

## p1-recipes

---

# EECS 183 Project 1: Focaccia Bread

---

**Project Due Friday, September 16, 2022, 8:00 pm Eastern**

---

### [Direct autograder link](#)

---

In this project, you will write a program to help you purchase the correct amount of ingredients to make [focaccia bread](#) for a party.

By completing this project, you will learn to:

- Read a specification document and translate it to a C++ program
- Write a large program by completing one small part at a time
- Follow the 4 steps of the development cycle to complete small parts of a program
  - i. Make examples of input and output for that part
  - ii. Find the pattern linking the input and output
  - iii. Write code
  - iv. Test code
- Create test inputs to an entire program and compare the output you expect to the actual output of the program
- Use the autograder system in EECS 183

You will apply the following skills you learned in lecture:

- Lecture 2
  - Write a program with a `main()` function
- Lecture 3
  - Use appropriate data types for integers and floating-point numbers
  - Use functions to round floating point numbers up and down to the nearest integer
  - Use constants to store unchanging values
- Lecture 4
  - Call a function that takes multiple arguments

### Vocabulary

specification	development cycle
magic number	test input
diffchecker	sample run

# Getting Started

---

## Starter Files

---

You can download the starter files [using this link](#).

The IDE setup tutorials for Visual Studio and XCode include a video about how to set up a project using the starter files. If you're on a Windows computer, you should download Visual Studio. If you're on a Mac, you should download XCode. You can access the tutorials here:

- [Visual Studio - Windows computers](#)
- [XCode - Mac computers](#)

## Submission and Grading

---

Submit your code to the [autograder here](#). You receive 4 submits each day and your best overall submission counts as your score. There are 60 possible points for correctness and 10 points for style.

The deadline is Friday, September 16, 2022 at 8PM Eastern. If your last submission is before Wednesday, September 14 at 11:59PM Eastern, you will receive a 5% bonus. If your last submission is before Thursday, September 15 at 11:59PM Eastern, you will receive a 2.5% bonus.

You have 3 late days that you can use any time during the semester for projects. There are 3 late days total, not 3 per project. To use a late day, submit to the autograder after the deadline. It will prompt you about using one of your late day tokens. There are more details about late days [in the syllabus](#).

## How to get help

---

Most students in EECS 183 need help from staff and faculty multiple times each project. We're here for you!

If your question is about the specification or about something about the project in general, Piazza is the fastest place to get help.

You can meet with us for help in office hours. You can [find details on group and 1-1 office hours here](#).

## Collaboration Policy

---

We want students to learn from and with each other, and we encourage you to collaborate. We also want to encourage you to reach out and get help when you need it. You are encouraged to:

- Give or receive help in understanding course concepts covered in lecture or lab.
- Practice and study with other students to prepare for assessments or exams.
- Consult with other students to better understand project specifications.
- Discuss general design principles or ideas as they relate to projects.
- Help others understand compiler errors or how to debug parts of their code.

To clarify the last item, you are permitted to look at another student's code to help them understand what is going on with their code. You are not allowed to tell them what to write for their code, and you are not allowed to copy their work to use in your own solution. If you are at all unsure whether your collaboration is allowed, please contact the course staff via the admin form before you do anything. We will help you determine if what you're thinking of doing is in the spirit of collaboration for EECS 183.

The following are considered Honor Code violations:

- Submitting others' work as your own.
- Copying or deriving portions of your code from others' solutions.
- Collaborating to write your code so that your solutions are identifiably similar.
- Sharing your code with others to use as a resource when writing their code.
- Receiving help from others to write your code.
- Sharing test cases with others if they are turned in as part of your solution.
- Sharing your code in any way, including making it publicly available in any form (e.g. a public GitHub repository or personal website).

The full collaboration policy can be [found in the syllabus](#).

