This fourth project will give you experience writing a program that involves branching (if). This project will use the same development process as in Programming Projects #2 & #3, and will be subject to the same rules and grading criteria, with some new criteria added. This specification is less detailed than for previous programming projects. YOU ARE EXPECTED TO KNOW HOW TO PERFORM BASIC TASKS WITHOUT HAVING TO BE TOLD EXPLICITLY.

You’ve just opened a fast food restaurant.

Each meal your restaurant serves can consist of an entree, a side dish and a drink.

Your fast food restaurant has a menu that features several entrees (burger, chicken, fish), several side dishes (fries, rice, onion rings), and several drinks (soda, coffee, ice tea).

Each entree, each side dish and each drink has two size options: small and large.

Each item price depends on the size choice: for each, the small size is cheaper than the large size.

A customer can order AT MOST one item from each category; for example, they can order chicken as their entree, rice as their side dish and coffee as their drink. But, they AREN’T REQUIRED to order one of each category; for example, they can order no entree or side dish at all, just a cup of coffee.

The prices are:

<table>
<thead>
<tr>
<th>Item</th>
<th>Small Price</th>
<th>Large Price</th>
<th>Item</th>
<th>Small Price</th>
<th>Large Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burger</td>
<td>$4.75</td>
<td>$6.50</td>
<td>Soda</td>
<td>$1.75</td>
<td>$2.25</td>
</tr>
<tr>
<td>Chicken</td>
<td>$5.00</td>
<td>$6.25</td>
<td>Coffee</td>
<td>$2.50</td>
<td>$3.00</td>
</tr>
<tr>
<td>Fish</td>
<td>$5.50</td>
<td>$6.50</td>
<td>Ice Tea</td>
<td>$1.50</td>
<td>$2.00</td>
</tr>
<tr>
<td>Fries</td>
<td>$1.50</td>
<td>$2.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rice</td>
<td>$1.25</td>
<td>$1.75</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Onion Rings</td>
<td>$2.00</td>
<td>$2.75</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Your community’s sales tax rate is 8.25%.

NOTE: You are welcome to substitute other items, as long as you do so consistently and continue to follow the rules. For example, you could change burger to vegetarian burger, chicken to bean burrito, etc, but the number and categories of items, and the prices, must remain the same.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!

To get full credit, you MUST use if blocks properly, including nested if blocks.
I. PROJECT DESCRIPTION

Write a program that simulates greeting a customer, taking their order, calculating the bill and producing the check. The computer will be the cashier, and the user will be the customer.

EXTRA PREPROCESSOR DIRECTIVE

Your program MUST start with the following TWO preprocessor directives, in this order:
#include <stdio.h>
#include <stdlib.h>

II. STRUCTURE OF THE PROGRAM

The program body MUST be broken into four subsections:
1. Greeting subsection: Greet the customer (user).
2. Input subsection: Prompt for and input their order, one item at a time, IDIOTPROOFING EACH VALUE AS SOON AS IT IS INPUT.
3. Calculation subsection: Determine the price of each item, the subtotal, the tax amount, and the grand total.
4. Output subsection: Output their bill in receipt form.

Please note that you are ABSOLUTELY FORBIDDEN to have:
- ANY executable statements in your declaration section;
- ANY declarations in your execution section (body);
- ANY inputs or calculations in your greeting subsection;
- ANY calculations, or outputs other than prompts and idiotproofing error messages, in your input subsection;
- ANY inputs or outputs in your calculation subsection;
- ANY inputs or calculations in your output subsection.

That is, the subsections MUST BE COMPLETELY SEPARATE, and MUST BE CLEARLY LABELED.

For this programming project, if blocks are not considered to be inputs, nor calculations, nor outputs; that is, in principle you may have an if block in ANY subsection of the program body. However, statements inside the clauses of an if block MUST follow the rules above.

