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.

1. PROJECT DESCRIPTION

Your community’s sales tax rate is 8.75%. A tip of 18% is automatically added to every bill.
II. WHAT TO DO FIRST
Add the new program into your makefile in the usual way, as well as the example program (see below).

III. WHAT TO DO SECOND
For the example program in “if Lesson 1,” slides #37-40:
Type in, compile and run that example program, using the input values on slides #41-43 of the same lecture slide packet.
Then, comment that example program, and compile and run it again, with the same inputs.
Then, create a script file for it, named
pp4_example.txt

IV. EXTRA PREPROCESSOR DIRECTIVE
Your C source file MUST start with the following TWO preprocessor directives, in this order:
#include <stdio.h>
#include <stdlib.h>

V. STRUCTURE OF THE PROGRAM
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, the tip 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 WITH COMMENTS.

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.
VI. 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), located between the input and output subsections.

VII. 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 whether the meal is lunch or dinner.** You can use integer-valued codes to represent the meals (lunch or dinner), and you may choose any REASONABLE values for coding these items. So, when the program prompts the customer for the meal, the OUTPUT might be something like:
   
   Which meal is this?
   Please enter:
   1 for lunch
   2 for dinner

2. **Input** the meal code.

3. **IDIOTPROOF** the meal code, 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. **Ask (prompt) the customer for their appetizer item choice.** Again, you can use integer-valued codes to represent the appetizer item choices, and you may choose any REASONABLE values for coding these items. So, when the program prompts the customer for their appetizer item choice, the OUTPUT might be something like:
   
   What appetizer item would you like?
   Please enter:
   0 for no appetizer
   1 for antipasto
   2 for tomato bruschetta
   3 for cheese crostini

5. **Input** the customer’s appetizer item choice.

6. **IDIOTPROOF** the appetizer 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.)
7. Ask (prompt) the customer for their entree item choice, giving them a list of entree items to choose from. The list of entree item choices will differ depending on whether the meal is lunch or dinner. DON'T LIST ENTREE ITEM CHOICES THAT ARE INAPPROPRIATE FOR THE MEAL. Again, 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 the customer 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 chicken parmigiana
   2 for beef rigatoni
   3 for vegetarian lasagna
   If the meal is dinner, the list of entree item choices will be longer.

8. Input the customer’s entree item choice.

9. 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.)

10. Ask (prompt) the customer for their dessert item choice, giving them a list of dessert item choices to choose from. Again, you can use integer-valued codes to represent the dessert item choices, and you may choose any REASONABLE values for coding these items. So, when the program prompts the customer for their dessert item choice, the OUTPUT might be something like:
    What dessert item would you like?
    Please enter:
    0 for no dessert
    1 for cannoli
    2 for tiramisu
    3 for gelato

11. Input the customer’s dessert item choice.

12. IDIOTPROOF the dessert 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.)

13. 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.

14. Input the customer’s drink item choice.

15. 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.)

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.

If you haven’t yet learned how to idiotproof (we’ll get to it in lecture while you’re working on PP#4), then work on the rest of your program, and come back to the idiotproofing once you’ve learned how to idiotproof.
IMPORTANT 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 appetizer, no entree, no dessert, 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. Appetizer price: This value isn’t calculated as such; it’s obtained from the appetizer item choice.
2. Entree price: This value isn’t calculated as such; it’s obtained from the entree item choice and which meal this is.
3. Dessert price: This value isn’t calculated as such; it’s obtained from the dessert item choice.
4. Drink price: This value isn’t calculated as such; it’s obtained from the drink item choice.
5. Subtotal: Calculate the subtotal amount of the food and drink.
6. Tax amount: Calculate the amount of tax on the food and drink. Both food and drink are taxed at the same tax rate (8.75%).
7. Tip amount: The restaurant automatically adds a tip of 18% of the total food and drink bill, excluding tax — that is, the tip is calculated based on the sum of the appetizer price, the entree price, the dessert price and the drink price, NOT on the sum of the appetizer price, the entree price, the dessert price, the drink price and the tax amount.
8. Grand Total: The grand total bill is the sum of the appetizer price, the entree price, the dessert price, the drink price, the tax amount and the tip amount.

D. Print the Bill

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

-----------------------------------------
Enrico’s Bistro Bill

Your order was:
Antipasto: $ 7.75
Chicken Parmigiana Lunch: $17.75
Cannoli: $ 8.75
Coffee: $ 4.75

-----------------------------------------
Food Total: $31.25
Tax: $ 2.73
Tip: $ 5.62

-----------------------------------------
Grand Total: $39.61

Thank you for visiting Enrico’s Bistro!