## What are Specifications?

---

The project descriptions in EECS 183 are *specifications*, which tell you what your program needs to do when you are finished. Specifications don't tell you how to write the program or the steps it

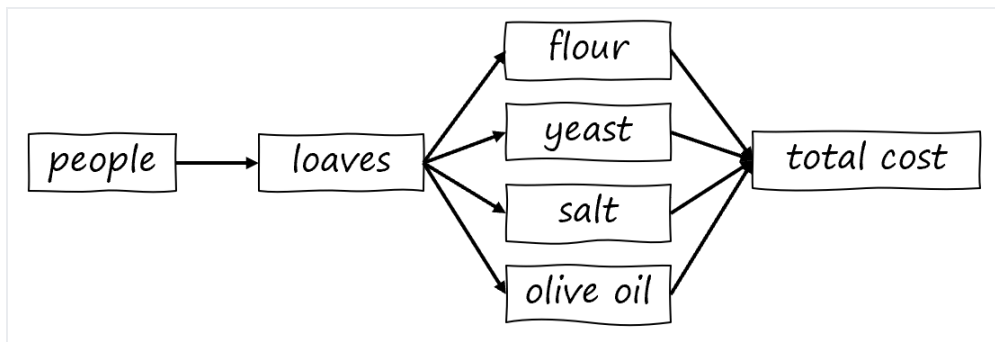
takes for you to write the program with the least amount of work. That means that you have a lot of freedom. At the beginning, this freedom means that it is hard to know where to get started.

Take a look at the specification [at the bottom of this page](#) before moving on.

