Workshop 8

Worth: 4.5% of final grade

Breakdown

- Part-1 Coding: 10%
- Part-2 Coding: 40%
- Part-2 Reflection: 50%

Submission Policy

- Part-1 is due **1-day** after your scheduled LAB class by the **end of day 23:59** EST (UTC – 5)
- Part-2 is due **5-days** after your scheduled LAB class by the **end of day 23:59** EST (UTC – 5)
- Source (.c) and text (.txt) files that are provided with the workshop MUST be used or your work will not be accepted. Resubmission will be required attracting a **15% deduction**
- **Late submissions will NOT be accepted**
- **All work must be submitted by the matrix submitter – no exceptions**
- **Reflections will not be read or graded** until the coding parts are deemed acceptable and graded.
- **All files** you create or modify MUST contain the following declaration at the top of all documents:

```
*****************************************************************************
              <assessment name example: Workshop - 8 (Part-1)>
Full Name  :
Student ID#:
Email      :
Section    :

Authenticity Declaration:
I declare this submission is the result of my own work and has not been
shared with any other student or 3rd party content provider. This submitted
piece of work is entirely of my own creation.
*****************************************************************************
```

Notes

- Due dates are in effect **even during a holiday**
- You are responsible for **backing up your work regularly**
- It is expected and assumed that for each workshop, you will plan your coding solution by using the computational thinking approach to problem solving and that **you will code your solution based on your defined pseudo code algorithm**.

Late Submission/Incomplete Penalties

If any Part-1, Part-2, or Reflection portions are missing, the mark will be **ZERO**.

Introduction

In this workshop, you will code and execute a C language program that evaluates the cheapest cat food product based on a simple analysis. Data will be entered for three similar products and then displayed back to the user in a tabular table format. The data will be analyzed and displayed with additional calculated information in the form of a formatted table revealing the analysis. The program concludes with the suggested cheapest product.

Topic(s)

- Modularity: Functions and Pointers

Learning Outcomes

Upon successful completion of this workshop, you will have demonstrated the abilities to:

- Create a modularized program using functions and multiple files
- Code functions to perform a specific task
- Call a function with several arguments of different types
- Pass an address to a function in a function call
- Refer to an address stored in a function's parameter
- Describe to your instructor what you have learned in completing this workshop

Part-1 (10%)

Instructions

Download or clone workshop 8 (**WS08**) from https://github.com/Seneca-144100/IPC-Workshops
Note: If you use the download option, make sure you **EXTRACT** the files from the .zip archive file

1. Carefully review the "Part-1 Output Example" (next section) to see how this program is expected to work
2. Do NOT MODIFY the "**main1.c**" file.
   - This launches some **pre-testing routines** for two functions you need to develop: **getIntPositive** and **getDoublePositive** (described later)
   - Following the pre-testing, it will call a function "**start**" that is the entry-point to the logic of your program
3. The code you write will be placed in the provided two additional files: "**w8p1.h**" (header file) and "**w8p1.c**" (source file)

File **w8p1.h** (header file)

1. This file will contain applicable **macro's** (#define), **structures**, and **function prototypes**
2. Create a **macro** that represents the maximum number of products to analyze (used in sizing the array later-on) – define this to be 3
3. Create a **macro** that represents the number of grams (64) in a suggested serving
4. Create a structure "**CatFoodInfo**" with the following related members that can hold:
   - a **whole number** for storing a product sku number (unique identifier)
   - a **double floating-point** number for the product price
   - a **whole number** for storing calories per suggested serving
   - a **double floating-point** number for the product weight in pounds (lbs)
5. You will need to create **seven (7) functions**. In this file, you need to code the **prototypes**. Here are the names (case sensitive) of those functions including a short description to help you create the necessary return types and parameter information for each:

1. **getIntPositive**
    - Returns an integer number
    - Receives an address that points to a whole number data type (provide a meaningful parameter name)
    - Note: This function returns the user-entered positive integer value in two ways:
        - Return value
        - Via the pointer argument
2. **getDoublePositive**
    - Returns a double floating-point number
    - Receives an address that points to a double floating-point data type (provide a meaningful parameter name)
    - Note: This function returns the user-entered positive double floating-point value in two ways:
        - Return value
        - Via the pointer argument
3. **openingMessage**
    - Does not return a value
    - Receives an argument that is an **unmodifiable** integer value representing the number of products the user will need to enter for analyzing (provide a meaningful parameter name)
4. **getCatFoodInfo**
    - Returns a "CatFoodInfo" type
    - Receives an **unmodifiable** integer number representing the sequence number of the product to be entered by the user (provide a meaningful parameter name)
5. **displayCatFoodHeader** (*this function is provided for you*)
    - Does not return a value nor receives any arguments
6. **displayCatFoodData**
    - Does not return a value
    - Receives multiple **unmodifiable** arguments representing each member of the "CatFoodInfo" type (not the structure itself)
    - Match the data types according to the member types in the "CatFoodInfo" type
    - Note: **integer** types are passed by value, but **double floating-point** types are passed by address
7. **start**
    - Does not return a value nor receive any arguments
    - This function acts as the main entry-point to the logic of the application (similar to what you are used to coding in "main" only this function does not return a value)

File **w8p1.c** (source file)

1. In this file, you will code the function definitions (the implementation of each function)
2. Include the necessary system and user-defined libraries you need for the application (system libraries use angle brackets (<>), while user-defined use double quotes ("")
3. Code each function definition implementation based on the prototypes declared in the header file. Below describes each function in a little more detail:
    1. **getIntPositive**
        - This function should accept user input for an integer value
        - If the value entered is a negative or zero value, an error message should be displayed (see example output for the message to display)
        - Logic should be designed to continue prompting until a positive value is entered
        - This function must return the entered value in **two ways**:

  - One: by assigning the entered value to the pointer argument (only if it is NOT NULL)
  - Two: by "return"ing the value.
  - This provides flexibility to the "caller" of the function where the value can be captured in two possible ways (or both). Refer to the **main1.c** file to see how this function is "pre-tested"

2. **getDoublePositive**
   - This is the same as the above described getIntPositive function, only this is for a **double floating-point type**.
3. **openingMessage**
   - See the example output "Cat Food Cost Analysis…"
   - Be sure to produce the number values by using the appropriate macro's defined earlier
4. **getCatFoodInfo**
   - Display a message to indicate the sequence number (use argument variable)
     - See the example output "Cat Food Product #..."
   - Prompt the user to enter data for **each member** of a "**CatFoodInfo**" type
   - **Call** appropriate **helper functions** to get validated user input **whenever possible**
     - Try calling the helper functions in different ways to get the return value 😊
     - Suggestion-1: Try getting the function output value via the **argument**
     - Suggestion-2: Try getting the function output value via the **returned value** (hint: send "NULL" as the argument)
     - **Warning: Under no circumstances should you receive the values using both the argument and return methods combined as this is extremely inefficient and redundant**
   - Return the **CatFoodInfo** value
5. **displayCatFoodHeader** (*this function is provided for you*)
   - Use the following formatting to display the table header:

```
printf("SKU         $Price    Bag-lbs Cal/Serv\n");
printf("------- ---------- ---------- --------\n");
```

6. **displayCatFoodData**
   - Use the following formatting do display a data row based on the arguments being received:

```
printf("%07d %10.2lf %10.1lf %8d\n"...
```

7. **start**
   - This is the entry-point to the logic portion of your application (called from main)
   - Create an **array** variable of type "**CatFoodInfo**" and size it using the appropriate macro defined in the header file (be sure to initialize it to a safe empty state)
   - You may need to declare other local variables as needed
   - Call the necessary function that will display an **opening message**
   - Nested inside an **iteration construct** that will iterate the necessary number of times, call the necessary function to **get user input** for each product and assign the function returned value to the appropriate **element** of the "**CatFoodInfo**" array variable (declared earlier). Hint: Send as an argument to the function, the iterator variable to represent the sequence number.
   - After obtaining all the product data from the user, call the necessary function that will display the **table header**
   - Nested inside an **iteration construct**, iterate the necessary number of times for each product, call the necessary function that will **display each "CatFoodInfo" record** (send the necessary arguments – reminder, some are pass-by-**value**, and others are pass-by-**address**).