(Substitute the name of your restaurant at the top and bottom.)
2. For all of the outputs in the itemized list (from the appetizer through the total), use the `printf` placeholder `%5.2f`, like so:

```c
printf(" Beef Rigatoni Lunch: $%5.2fn", 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 and tip amount will have more than two digits to the right of the decimal point. Because of rounding, we will accept results within five cents of the exactly correct bill.

---

### VIII. RUNS

In your script, run the program 6 times, using the following inputs, in the following order:

1. dinner, no appetizer, no entree, no dessert, no drink
2. lunch, antipasto, chicken Parmigiana, cannoli, coffee
3. dinner, tomato bruschetta, beef rigatoni, tiramisu, soda
4. lunch, cheese crostini, vegetarian lasagna, gelato, tea
5. dinner, linguini and clams, tea
6. dinner, 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.
IX. 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 explanatory 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:

   ```c
   if (appetizer_item_code == no_item_code) {
       appetizer_price = no_item_price;
   } /* if (appetizer_item_code == no_item_code) */
   else if (appetizer_item_code == antipasto_item_code) {
       appetizer_price = antipasto_price;
   } /* if (appetizer_item_code == antipasto_item_code) */
   else if (appetizer_item_code == tomato_bruschetta_item_code) {
       appetizer_price = tomato_bruschetta_price;
   } /* if (appetizer_item_code == tomato_bruschetta_item_code) */
   else if (appetizer_item_code == cheese_crostini_item_code) {
       appetizer_price = cheese_crostini_price;
   } /* if (appetizer_item_code == cheese_crostini_item_code) */
   ```

2. **Block open**: No source code text on the same line as, and after, a block open.

3. **Block close**: Only comment text on same line after a block close (see “Commenting if Blocks”).
4. **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 ((appetizer_item_code != no_item_code) &&
    (appetizer_item_code != antipasto_item_code) &&
    (appetizer_item_code != tomato_bruschetta_item_code) &&
    (appetizer_item_code != cheese_crostini_item_code)) {
    printf("ERROR: unknown appetizer item code %d.\n",
            appetizer_item_code);
    exit(program_failure_code);
} /* if ((appetizer_item_code != no_item_code) && ...) */
```