III. IMPLEMENTATION ORDER

Because the program will be complicated, you are STRONGLY advised to IMPLEMENT ONE PART AT A TIME, thoroughly test and debug it, and then go on to the next part. Also, it would probably be best to implement the subsections, NOT in the order in which they appear in the program (as described above), but rather in the following order:

1. Greeting subsection.
2. Input subsection (developed one item at a time). NOTE: If you’re unclear on how to idiotproof, then you can skip the idiotproofing during initial implementation and then develop the idiotproofing code later.
3. Output subsection (developed one item at a time). Note that, at this stage, some or all of your outputs will be garbage, because you haven’t yet written the calculation subsection.
4. Calculation subsection (developed one item at a time).
IV. DETAILS OF THE PROGRAM STRUCTURE

A. Greet the Customer
Welcome the customer to the restaurant.

B. Input the Customer’s Order

1. **Ask (prompt) the customer for their entree item choice**, giving them a list of entree items to choose from. You can use integer-valued codes to represent the entree items, and you may choose any REASONABLE values for coding these items. So, when the program prompts them for their entree item choice, the OUTPUT might be something like:

   What entree item would you like?
   Please enter:
   0 for no entree
   1 for burger
   2 for chicken
   3 for fish

2. **Input** the customer’s entree item choice.

3. **IDIOTPROOF** the entree item choice, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)

4. If the customer chose to purchase an entree, then:

   (a) **Ask (prompt) the customer for their entree size choice**, giving them a list of sizes to choose from. Again, you can use integer-valued codes to represent the entree sizes, and you may choose any REASONABLE values for coding these sizes. So, when the program prompts them for their entree size, the OUTPUT might be something like:

   What entree size would you like?
   Please enter:
   1 for small
   2 for large

   (b) **Input** the customer’s entree size.

   (c) **IDIOTPROOF** the entree size, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)
5. **Ask (prompt) the customer for their side dish item choice**, giving them a list of side dish items to choose from. Again, you can use integer-valued codes to represent the side dish items, and you may choose any **REASONABLE** values for coding these side dish items. So, when the program prompts them for their side dish item, the **OUTPUT** might be something like:

   What side dish would you like? Please enter:
   
   0 for no side dish
   1 for fries
   2 for rice
   3 for onion rings

6. **Input** the customer’s side dish item choice.

7. **IDIOTPROOF** the side dish item choice, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)

8. If the customer chose to purchase an side dish, then:
   
   (a) **Ask (prompt) the customer for their side dish size choice**, giving them a list of sizes to choose from.
   
   (b) **Input** the customer’s side dish size.
   
   (c) **IDIOTPROOF** the side dish size, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)

9. **Ask (prompt) the customer for their drink item choice**. Again, you can use integer-valued codes to represent the drink item choices, and you may choose any **REASONABLE** values for coding these items.

10. **Input** the customer’s drink item choice.

11. **IDIOTPROOF** the drink item choice, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)

12. If the customer chose to purchase a drink, then:
   
   (a) **Ask (prompt) the customer for their drink size**.
   
   (b) **Input** the customer’s drink size.
   
   (c) **IDIOTPROOF** the drink size, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)

**NOTE:** YOU MUST FULLY IDIOTPROOF EVERY INPUT THAT NEEDS IDIOTPROOFING. YOU ARE RESPONSIBLE FOR DETERMINING ALL POSSIBLE FORMS OF IDIOCY. Idiotproofing error messages MUST be HELPFUL and sufficiently detailed that even an idiot could figure out SPECIFICALLY what they’ve done wrong.

**IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!**

**ADVICE:** Avoid using ambiguous names for variables and named constants. Specifically, in this project, **DON’T** use names such as **drink** or **coffee**. Instead, use names that **CLEARLY** state the **ROLE** of the variable or named constant, such as **drink_item_code** or **coffee_price**.