Part-1 Output Example (Note: Use the YELLOW highlighted user-input data for submission)

```
============================
Pre-testing Helper Functions
============================

------------------------
Function: getIntPositive
------------------------
For each of these tests, enter the following
three values (space delimited):  -1 0 24

TEST-1: -1 0 24
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>
TEST-2: -1 0 24
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>
TEST-3: -1 0 24
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>

---------------------------
Function: getDoublePositive
---------------------------
For each of these tests, enter the following
three values (space delimited):  -1 0 82.5

TEST-1: -1 0 82.5
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>
TEST-2: -1 0 82.5
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>
TEST-3: -1 0 82.5
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>

===========================
Starting Main Program Logic
===========================

Cat Food Cost Analysis
======================

Enter the details for 3 dry food bags of product data for analysis.
NOTE: A 'serving' is 64g

Cat Food Product #1
--------------------
SKU           : 0
ERROR: Enter a positive value: 12221
PRICE         : $0
ERROR: Enter a positive value: 26.99
WEIGHT (LBS)  : 0
ERROR: Enter a positive value: 2.5
CALORIES/SERV.: 0
ERROR: Enter a positive value: 325

Cat Food Product #2
```

Function
openingMessage

Function
getCatFoodInfo

```
--------------------
SKU           : 23332
PRICE         : $41.99
WEIGHT (LBS)  : 5.5
CALORIES/SERV.: 325

Cat Food Product #3
--------------------
SKU           : 34443
PRICE         : $71.99
WEIGHT (LBS)  : 13.0
CALORIES/SERV.: 325

SKU         $Price    Bag-lbs Cal/Serv
------- ---------- ---------- --------
0012221      26.99        2.5      325
0023332      41.99        5.5      325
0034443      71.99       13.0      325
```

Functions
displayCatFoodHeader
displayCatFoodData

Part-1 Submission

1. Upload (file transfer) your source files: "**main1.c**", "**w8p1.h**", and "**`w8p1.c`**" to your matrix account
2. Login to matrix in an SSH terminal and change directory to where you placed your workshop source code.
3. Manually compile and run your program to make sure everything works properly:
   ```
   gcc -Wall main1.c w8p1.c -o w8 <ENTER>
   ```
   If there are no errors/warnings generated, execute it: ***w8*** *<ENTER>*
4. Run the submission command below (replace **profname.proflastname** with **your professors** Seneca userid and replace **NAA** with your section):
   ```
   ~profName.proflastname/submit 144w8/NAA_p1 <ENTER>
   ```
5. Follow the on-screen submission instructions

Part-2 (40%)

Instructions

This part will expand on Part-1 by adding the analysis component and will display the results of the analysis.

1. Carefully copy/paste your code from w8p1.h and w8p1.c and INSERT it into the respective provided Part-2 source files **w8p2.h** and **w8p2.c** where the comments indicate.
2. Review the "Part-2 Output Example" (next section) to see how the program is expected to work
3. Do NOT MODIFY the "**main2.c**" file.
   - This launches some additional **pre-testing routines** for three more helper functions you need to develop: **convertLbsKg**, **convertLbsG**, and **calculateLbs** (described later)
   - It finishes with a function call to "**start**" that is the entry-point to the logic of your program

File **w8p2.h** (header file)

1. Add another **macro** that represents the conversion factor of the number of U.S. pounds (lbs) in a KG unit value (2.20462) – this will be used in conversions from lbs to metric units later on
2. Add another structure "**ReportData**" with the following related members that can hold:
   - a **whole number** for storing a product sku number (unique identifier)
   - a **double floating-point** number for the product price
   - a **whole number** for storing calories per suggested serving
   - a **double floating-point** number for the product weight in pounds (lbs)
   - a **double floating-point** number for the product weight in kilograms (kg)
   - a **whole number** for the product weight in grams (g)
   - a **double floating-point** number for the total servings
   - a **double floating-point** number for the cost per serving
   - a **double floating-point** number for the cost of calories per serving
3. You will need to create an additional **ten (10) functions**. In this file, add the necessary prototypes (numbering continues from Part-1). Here are the names (case sensitive) for these new functions along with a short description to help you create the necessary return types and parameter information for each:

   1. **convertLbsKg**
      - Returns a double floating-point number
      - Receives an address that points to an **unmodifiable** double floating-point data type representing the pounds (lbs) to be converted (provide a meaningful parameter name)
      - Receives an address that points to a double floating-point data type representing the conversion result (equivalent kilograms (kg)) – provide a meaningful parameter name
      - Note: This function returns the pounds (lbs) to kilograms (kg) conversion in two ways:
        - Return value
        - Via the pointer argument

   2. **convertLbsG**
      - Returns a whole number
      - Receives an address that points to an **unmodifiable** double floating-point data type representing the pounds (lbs) to be converted (provide a meaningful parameter name)
      - Receives an address that points to a whole number data type representing the conversion result (equivalent grams (g)) – provide a meaningful parameter name
      - Note: This function returns the pounds (lbs) to grams (g) conversion in two ways:
        - Return value
        - Via the pointer argument