5. **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", &appetizer_item_code);
if ((appetizer_item_code != no_item_code) &&
    (appetizer_item_code != antipasto_item_code) &&
    (appetizer_item_code != tomato_bruschetta_item_code) &&
    (appetizer_item_code != cheese_crostini_item_code)) {
    printf("ERROR: unknown appetizer item code %d.\n",
            appetizer_item_code);
    exit(program_failure_code);
} /* if ((appetizer_item_code != no_item_code) && ...) */
printf("What entree would you like? Enter: \n");
```

6. **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 (appetizer_item_code == antipasto_item_code) {
    printf(" Antipasto: $%5.2f\n",
            appetizer_price);
} /* if (appetizer_item_code == antipasto_item_code) */
else if (appetizer_item_code == tomato_bruschetta_item_code) {
    printf(" Tomato Bruschetta: $%5.2f\n",
            appetizer_price);
} /* if (appetizer_item_code == tomato_bruschetta_item_code) */
else if (appetizer_item_code == cheese_crostini_item_code) {
    printf(" Cheese Crostini: $%5.2f\n",
            appetizer_price);
} /* if (appetizer_item_code == cheese_crostini_item_code) */
7. **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.

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

   ```c
   if ((meal_code != lunch_code) &&
       (meal_code != dinner_code)) {
     printf("ERROR: unknown meal code %d.\n", meal_code);
     exit(program_failure_code);
   } /* if ((meal_code != lunch_code) && ...) */
   ```

   Notice the `exit` statement, which causes the program to immediately halt. **ALL** idiotproofing checks **MUST** include an `exit` statement. 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`**. **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) Some idiotproof checks will depend on whether the item to be idiotproofed was actually input. You **MUSTN’T** idiotproof variables that haven’t been input.

**X. WHAT TO SUBMIT**

Upload to Canvas in the usual way, in the usual style and format, the cover, summary essay, example script file, C source file and script file.
EXTRA CREDIT

HELP SESSION BONUS EXTRA CREDIT

You can receive an extra credit bonus of as much as 5% of the total value of PP#4 as follows:

1. Attend at least one regularly scheduled CS1313 help session for at least 30 minutes, through Wed Oct 26.
2. During the regularly scheduled 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.**

**BONUS VALUE NOTICE:** Through Wed Oct 19, the extra credit bonus will be worth 5% of the total value of PP#4; from Mon Oct 24 through Wed Oct 26, the extra credit bonus will be worth only 2.5% of the total value of PP#4. That is, **YOU’LL GET TWICE AS MUCH EXTRA CREDIT DURING THE FIRST TWO WEEKS AS DURING THE FINAL WEEK.**

**NOTE:** This extra credit bonus **WON’T** be available on any other programming project unless explicitly stated so in the project’s specification.
Example program: I typed in, compiled, ran, comments, recompiled, reran, and created a script for the example program, (as described in the PP#4 specification, page 2, section III).

#include directives: I used the correct #include directives, in the correct order (as described in the PP#4 specification, page 2, section IV).

Named constants: I declared several named constants, for prices, item codes and tax rate (as described in the PP#4 specification, page 1).

Named constant names and variable names: My named constant names and variable names are sufficiently specific that I can easily tell what they refer to which are prices, which are item codes and so on, for example coffee_price and drink_item_code (as described in the PP#4 specification, page 4, section VII.B, the ADVICE).

Program structure: I used the correct program structure (as described in the PP#4 specification, page 2, section V).

Implementation order: I implemented the subsections of my program’s execution section (body) in the correct order (as described in the PP#4 specification, page 3, section VI).

Greeting: I wrote an appropriate greeting (as described in the PP#4 specification, page 3, item VII.A).

Meal prompt: I wrote an appropriate prompt for the meal code (as described in the PP#4 specification, page 3, item IV.B.1).

Meal input: I wrote an appropriate input for the meal code (as described in the PP#4 specification, page 3, item IV.B.2).

Meal idiotproof: I wrote an appropriate idiotproof for the meal code (as described in the PP#4 specification, page 3, item IV.B.3, and page 10, grading criterion 8a).

Appetizer item prompt: I wrote an appropriate prompt for the appetizer item code (as described in the PP#4 specification, page 3, item IV.B.4).

Appetizer item input: I wrote an appropriate input for the appetizer item code (as described in the PP#4 specification, page 4, item IV.B.5).

Appetizer item idiotproof: I wrote an appropriate idiotproof for the appetizer item code (as described in the PP#4 specification, page 4, item IV.B.6, and page 10, grading criterion 8a).

Entree item prompt: I wrote an appropriate prompt for the entree item code (as described in the PP#4 specification, page 3, item IV.B.4).

Entree item input: I wrote an appropriate input for the entree item code (as described in the PP#4 specification, page 4, item IV.B.5).

Entree item idiotproof: I wrote an appropriate idiotproof for the entree item code (as described in the PP#4 specification, page 4, item IV.B.6, and page 10, grading criterion 8a).

Dessert item prompt: I wrote an appropriate prompt for the dessert item code (as described in the PP#4 specification, page 4, item IV.B.8).

Dessert item input: I wrote an appropriate input for the dessert item code (as described in the PP#4 specification, page 4, item IV.B.9).

Dessert item idiotproof: I wrote an appropriate idiotproof for the dessert item code (as described in the PP#4 specification, page 4, item IV.B.10, and page 9, grading criterion 8).

Drink item prompt: I wrote an appropriate prompt for the drink item code (as described in the PP#4 specification, page 4, item IV.B.10).

Drink item input: I wrote an appropriate input for the drink item code (as described in the PP#4 specification, page 4, item IV.B.11).
Drink item idiotproof: I wrote an appropriate idiotproof for the drink item code (as described in the PP#4 specification, page 4, item IV.B.12, and page 9, grading criterion 6).

Test for buying nothing: I wrote appropriate code to test whether the user ordered nothing, in which case they would be thanked, and the program would exit without printing a bill (as described in the PP#4 specification, the bottom of page 4, the NOTE just before section IV.C).

Determine appetizer price: I wrote appropriate code to determine the appetizer price (as described in the PP#4 specification, page 5, item IV.C.1).