**NOTE:** If the customer chooses to buy nothing — no entree, no side dish, no drink — then thank them and **EXIT** the program with a return code of zero (using an appropriate named constant), rather than calculating and printing an empty bill.
C. Calculate the Bill

1. **Entree price:** This value isn’t calculated as such, but is obtained from the entree item choice and the entree size choice.

2. **Side dish price:** This value isn’t calculated as such, but is obtained from the side dish item choice and the side dish size choice.

3. **Drink price:** This value is not calculated as such, but is obtained from the drink item choice and the drink size choice.

4. **Subtotal:** Calculate the subtotal of the food and drink prices.

5. **Tax amount:** Calculate the amount of tax on the subtotal. Both food and drink are taxed at the same tax rate (8.25%).

6. **Grand Total:** The grand total bill is the sum of the subtotal and the tax amount.

D. Print the Bill

1. Present the customer with an itemized bill in receipt form, using the placeholder below. For example:

   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Uncle Henry’s Qwik-Yums -- Receipt
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   Large Chicken: $ 6.25
   Small Rice: $ 1.25
   Large Coffee: $ 3.00

   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Subtotal: $10.50
   Tax: $ 0.87

   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Grand Total: $11.37

   Thank you for visiting Uncle Henry’s Qwik-Yums!

   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   (Substitute the name of your restaurant at the top.)

2. For all of the outputs in the itemized list (from the entree through the total), use the printf placeholder %5.2f, like so:

   printf(" Large Chicken: $%5.2f\n", entree_price);

   The **conversion format** in the placeholder tells the compiler that the printf statement will output some literal text, followed by a floating point number that takes up at least five spaces, two of which are to the right of the decimal point.

3. Item names **MUST** line up on the left side of the bill, dollar signs **MUST** line up, and the printf placeholder will cause prices to be flush to the right of the line. For the conversion format to work, **all dollar amounts MUST be float.**

4. List the bill entries in the order shown, using the name of each menu choice (table, page 1).

5. In some cases, the exact tax amount will have more than two digits to the right of the decimal point. We will accept results within five cents of the exactly correct bill.
V. RUNS
In your script, run the program 8 times, using the following inputs, in the following order:

1. no entree, no side dish, no drink
2. small burger, large fries, small soda
3. large chicken, small rice, large coffee
4. small fish, large onion rings, small ice tea
5. large burger, large rice, large ice tea
6. small chicken, small onion rings, small coffee
7. large fish, small fries, large soda
8. small coffee

In addition, **RUN THE PROGRAM ONCE FOR EACH POSSIBLE CASE OF IDIOCY** that a user might exhibit; that is, you **MUST** have runs that **COMPLETELY TEST EACH AND EVERY IDIOTPROOF CHECK. YOU ARE RESPONSIBLE FOR DETERMINING ALL POSSIBLE FORMS OF IDIOCY.** In your script file, the idiotproof test runs **MUST** occur **AFTER** the runs listed above.

**ADVICE:** Calculate each (non-idiotproof) run’s result by hand, then compare your hand-calculated values to the output of the program, to determine whether the program is running correctly.

VI. GRADING CRITERIA
A. SUBJECTIVE GRADING OF COMMENTS IN THE PROGRAM BODY
In previous programming projects, one of the grading criteria for comments in the program body has been that **EVERY** statement in the program body had to be preceded by a clear, helpful ex-
planatory comment.

- For PP#4 and beyond, you may choose to write fewer comments than this (though still in the format described in the PP#2 specification), in which case **YOU AGREE TO ACCEPT WITHOUT ARGUMENT** the graders’ **SUBJECTIVE** opinion on whether the amount and nature of your comments is sufficient.
- Alternatively, you may choose to continue to comply with the old criterion, preceding **EVERY** statement in the program body with a clear, helpful explanatory comment, in which case you are guaranteed to get full credit for this aspect of documentation in the program body (assuming that your comments comply with the original grading criteria for comments in the PP#2 specification).
B. NEW GRADING CRITERIA