   3. **convertLbs**
      - Does not return a value
      - Receives an address that points to an **unmodifiable** double floating-point data type representing the pounds (lbs) to be converted (provide a meaningful parameter name)
      - Receives an address that points to a double floating-point data type representing the conversion result (equivalent kilograms (kg)) – provide a meaningful parameter name
      - Receives an address that points to a whole number data type representing the conversion result (equivalent grams (g)) – provide a meaningful parameter name

4. **calculateServings**
   - Returns a double floating-point number
   - Receives an **unmodifiable** whole number representing the serving size in grams
   - Receives an **unmodifiable** whole number representing the total grams for a product
   - Receives an address that points to a double floating-point data type representing the calculated result (number of servings)
   - Note: This function returns the calculated result in two ways:
     - Return value
     - Via the pointer argument
5. **calculateCostPerServing**
   - Returns a double floating-point number
   - Receives an address to an **unmodifiable** double floating-point data type representing the product price
   - Receives an address to an **unmodifiable** double floating-point data type representing the total number of servings
   - Receives an address to a double floating-point data type representing the calculated result (cost per serving)
   - Note: This function returns the calculated result in two ways:
     - Return value
     - Via the pointer argument
6. **calculateCostPerCal**
   - Returns a double floating-point number
   - Receives an address to an **unmodifiable** double floating-point data type representing the product price
   - Receives an address to an **unmodifiable** double floating-point data type representing the total number of calories
   - Receives an address to a double floating-point data type representing the calculated result (cost per calorie)
   - Note: This function returns the calculated result in two ways:
     - Return value
     - Via the pointer argument
7. **calculateReportData**
   - Returns a "ReportData" type
   - Receives an **unmodifiable** "CatFoodInfo" data type representing and item of product data
8. **displayReportHeader** (*this function is provided for you*)
   - Does not return a value nor receive any arguments
9. **displayReportData**
   - Does not return a value
   - Receives an **unmodifiable** "ReportData" data type representing the data to be displayed for a single row in the report
   - Receives an **unmodifiable** whole number representing if the received record is the cheapest product option
10. **displayFinalAnalysis**
    - Does not return a value
    - Receives an **unmodifiable** "CatFoodInfo" data type representing the data of the cheapest product

File **w8p2.c** (source file)

1. In this file, you will continue to code the function definitions (implementation) for the new functions prototyped in the header file.

2. Include the necessary system and user-defined libraries you need for the application (system libraries use angle brackets (<>), while user-defined use double quotes (""))
3. Use the supplied **w6p2.c** file **comments** to help you locate **where to insert your code logic**
   Note: Move function #7 ("**start**") to the end of the file so it is easy to locate and see how the main logic is implemented
4. Code each function definition implementation based on the prototypes declared in the header file. Below describes each function in a little more detail (numbering continues from the function listing in Part-1 and matches the header file function listing described earlier):
   8. **convertLbsKg**
      - This function should convert the received argument representing the pounds value (lbs) to kilograms (kg) by coding the necessary mathematical statement(s)
      - Use the macro previously created to help in the conversion (see previous header file section)
      - This function must return the converted value in **two ways**:
        - One: by assigning the result to the 2nd pointer argument (only if it is NOT NULL)
        - Two: by "return"ing the result
        - Refer to the **main2.c** file to see how this function is "pre-tested"
   9. **convertLbsG**
      - This function should convert the received argument representing the pounds value (lbs) to grams (g) by coding the necessary mathematical statement(s)
        Hint: There may be a function you already coded to help with this
      - This function must return the converted value in **two ways**:
        - One: by assigning the result to the 2nd pointer argument (only if it is NOT NULL)
        - Two: by "return"ing the result
        - Refer to the **main2.c** file to see how this function is "pre-tested"
   10. **convertLbs**
       - This function should convert the received argument representing the pounds value (lbs) to both, kilograms (kg), and grams (g)
       - Use the previously coded functions to perform these needed conversions
         Note: return the converted values by assigning the results to the respective pointer argument variables
       - Refer to the **main2.c** file to see how this function is "pre-tested"
   11. **calculateServings**
       - Using the values supplied in the first two received arguments, code the necessary mathematical statement(s) to derive the total servings
       - This function must return the converted value in **two ways**:
         - One: by assigning the result to the 3rd pointer argument (only if it is NOT NULL)
         - Two: by "return"ing the result
   12. **calculateCostPerServing**
       - Using the values supplied in the first two received arguments, code the necessary mathematical statement(s) to derive the cost per serving
       - This function must return the converted value in **two ways**:
         - One: by assigning the result to the 3rd pointer argument (only if it is NOT NULL)
         - Two: by "return"ing the result
   13. **calculateCostPerCal**
       - Using the values supplied in the first two received arguments, code the necessary mathematical statement(s) to derive the cost per calorie

