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**

There are approximately ten different major categories of clouds, ranging from pretty puffy cumulus clouds to high flying cirro-stratus clouds. In this project, you will be creating a classifier that can automatically determine the type of cloud that you are viewing, based on just three characteristics of the cloud and the surrounding atmosphere. Figure 1 shows the ten possible types of clouds, nine of which you will be distinguishing in this project. Note that we won’t be covering fog, shown at the bottom of the figure, nor nimbus clouds.

Your task is to write a program that queries the user for the relevant characteristics and outputs the cloud type. The program should only ask about the characteristics needed to classify the cloud. For example, if the temperature is between -20 degrees C and -10 degrees C and the stability factor is nonnegative, then you are **ABSOLUTELY FORBIDDEN** to ask about the depth of the cloud.

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

To get full credit on PP#4, you **MUST** use if blocks properly.

For PP#4, you are **ABSOLUTELY FORBIDDEN** to use loops of any kind.

**Figure 1: Cloud Types**
You will be creating an automated system for cloud classification, based on the system published in 1998 by Zhang, Carr and Brewster\(^1\) of OU’s Center for Analysis & Prediction of Storms. This classification system relies on three characteristics:

1. The temperature of the air surrounding the cloud, measured in degrees Celsius. Note that it’s pretty cold where clouds form!

2. The *depth* of the cloud, measured in meters. This is how tall the cloud is (that is, the difference between the altitude of the top of the cloud and the altitude of the bottom of the cloud).

3. The stability of the atmosphere, measured in dz. Unstable air produces puffy cumulus clouds, whereas stable air produces flat stratus clouds. This is measured by examining the change in the equivalent potential temperature over altitude.

---

More specifically:

- if the temperature is less than -20 degrees C, then:
  - if the depth is greater than 5000 m **AND** the stability factor is negative, then:
    the cloud is cumulo-nimbus;
  - otherwise, if the stability factor is less than -0.0005 dz, then:
    the cloud is cirro-cumulus;
  - otherwise, if the stability factor is greater than or equal to -0.0005 dz **AND** the stability factor is less than 0.0005 dz, then:
    the cloud is cirrus;
  - otherwise, if the stability factor is greater than or equal to 0.0005 dz, then:
    the cloud is cirro-stratus;

- if the temperature is greater than or equal to -20 C **AND** the temperature is less than -10 C, then:
  - if the stability factor is negative, then:
    - if the depth is less than or equal to 5000 m, then:
      the cloud is alto-cumulus;
    - otherwise, the cloud is cumulo-nimbus;
  - otherwise, the cloud is alto-stratus;

- if the temperature is greater than or equal to -10 C, then:
  - if the stability factor is less than -0.0050 dz, then:
    - if the depth is less than or equal to 5000 m, then:
      the cloud is cumulus;
    - otherwise, the cloud is cumulo-nimbus;
  - otherwise, if the stability factor is greater than or equal to -0.0050 dz **AND** the stability factor is less than -0.0010 dz, then:
    the cloud is cumulus;
  - otherwise, if the stability factor is greater than or equal to -0.0010 dz **AND** the stability factor is less than 0.0010 dz, then:
    the cloud is strato-cumulus;
  - otherwise, if the stability factor is greater than or equal to 0.0010 dz, then:
    the cloud is stratus.
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 #35-38:
Type in, compile and run that example program, using the input values on slides #39-41 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
The program body MUST be broken into four subsections:
1. Greeting subsection: Greet the user.
2. Input subsection: Input the appropriate characteristics, IDIOTPROOFING EACH VALUE AS SOON AS IT IS INPUT if needed.
3. Calculation subsection: Determine the cloud type.
4. Output subsection: Output the cloud type.

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 User

Tell the user what the program will do.

B. Input the Characteristics

1. **Prompt for and then input the temperature in degrees C.**
2. **Idiotproof the temperature**, to ensure that the value that the user has input is a valid temperature in degrees C. (What would be an invalid temperature in degrees C?)
3. **Prompt for and then input the stability factor in dz.**
4. **IF THE CLOUD DEPTH IS NEEDED:**
   (a) **Prompt for and then input the cloud depth in m.**
   (b) **Idiotproof the cloud depth**, to ensure that the value that the user has input is valid.

**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 temperature or cloud. Instead, use names that CLEARLY state the ROLE of the variable or named constant, such as minimum_cloud_depth_in_m or cloud_type_code.
C. Determine the Cloud Type

This value is not calculated as such, but can be immediately obtained from the temperature, the stability factor and the depth, in a rather large `if` block.

