPUT.txt`: # Expected results (generated by executing **./A3_sample > EXPECTED_OUTPUT.txt**).
- **Test Case Sets:** Multiple test case sets ranging from simple to rigorous complexity. As you progress with your implementation, copy each of these test cases into `TESTCASES.txt` one by one. **You are required to pass all the test cases** !
    - `testcases_simple.txt` / `expected_output_simple.txt`: Basic validation tests.
    - `testcase_moderate.txt` / `expected_output_moderate.txt`: Intermediate complexity tests.
    - `testcase_rigorous.txt` / `expected_output_rigorous.txt`: Comprehensive stress tests.

4) Instructor's Reference Implementation

- `A3_sample`: The instructor-provided reference executable that demonstrates the **complete, correct, and required implementation** of this assignment. 💡 **This executable serves as your primary reference for understanding the expected behavior.**
- ⚠️ **Important**: Your program's output must **exactly match** that of the instructor-provided executable for equivalent input. Any deviation in behavior, logic, or output formatting may prevent the autograder from evaluating your submission.

5) Data Structure Requirements

- All data structures must use the definitions provided in `restaurant.h`. Refer to this header file **thoroughly** to understand the constants, macros, function prototypes, and other framework components required for your implementation.

6) Provided Files Layout

```
A3_PROVIDED_FILES/
├── Sample_Executable/
    └── A3_sample                                   # Reference implementation
├── Framework_Files/
    ├── restaurant.h                                # Header file (DO NOT MODIFY)
    ├── driver.c                                    # Main program (DO NOT MODIFY)
    ├── Makefile                                    # Build system (DO NOT MODIFY)
    ├── autograder_restaurant_management_system.sh  # Autograder
    ├── batchgrader_restaurant_management_system.sh # Batch autograder
    ├── TESTCASES.txt                               # Test input data
    ├── EXPECTED_OUTPUT.txt                         # Expected output
    ├── copyString_skeleton.c                       # Your implementation file
    ├── findMenuItem_skeleton.c                     # Your implementation file
    ├── calculateSum_skeleton.c                     # Your implementation file
    ├── addMenuItem_skeleton.c                      # Your implementation file
    ├── processOrder_skeleton.c                     # Your implementation file
    └── sortMenu_skeleton.c                         # Your implementation file
├── Testcases/
```

```
    ├── testcases_simple.txt                    # Simple testcases
    ├── testcase_moderate.txt                   # Moderate testcases
    └── testcase_rigorous.txt                   # Rigorous testcase
├── A3_Specification.pdf                        # This specification document
└── README.txt                                  # Team details template
```

7) ✅ Execution Permissions Notice

If you **do not** have execute permissions for instructor-provided framework files, you can manually assign the necessary permissions to specific files or to all files within a directory using the following command on a **Linux system**:

> **Note:** When files are uploaded or transferred to Ocelot server (`ocelot-bbhatkal.aul.fiu.edu`), execute permissions **may be stripped** due to security restrictions. In such cases, you must explicitly grant **read (r)**, **write (w)**, and **execute (x)** permissions.

Granting Permissions to All Items in the Current Directory

```
# Check current permissions
ls -l

# Grant read, write, and execute permissions (safe and preferred – applies only what's missing)
chmod +rwx *

# Assign full permissions (read, write, execute) to all users for all files in the current directory
# ⚠️ Not preferred – use only as a last resort!
chmod 777 *

# Verify that permissions were applied
ls -l
```

⚠️ **Caution: Use `chmod 777` with care!**
Granting full permissions to everyone can:

- Expose sensitive data
- Allow unintended modifications or deletions
- Introduce security risks, especially in shared or multi-user environments

8) ☑️ YOU MUST!

- **Test your implementation thoroughly** on the **Ocelot** server (`ocelot-bbhatkal.aul.fiu.edu`)
- **Ensure exact output matching** with the instructor sample executable `A2_sample`
- **Validate all calculations** and logic against the sample
- **Test edge cases** and error handling extensively
- **Verify compilation** with the specified compiler flags

---

6. 🔧 Development & Testing

Required Files for Development

All the following files must be in the same directory:

1. **Framework files**: `driver.c`, `Makefile`, `restaurant.h`
2. **Your implementation files**: `copyString.c`, `findMenuItem.c`, `calculateSum.c`, `addMenuItem.c`, `processOrder.c`, `sortMenu.c`
3. **Testing tools**: `autograder_restaurant_management_system.sh`, `batchgrader_restaurant_management_system.h` `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 A3_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 `A3_sample` is located.

```
# Generate expected output from instructor sample
./A3_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**: `./resturant > 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

---

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: `A3_FirstName_LastName.zip` (e.g., **A3_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: copyString.c, findMenuItem.c, calculateSum.c, addMenuItem.c, processOrder.c, sortMenu.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 `A3_Firstname_Lastname.zip` (**exact format required**).
- **You will be held responsible for receiving a ZERO grade if the submission guidelines are not followed**.

```
A3_FirstName_LastName.zip
├── copyString.c          # String copy implementation
├── findMenuItem.c        # Search implementation
├── calculateSum.c        # Sum calculation implementation
├── addMenuItem.c         # Add menu item implementation
├── processOrder.c        # Order processing implementation
├── sortMenu.c            # Sorting 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

Component	Points
Compilation & Build	10
`copyString` implementation	10
`findMenuItem` implementation	10
`calculateSum` implementation	10
`addMenuItem` implementation	15
`processOrder` implementation	15
`sortMenu` implementation	20
AUTOGRADER TOTAL	**90**
Code quality, comments, README	**10**
FINAL TOTAL	**100**

Good luck! 💪