- This function must return the converted value in **two ways**:
    - One: by assigning the result to the 3rd pointer argument (only if it is NOT NULL)
    - Two: by "return"ing the result

14. calculateReportData

- Review the **function prototype description** in this document to relate the arguments received by this function
- This function must return a **ReportData** data type, therefore, you will need to create a local variable of this type, assign the required values to it, and return the variable accordingly.
- The **ReportData** variable you create, will contain the required data for a single record in the analysis report.
- All the data assigned will be **derived from** the 1st argument received by this function
- The **first 4 members** of the **ReportData** variable can be directly assigned using the 1st parameter received to this function (referencing the appropriate members)
- The remaining members of the **ReportData** variable are **calculated values**. You should apply the appropriate previously created functions to help you accomplish this.
- All the data required for the calculations is at your disposal either via the 1st argument, or the values already calculated and stored to the **ReportData** variable
- The last statement in your function should be the returning of the local variable of type **ReportData**

15. displayReportHeader

- This function definition is provided for you – however, **you will need to complete one missing part** that substitutes the suggested "serving" size (in grams) as noted in yellow highlighted "**???**" below. This value should be supplied using **one of the macro's you created** (see the header file section).

```
printf("Analysis Report (Note: Serving = %dg\n", ???);
printf("---------------\n");
printf("SKU         $Price    Bag-lbs     Bag-kg     Bag-g Cal/Serv Servings  $/Serv   $/Cal\n");
printf("------- ---------- ---------- ---------- --------- -------- -------- ------- -------\n");
```

16. displayReportData

- This function will display the values of a ReportData type as a formatted row in the report
- Use the following formatting and fill-in the missing parts as required:

```
printf("%07d %10.2lf %10.1lf %10.4lf %9d %8d %8.1lf %7.2lf %7.5lf",...
```

- Note: If the 2nd argument received by this function is a non-zero value, you must append to the displayed row, a series of three asterisks "*******" to indicate that this product has been identified as the cheapest.

17. displayFinalAnalysis

- This function should display the final analysis recommendation message
- Use the argument received by this function to access the required data details
- Also, include a closing message "Happy shopping!"

7. start

- Upgrade this function to include the data analysis component
- You need to create an **array** variable of type "**ReportData**" and size it using the appropriate macro defined in the header file – this will be the same size as what was used for the array of "**CatFoodInfo**" done in Part-1 (be sure to initialize it to a safe empty state)
- You will need to create other local variables to help in the determination of the **cheapest product** which is based on **cost per serving** (a calculated value)
- Locate the logic in this function that processes user input for the product data (this should be nested inside an iteration construct). Immediately following that line, add a function call to "**calculateReportData**" and assign the returned value to the new array you just created (of type **ReportData**).

Hints

- **Use the same iterator variable** for the index as was used in the assignment of the **CatFoodInfo** in the previous statement
- Send as the 1st argument to "**calculateReportData**" the "**CatFoodInfo**" data just entered by the user

- You must also at some point determine which **CatFoodInfo** product is the cheapest based on the **cost per serving**.
    - This can be coded in a few places within this function, so you can determine the necessary logic and variables required to accomplish this
    - Hint: When you determine which **CatFoodInfo** product is the cheapest, you will need to store the **array index** of this product so you can reference the element whenever you need to afterwards/later in the function
- After displaying the **CatFoodInfo** list of products (the formatted table done in Part-1), display the results of the "ReportData" array in a formatted table. Use the new functions you developed to accomplish this.
- Finally, end the function with a call to the function that displays the final analysis results (Hint: this is where the saved index from before comes in 😊)