Determine entree price: I wrote appropriate code to determine the entree price (as described in the PP#4 specification, page 5, item IV.C.1).

Determine dessert price: I wrote appropriate code to determine the dessert price (as described in the PP#4 specification, page 5, item IV.C.3).

Determine drink price: I wrote appropriate code to determine the drink price (as described in the PP#4 specification, page 5, item IV.C.3).

Calculate subtotal: I wrote appropriate code to calculate the subtotal (as described in the PP#4 specification, page 5, item IV.C.5).

Calculate tax amount: I wrote appropriate code to calculate the tax amount (as described in the PP#4 specification, page 5, item IV.C.5).

Calculate tip amount: I wrote appropriate code to calculate the tip amount, based on the subtotal EXCLUDING THE TAX AMOUNT (as described in the PP#4 specification, page 5, item IV.C.7).

Calculate grand total: I wrote appropriate code to calculate the grand total (as described in the PP#4 specification, page 5, item IV.C.6).

Output bill header: I wrote appropriate code to output the header of the bill (as described in the PP#4 specification, page 5, item IV.D.1).

Output appetizer name and its price: I wrote appropriate code to output the appetizer name and its price (as described in the PP#4 specification, page 5, item IV.D.1).

Output entree name and its price: I wrote appropriate code to output the entree name and its price (as described in the PP#4 specification, page 5, item IV.D.1).

Output dessert name and its price: I wrote appropriate code to output the dessert name and its price (as described in the PP#4 specification, page 5, item IV.D.1).

Output drink name and its price: I wrote appropriate code to output the drink name and its price (as described in the PP#4 specification, page 5, item IV.D.1).

Output subtotal: I wrote appropriate code to output the subtotal (as described in the PP#4 specification, page 5, item IV.D.1).

Output tax amount: I wrote appropriate code to output the tax amount (as described in the PP#4 specification, page 5, item IV.D.1).

Output tip amount: I wrote appropriate code to output the tip amount (as described in the PP#4 specification, page 5, item IV.D.1).

Output grand total: I wrote appropriate code to output the grand total (as described in the PP#4 specification, page 5, item IV.D.1).

Output bill footer: I wrote appropriate code to output the footer of the bill (as described in the PP#4 specification, page 5, item IV.D.1).
Format of dollar figures in output: In my outputs of various dollar amounts, I used the appropriate conversion format in my placeholder (as described in the PP#4 specification, page 5, item IV.D.2).

Dollar figures in output line up: In my outputs of various dollar amounts, my dollar figures line up properly (as described in the PP#4 specification, page 5, item IV.D.3).

Order of outputs: In my outputs of various dollar amounts, I output them in the appropriate order (as described in the PP#4 specification, page 5, item IV.D.4).

Accuracy of outputs: In my outputs of various dollar amounts, my outputs are correct to within 5 cents (as described in the PP#4 specification, page 5, item IV.D.5).

Runs: In my script file, I did the correct runs in the correct order (as described in the PP#4 specification, page 6, section VIII).

Idiotproof runs: In my script file, I did all appropriate idiotproof runs, which all come after my regular runs (as described in the PP#4 specification, page 6, section VIII).

Calculation by hand: To verify that my program is correct, I calculated by hand the correct bill for each test run, and checked that result against my runs (as described in the PP#4 specification, page 6, section VIII).

Format of if statements: In my program, my if statements have the correct format (as described in the PP#4 specification, page 7, grading criterion 1).

No source code text on same line after a block open: In my program, my if statements have no source code text on the same line after the block open (as described in the PP#4 specification, page 8, grading criterion 2).

Only comment text on same line after a block close: In my program, my if block closes have no source code text on the same line after the block close except the comment that labels the block close (as described in the PP#4 specification, page 8, grading criterion 3).

Format of if conditions and else if conditions: In my program, my if conditions and my else if conditions have the correct format (as described in the PP#4 specification, page 8, grading criterion 4).

Indenting OF if blocks: In my program, my if blocks are properly indented (as described in the PP#4 specification, page 8, grading criterion 5).

Indenting INSIDE if blocks: In my program, statements inside my if blocks are properly indented (as described in the PP#4 specification, page 9, grading criterion 6).

Commenting if blocks: In my program, the block closes of my if blocks are properly labeled with comments on the same line (as described in “Commenting if Blocks”).

Idiotproofing of all inputs: In my program, every input has an idiotproof (as described in the PP#4 specification, page 10, grading criterion 8a).

Idiotproofing exit statement: In my program, every idiotproof has an exit statement (as described in the PP#4 specification, page 10, grading criterion 8a).

Idiotproofing exit statement inside if block: In my program, every idiotproof’s exit statement is inside the idiotproof if block (as described in the PP#4 specification, page 10, grading criterion 8a).

Idiotproofing exit statement indented properly: In my program, every idiotproof’s exit statement is indented properly (as described in the PP#4 specification, page 10, grading criterion 8a).
Idiotproofing exit statement uses program_failure_code: In my program, every idiotproof’s exit statement takes the argument program_failure_code, which is initialized to the correct value (as described in the PP#4 specification, page 10, grading criterion 8a).

Idiotproofing immediately after input: In my program, every idiotproof occurs immediately after the associated input (as described in the PP#4 specification, page 10, grading criterion 8b).

Idiotproofing error messages: In my program, every idiotproof error message is unique and unambiguous (as described in the PP#4 specification, page 10, grading criterion 8c).

Idiotproofing only variables that have been input: In my program, I only idiotproof variables that have been input (as described in the PP#4 specification, page 10, grading criterion 8d).

Uploads: I’ve uploaded the correct files to the Canvas PP#4 dropbox (as described in the PP#4 specification, page 10, section X).