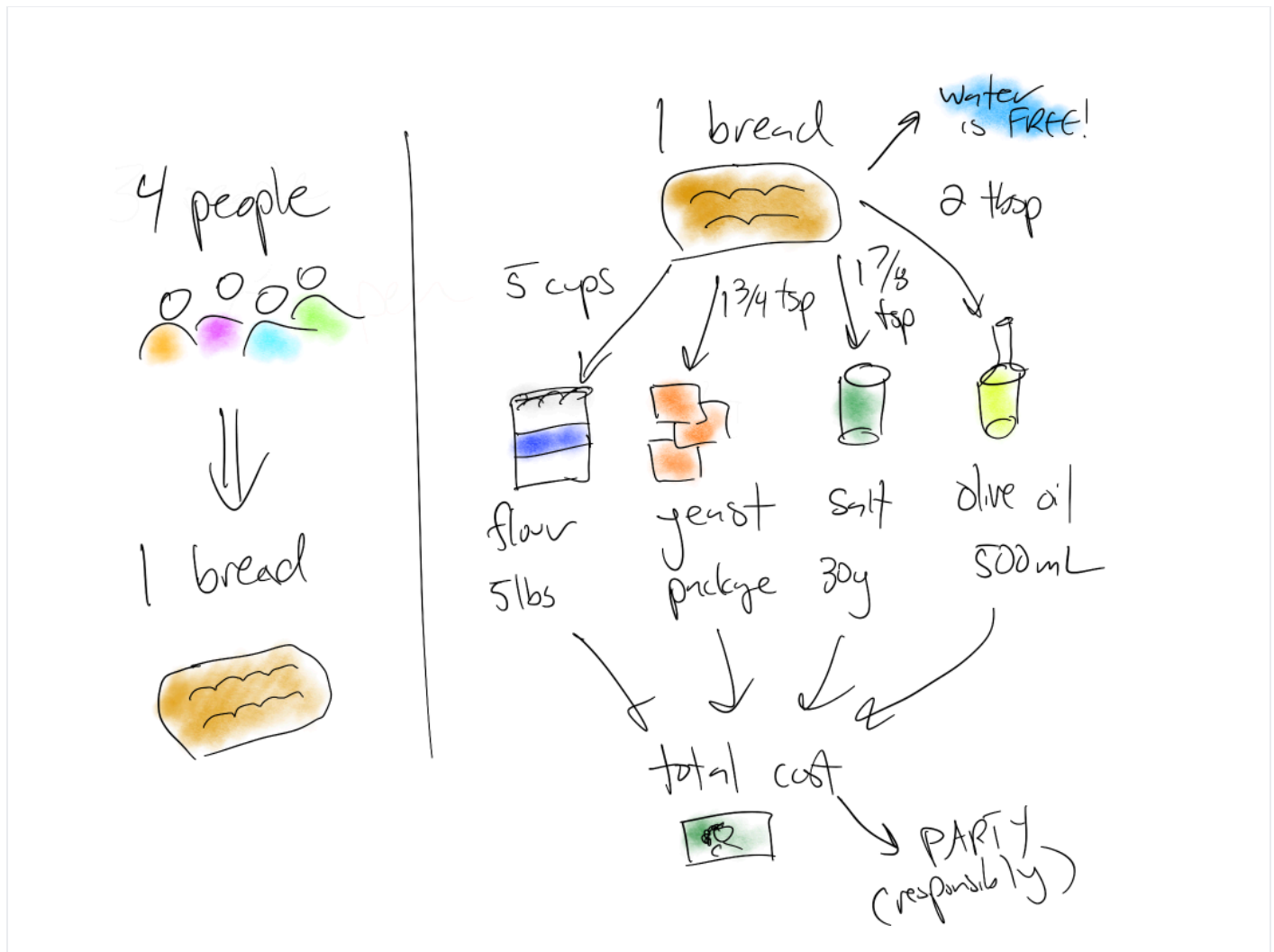
## Making Smaller Parts

You probably noticed that there are many interconnected parts to this program. For example, the total cost depends on the number of packages of each ingredient, which depends on the number of loaves. Future 183 projects will have more parts that depend on each other in more complicated ways. Because of this, the first step in making a large program is breaking it apart into smaller parts. We'll build up the program by programming one smaller part at a time.

One way to represent the way things are connected is with a picture. Here is one possible picture for this project:



Here is another:



## Why work on one part at a time?

In programming, even experts make mistakes the first time they write part of a program. (The professors can't write this project without making mistakes!) What makes expert programmers productive is the process they use as write their program which makes these mistakes easier to discover and fix.

As an example, let's say that we completed Project 1 all at once, then got this incorrect output on a sample run.

Expected Output	Actual Ou
<pre> 1  You need to make: 3334 loaves of focaccia 2 3  Shopping List for Focaccia Bread 4  ----- 5  834 bags of flour 6  2594 packages of yeast </pre>	<pre> 1  You need to make: 1024 2 3  Shopping List for Foca 4  ----- 5  321 bags of flour 6  1194 packages of yeast </pre>

```

7  1042 canisters of salt
8  198 bottles of olive oil
9
10 Total expected cost of ingredients: $5056.86
11
12 Have a great party!

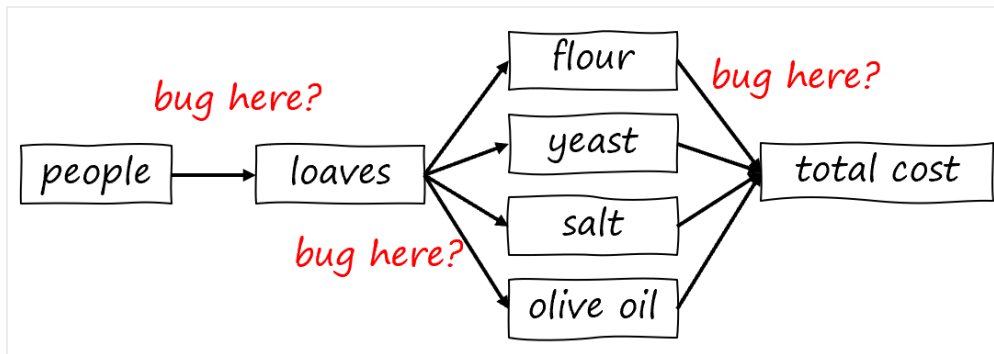
```

```

7  9342 canisters of salt
8  12 bottles of olive oil
9
10 Total expected cost of
11
12 Have a great party!