Part-2 Output Example (Note: Use the YELLOW highlighted user-input data for submission)

```
============================
Pre-testing Helper Functions
============================

------------------------
Function: getIntPositive
------------------------
For each of these tests, enter the following
three values (space delimited):  -1 0 24

TEST-1: -1 0 24
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>
TEST-2: -1 0 24
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>
TEST-3: -1 0 24
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>

---------------------------
Function: getDoublePositive
---------------------------
For each of these tests, enter the following
three values (space delimited):  -1 0 82.5

TEST-1: -1 0 82.5
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>
TEST-2: -1 0 82.5
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>
TEST-3: -1 0 82.5
ERROR: Enter a positive value: ERROR: Enter a positive value: <PASSED>

---------------------------
Function: convertLbsKg
---------------------------
Test-1: <PASSED>
Test-2: <PASSED>
Test-3: <PASSED>
---------------------------
Function: convertLbsG
---------------------------
Test-1: <PASSED>
```

```
Test-2: <PASSED>
Test-3: <PASSED>
---------------------------
Function: convertLbs
---------------------------
Test-1: <PASSED>

===========================
Starting Main Program Logic
===========================

Cat Food Cost Analysis
======================

Enter the details for 3 dry food bags of product data for analysis.
NOTE: A 'serving' is 64g

Cat Food Product #1
--------------------
SKU           : 0
ERROR: Enter a positive value: 12221
PRICE         : $0
ERROR: Enter a positive value: 26.99
WEIGHT (LBS)  : 0
ERROR: Enter a positive value: 2.5
CALORIES/SERV.: 0
ERROR: Enter a positive value: 325

Cat Food Product #2
--------------------
SKU           : 34443
PRICE         : $71.99
WEIGHT (LBS)  : 13.0
CALORIES/SERV.: 325

Cat Food Product #3
--------------------
SKU           : 23332
PRICE         : $41.99
WEIGHT (LBS)  : 5.5
CALORIES/SERV.: 325


SKU         $Price    Bag-lbs Cal/Serv
------- ---------- ---------- --------
0012221      26.99        2.5      325
0034443      71.99       13.0      325
0023332      41.99        5.5      325

Analysis Report (Note: Serving = 64g)
---------------
SKU         $Price    Bag-lbs     Bag-kg     Bag-g Cal/Serv Servings  $/Serv   $/Cal
------- ---------- ---------- ---------- --------- -------- -------- ------- -------
0012221      26.99        2.5     1.1340      1133      325     17.7    1.52 0.00469
0034443      71.99       13.0     5.8967      5896      325     92.1    0.78 0.00240 ***
0023332      41.99        5.5     2.4948      2494      325     39.0    1.08 0.00332

Final Analysis
--------------
Based on the comparison data, the PURRR-fect economical option is:
SKU:0034443 Price: $71.99

Happy shopping!
```

Reflection (50%)

Instructions

Record your answer(s) to the reflection question(s) in the provided "**reflect.txt**" text file

1. Several helper functions were designed to return values in two different ways (via an argument and/or by return). Explain **at least one benefit** of this "feature" and how it increases usability/flexibility and **at least one negative** reason why having functions designed to work in this way may not be desirable.

2. According to the design principles for structured design, functions should be **highly cohesive** and have **low coupling**. Identify one function in the workshop that demonstrates less than perfect cohesion, and one function that demonstrates coupling that could be improved. For each identified function, briefly explain what you would do to improve them.

3. What advantages are there in passing structures to functions? Based on your readings this week (for next week's topics) what could be done in the passing of this data to functions to make it more efficient? Explain how the "displayCatfoodData" function could be improved.

Academic Integrity

It is a violation of academic policy to copy content from the course notes or any other published source (including websites, work from another student, or sharing your work with others).

Failure to adhere to this policy will result in the filing of a violation report to the Academic Integrity Committee.

Part-2 Submission

1. Upload your source files: "**main2.c**", "**w8p2.h**", and "**`w8p2.c`**" to your matrix account
2. Upload your reflection file "**`reflect.txt`**" to your matrix account (to the same directory)
3. Login to matrix in an SSH terminal and change directory to where you placed your workshop source code.
4. Manually compile and run your program to make sure everything works properly:
```
gcc -Wall main2.c w8p2.c -o w8 <ENTER>
```
   If there are no errors/warnings generated, execute it: ***w8*** *<ENTER>*
5. Run the submission command below (replace **profname.proflastname** with **your professors** Seneca userid and replace **NAA** with your section):
```
~profName.proflastname/submit 144w8/NAA_p2 <ENTER>
```
6. Follow the on-screen submission instructions