1. **Format** of if statements, else if statements and else statements:
For each if statement, the if keyword **MUST** be followed by a blank space and then the open parenthesis that begins the condition. After the close parenthesis that ends the condition, there **MUST** be a blank space, followed by the block open.
For each else if statement, the same.
For each else statement, there **MUST** be a single blank space between the else keyword and the block open. For example:

```java
if (entree_item_code == no_item_code) {
    entree_price = no_item_price;
} /* if (entree_item_code == no_item_code) */
else if (entree_item_code == burger_code) {
    if (entree_size_code == small_code) {
        entree_price = small_burger_price;
    } /* if (entree_size_code == small_code) */
    else if (entree_size_code == large_code) {
        entree_price = large_burger_price;
    } /* if (entree_size_code == large_code) */
} /* if (entree_item_code == burger_code) */
else if (entree_item_code == chicken_code) {
    if (entree_size_code == small_code) {
        entree_price = small_chicken_price;
    } /* if (entree_size_code == small_code) */
    else if (entree_size_code == large_code) {
        entree_price = large_chicken_price;
    } /* if (entree_size_code == large_code) */
} /* if (entree_item_code == chicken_code) */
else if (entree_item_code == fish_code) {
    if (entree_size_code == small_code) {
        entree_price = small_fish_price;
    } /* if (entree_size_code == small_code) */
    else if (entree_size_code == large_code) {
        entree_price = large_fish_price;
    } /* if (entree_size_code == large_code) */
} /* if (entree_item_code == fish_code) */
```

2. **Format** of **if** conditions and **else if** conditions:
   For each **if** statement and each **else if** statement, in the condition, any binary operators — including relational operators such as `==` and Boolean operators such as `&&` — **MUST** be surrounded by one or more blank spaces on each side. Unary operators such as `!` **MUSTN'T** be surrounded by blank spaces. For example:
   
   ```c
   if ((entree_item_code != no_item_code) &&
       (entree_item_code != burger_code) &&
       (entree_item_code != chicken_code) &&
       (entree_item_code != fish_code)) {
       printf("ERROR: unknown entree item code %d.\n",
               entree_item_code);
       exit(program_failure_code);
   } /* if ((entree_item_code != no_item_code) && ...) */
   ```

3. **Indenting of **if** blocks:**
   For a given **if** block, the **if** statement, all **else if** statements (if any), the **else** statement (if any) and the block closes associated with these statements **MUST** all line up, both with each other and with other statements as appropriate. For example:
   
   ```c
   scanf("%d", &entree_item_code);
   if ((entree_item_code != no_item_code) &&
       (entree_item_code != burger_code) &&
       (entree_item_code != chicken_code) &&
       (entree_item_code != fish_code)) {
       printf("ERROR: unknown entree item code %d.\n",
               entree_item_code);
       exit(program_failure_code);
   } /* if ((entree_item_code != no_item_code) && ...) */
   if (entree_item_code != no_item_code) {
   ```
4. **Indenting INSIDE if blocks:**
   For a given if block, all statements INSIDE any clause of the if block MUST be indented **FOUR SPACES** farther than the associated if statement. This applies especially to if blocks nested inside other if blocks. For example:

   ```c
   if (entree_item_code != no_item_code) {
      printf("What size entree do you want?\n");
      printf(" %d for small\n", small_code);
      printf(" %d for large\n", large_code);
      scanf("%d", &entree_size_code);
      if ((entree_size_code != small_code) &&
           (entree_size_code != large_code)) {
         printf("ERROR: unknown entree size code %d.\n",
                entree_size_code);
         exit(program_failure_code);
      } /* if ((entree_size_code != small_code) && ...) */
   } /* if (entree_item_code != no_item_code) */
   ```

5. **Commenting if blocks:** You MUST follow all of the rules that are specified in the packet titled “Commenting if Blocks,” which will be posted on the course website soon.
6. **Idiotproofing**