The best way to accomplish this task is to represent the cloud type as an integer-valued code. For example, the code for cirrus might be 1, the code for cumulus might be 2, and so on.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
You will need to declare and initialize named constants for both the various thresholds on the input variables and also the various cloud types.

D. Output the Cloud Type

Output the cloud type in a complete English sentence, using a large `if` block.

VIII. RUNS

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

1. temperature -24.5 C, stability factor -0.0006, depth 5000.5 m
2. temperature -23.5 C, stability factor -0.0006, depth 5000.0 m
3. temperature -22.5 C, stability factor -0.0005 dz, depth 5000.0 m
4. temperature -21.5 C, stability factor +0.0005 dz
5. temperature -20.0 C, stability factor -0.0001, depth 5000.0 m
6. temperature -19.5 C, stability factor -0.0001, depth 5000.5 m
7. temperature -18.5 C, stability factor 0.0000
8. temperature -10.0 C, stability factor -0.0051, depth 5000.0 m
9. temperature -10.0 C, stability factor -0.0051, depth 5000.5 m
10. temperature -9.5 C, stability factor -0.0050
11. temperature -8.5 C, stability factor -0.0010
12. temperature -7.5 C, stability factor +0.0010

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.
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:

   ```
   if (cloud_type_code == cirrus_code) {
       printf("The cloud is a cirrus cloud.\n");
   } /* if (cloud_type_code == cirrus_code) */
   else if (cloud_type_code == cumulus_code) {
       printf("The cloud is a cumulus cloud.\n");
   } /* if (cloud_type_code == cumulus_code) */
   else if (cloud_type_code == stratus_code) {
       printf("The cloud is a stratus cloud.\n");
   } /* if (cloud_type_code == stratus_code) */
   else if (cloud_type_code == alto_stratus_code) {
       printf("The cloud is a alto-stratus cloud.\n");
   } /* if (cloud_type_code == alto_stratus_code) */
   else if (cloud_type_code == alto_cumulus_code) {
       printf("The cloud is a alto-cumulus cloud.\n");
   } /* if (cloud_type_code == alto_cumulus_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 (temperature_in_degrees_C < absolute_zero_in_degrees_C) {
     printf("ERROR: can’t have a temperature less than");
     printf(" %f degrees C.\n", absolute_zero_in_degrees_C);
     exit(program_failure_code);
   } /* if (temperature_in_degrees_C < absolute_zero_in_degrees_C) */
   ```

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("%f", &temperature_in_degrees_C);
   if (temperature_in_degrees_C < absolute_zero_in_degrees_C) {
     printf("ERROR: can’t have a temperature less than");
     printf(" %f degrees C.\n", absolute_zero_in_degrees_C);
     exit(program_failure_code);
   } /* if (temperature_in_degrees_C < absolute_zero_in_degrees_C) */
   printf("What is the stability factor in dz?\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 (temperature_in_degrees_C < temperature_in_C_threshold_low) {
     if (stability_factor_in_dz <
         stability_factor_in_dz_threshold_medium) {
       printf("What is the cloud depth in m?\n");
       scanf("%f", &cloud_depth_in_m);
       if (cloud_depth_in_m < minimum_cloud_depth_in_m) {
         printf("ERROR: can’t have a negative cloud depth.\n",
             absolute_zero_in_degrees_C);
         exit(program_failure_code);
       } /* if (cloud_depth_in_m < minimum_cloud_depth_in_m) */
     } /* if (stability_factor_in_dz < ...) */
   } /* if (temperature_in_degrees_C < ...) */
   ```

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 (temperature_in_degrees_C < absolute_zero_in_degrees_C) {
    printf("ERROR: can’t have a temperature less than");
    printf(" %f degrees C.\n", absolute_zero_in_degrees_C);
    exit(program_failure_code);
} /* if (temperature_in_degrees_C < ...) */
```

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) 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**

Submit cover, summary essay, example script, script, checklist, 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.
XI. EXTRA CREDIT

You can receive an extra credit bonus of as much as 2.5% 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 25.
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 paper submission, **AFTER** the checklist, and keep the top half for your own records.

**BONUS VALUE NOTICE:** Through Wed March 11, the extra credit bonus will be worth 2.5% of the total value of PP#4; from Mon March 23 through Wed March 25, 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 BEFORE SPRING VACATION AS AFTER.**

**NOTE:** This extra credit bonus **WON’T** be available on any other programming project unless explicitly stated so in the project’s specification.
**PP#4 CHECKLIST** (Print this checklist, check all that apply, and include this in your paper submission.  