(a) **ALL** inputs **MUST** include idiotproofing checks. Here’s a typical idiotproofing check:

```c
if ((entree_item_code != no_item_code) && 
    (entree_item_code != burger_code) && 
    (entree_item_code != chicken_code) && 
    (entree_item_code != fish_code)) {
    printf("ERROR: unknown entree item code %d.\n", 
            entree_item_code);
    exit(program_failure_code);
}
```

Notice the `exit` statement, which causes the program to immediately terminate. **ALL** idiotproofing checks **MUST** include an `exit` statement. Notice that the `exit` statement is **INSIDE** the `if` block, and should be indented appropriately (see above). Also, your program **MUST** have a named constant `program_failure_code` whose value is `-1`.

(b) Each idiotproofing check **MUST occur IMMEDIATELY after the associated `scanf` statement.** **ALL IDIOTPROOFING MUST BE COMPLETED BEFORE ANY CALCULATIONS ARE PERFORMED;** that is, idiotproofing belongs in the input subsection.

(c) Idiotproof error messages **MUST CLEARLY AND UNAMBIGUOUSLY** state the nature of the error. Thus, no two error messages should be the same.

(d) You **DON’T** have to idiotproof the data type. For example, when inputting the entree item code, you **DON’T** have to check whether the user input something like 2.73. Idiotproofing the data type is far beyond the scope of material covered in this course. In fact, you’d probably have to be in at least the second programming course for CS majors before you’d have covered that kind of material.

**VII. WHAT TO SUBMIT**

Submit cover, summary essay, script and extra credit bonus form (if any) in the usual style, format and order, and upload your source file and script file to Canvas in the usual way.
VIII. EXTRA CREDIT

You can receive an extra credit bonus of as much as 5% of the total value of Programming Project #4 by doing the following:

1. Attend at least one CS1313 help session for at least 30 minutes, through Wed March 29.
2. During the help session that you attend, work on CS1313 assignments (ideally PP#4, but any CS1313 assignment is acceptable). **YOU CANNOT GET EXTRA CREDIT IF YOU DON’T WORK ON CS1313 ASSIGNMENTS DURING THE HELP SESSION.**
3. Before you leave the help session, fill out **BOTH** halves of the form on the last page of this project specification and have the help session leader (instructor or TA) sign **BOTH** halves. **THE FORM CANNOT BE SIGNED UNTIL IT IS COMPLETELY FILLED OUT.**
4. Attach the bottom half of the form to your PP#4 script printout, **AFTER** the script itself, and keep the top half for your own records.

**BONUS VALUE NOTICE:** Through Wed March 8, the extra credit bonus will be worth **5%** of the total value of PP#4; from Mon March 20 through Wed March 22, the extra credit bonus will be worth **only 2.5%** of the total value of PP#4; from Mon March 27 through Wed March 29, the extra credit bonus will be worth **only 1.25%** of the total value of PP#4. That is, **YOU’LL GET TWICE AS MUCH EXTRA CREDIT DURING THE FIRST WEEK AS THE SECOND WEEK, AND TWICE AS MUCH EXTRA CREDIT DURING THE SECOND WEEK AS THE THIRD WEEK.**

**NOTE:** This extra credit bonus **WON’T** be available on any other programming project unless explicitly stated so in the project’s specification.
CS1313 PROGRAMMING PROJECT #4 BONUS REQUEST FORM

Name _________________________________ Lab ___________
Help Session Date ________________
Help Session Time (Arrive) ________________ Help Session Time (Depart) ____________

Instructor Signature ________________________________

Keep this copy for your records.

CS1313 PROGRAMMING PROJECT #4 BONUS REQUEST FORM

Name _________________________________ Lab ___________
Help Session Date ________________
Help Session Time (Arrive) ________________ Help Session Time (Depart) ____________

Instructor Signature ________________________________

Submit this copy.
In your submission, attach this copy AFTER your script file printout.
If you put this in the wrong place in your submission, then you WON’T get the extra credit.