<table>
<thead>
<tr>
<th>NAME</th>
<th>~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~</th>
<th>LAB SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐ 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 4, section III).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ #include directives: I used the correct #include directives, in the correct order (as described in the PP#4 specification, page 4, section IV).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Named constants: I declared several named constants (as described in the PP#4 specification, page 3).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ 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 (as described in the PP#4 specification, page 5, section VI.B, the ADVICE).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Program structure: I used the correct program structure (as described in the PP#4 specification, page 4, section V).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ 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 5, section VI).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Greeting: I wrote an appropriate greeting (as described in the PP#4 specification, page 5, item VII.A).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Temperature prompt: I wrote an appropriate prompt for the temperature in degrees C (as described in the PP#4 specification, page 5, item VII.B.1).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Temperature input: I wrote an appropriate input for the temperature in degrees C (as described in the PP#4 specification, page 5, item VII.B.1, and on page 8, grading criterion 5, the example code).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Temperature idiotproof: I wrote an appropriate idiotproof for the temperature in degrees C (as described in the PP#4 specification, page 3, in item VII.B.2, and page 9, grading criterion 6a, and on page 8, grading criterion 4, the example code).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Stability factor prompt: I wrote an appropriate prompt for the stability factor in dz (as described in the PP#4 specification, page 5, item VII.B.3, and on page 8, grading criterion 5, the example code).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Stability factor input: I wrote an appropriate input for the stability factor in dz (as described in the PP#4 specification, page 5, item VII.B.3).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Cloud depth prompt: I wrote an appropriate prompt for the cloud depth in m, <strong>BUT ONLY IF THE CLOUD DEPTH IS NEEDED</strong> (as described in the PP#4 specification, page 5, item VII.B.4.a, and on page 8, grading criterion 6, the example code).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Cloud depth input: I wrote an appropriate input for the cloud depth in m, <strong>BUT ONLY IF THE CLOUD DEPTH IS NEEDED</strong> (as described in the PP#4 specification, page 5, item VII.B.4.a, and on page 8, grading criterion 6, the example code).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Cloud depth idiotproof: I wrote an appropriate idiotproof for the cloud depth in m, <strong>BUT ONLY IF THE CLOUD DEPTH IS NEEDED</strong> (as described in the PP#4 specification, page 3, item VII.B.4.b, and on page 8, grading criterion 6, the example code).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Determine cloud type: I wrote appropriate code to determine the cloud type (as described in the PP#4 specification, page 6, item VII.C.1, the description on page 3).</td>
<td>☐</td>
<td></td>
</tr>
<tr>
<td>☐ Output cloud type: I wrote appropriate code to output the cloud type (as described in the PP#4 specification, page 6, item VII.D, and page 7, grading criterion 1, the example code).</td>
<td>☐</td>
<td></td>
</tr>
</tbody>
</table>
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, and my idiotproof runs all come after my regular runs (as described in the PP#4 specification, page 6, section VIII).

Format of \textit{if} statements: In my program, my \textit{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 \textit{if} statements have no source code text on the same line after the block open (as described in the PP#4 specification, page 7, grading criterion 2).

Only comment text on same line after a block close: In my program, my \textit{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 \textit{if} conditions and \textit{else if} conditions: In my program, my \textit{if} conditions and my \textit{else if} conditions have the correct format (as described in the PP#4 specification, page 7, grading criterion 4).

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

Indenting INSIDE \textit{if} blocks: In my program, statements inside my \textit{if} blocks are properly indented (as described in the PP#4 specification, page 8, grading criterion 6).

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

Idiotproofing of all inputs: In my program, every input has an idiotproof (as described in the PP#4 specification, page 9, grading criterion 8a).

Idiotproofing \textit{exit} statement: In my program, every idiotproof has an \textit{exit} statement (as described in the PP#4 specification, page 9, grading criterion 8a).

Idiotproofing \textit{exit} statement inside \textit{if} block: In my program, every idiotproof’s \textit{exit} statement is inside the idiotproof \textit{if} block (as described in the PP#4 specification, page 9, grading criterion 8a).

Idiotproofing \textit{exit} statement indented properly: In my program, every idiotproof’s \textit{exit} statement is indented properly (as described in the PP#4 specification, page 9, grading criterion 8a).

Idiotproofing \textit{exit} statement uses \texttt{program\_failure\_code}: In my program, every idiotproof’s \textit{exit} statement takes the argument \texttt{program\_failure\_code}, which is initialized to the correct value (as described in the PP#4 specification, page 9, 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 9, 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 9, 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 9, grading criterion 8d).
Paper submission: My paper submission has the correct printouts in the correct order (as described in the PP#4 specification, page 9, section X).

Uploads: I’ve uploaded the correct files to the Canvas PP#4 dropbox (as described in the PP#4 specification, page 9, section X).
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.

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.