```

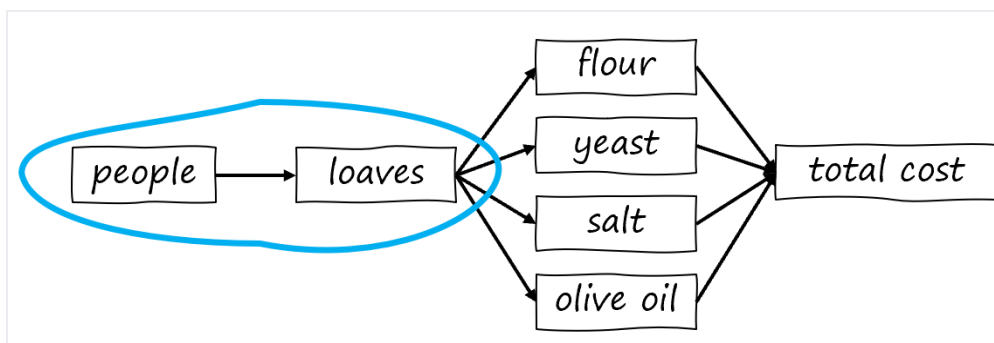
Notice how hard it is to find all the problems and also how it's not clear where to start to fix them all. A bug in one part of the program might be affecting a later part of the program. To put this in the picture from earlier, we don't know which of the steps is the problem:



## Development cycle

A better way to write the program would have been focusing on getting one part 100% correct, and only moving onto the next part when it is finished. Here's how we did this in our Project 1.

Because everything else depends on getting the number of loaves right, we decided to focus on that part first.



The *development cycle* describes 4 steps you can follow to build one small part to be 100% correct.

1. Make examples of input and the correct output
2. Find the pattern linking the input and output

3. Write C++ code
4. Test your code

You'll be following these steps yourself to complete the rest of the project, so follow along with these steps in your IDE.

## Step 1: Make examples of input and output

---

From the specification, we knew that every 4 people requires a loaf of focaccia. To make sure we understood, we wrote out a table of input (number of people) and output (number of loaves).

Input (people)	Output (loaves)
1	1
2	1
4	1
5	2
0	0

## Step 2: Find the pattern

---

Once we have the examples, we used them to find the relationship between them. In our experience, there's a difference between knowing how to do something in your head and how to write it down or explain it to someone else. In this step, we write the algorithm we think will work, then check it against the examples. Most of the time, even for experienced programmers, it takes multiple guesses.

**Guess 1:** Divide the number of people by 4 using integer division.

**Check:**

Input (people)	Output (loaves)
1	$1 / 4 = 0$ (not correct)

**Guess 2:** Divide the number of people by 4 using integer division, then add 1

**Check:**

Input (people)	Output (loaves)
1	$(1 / 4) + 1 = 1$
2	$(2 / 4) + 1 = 1$
4	$(4 / 4) + 1 = 2$ (not correct)

**Guess 3:** Divide the number of people by 4 using double division, then round up

**Check:**

Input (people)	Output (loaves)
1	$1 / 4.0 = 0.25$ rounded up gives 1
2	$2 / 4.0 = 0.5$ rounded up gives 1
4	$4 / 4.0 = 1.0$ rounded up gives 1
5	$5 / 4.0 = 1.25$ rounded up gives 2
0	$0 / 4.0 = 0.0$ rounded up gives 0

All our examples match, which makes us confident this algorithm will work.

## Step 3: Write C++ code

Now that we have the pattern, we need to translate the steps into C++. From Lecture 3, we remembered that to divide with double division, at least one of the operands needs to be of the type `double`. From there, we can round up the result using the `ceil()` function.

We wrote our code in the `main()` function of `focaccia.cpp`:

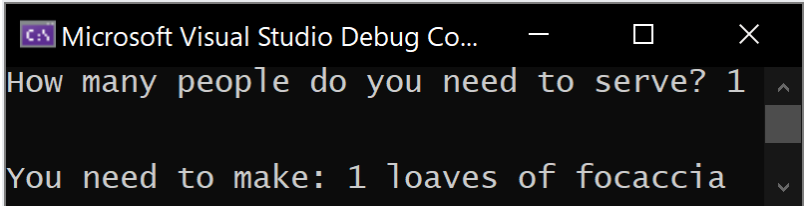
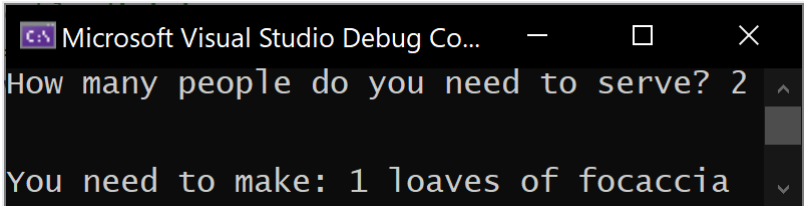
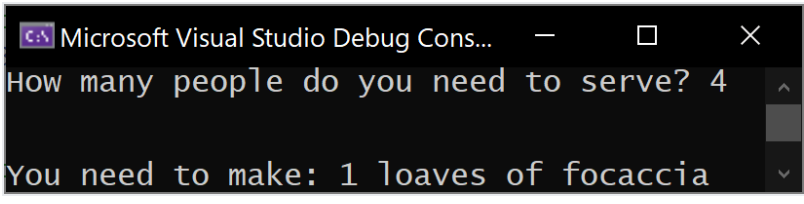
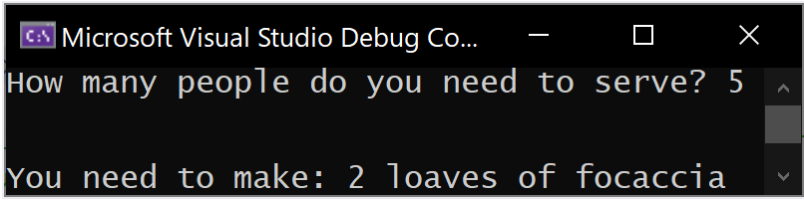
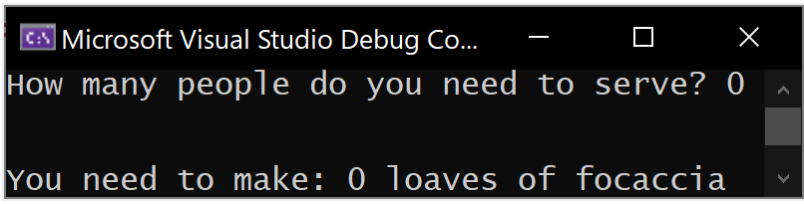
```

1  int main() {
2      int people = 0;
3      cout << "How many people do you need to serve? ";
4      cin >> people;
5      cout << endl;
6
7
8
9      return 0;
10 }
```



## Step 4: Test your code

Using the examples we thought of in Step 1, we can test whether our code works the way we expect by running the code in our IDE.

Input (people)	Output (loaves)
1	 <pre>Microsoft Visual Studio Debug Co... How many people do you need to serve? 1 You need to make: 1 loaves of focaccia</pre>
2	 <pre>Microsoft Visual Studio Debug Co... How many people do you need to serve? 2 You need to make: 1 loaves of focaccia</pre>
4	 <pre>Microsoft Visual Studio Debug Cons... How many people do you need to serve? 4 You need to make: 1 loaves of focaccia</pre>
5	 <pre>Microsoft Visual Studio Debug Co... How many people do you need to serve? 5 You need to make: 2 loaves of focaccia</pre>
0	 <pre>Microsoft Visual Studio Debug Co... How many people do you need to serve? 0 You need to make: 0 loaves of focaccia</pre>

Everything looks good, so we have evidence that we have both a correct algorithm and a correct C++ implementation of it.

## Your turn: bags of flour

We'll follow the same steps together for the next small piece of our program: computing the number of bags of flour.

## Step 1: Make examples of input and output

The output of this piece of the program will be the number of bags of flour, but we could choose input to be either the number of people or the number of loaves. Using the number of loaves is the better choice, because we already know how to translate the number of people to loaves. We've given you some useful test inputs – your job is to compute what the outputs will be.

This isn't a worksheet you need to hand in, so you can do this in your notes.

Input (loaves)	Output (bags of flour)
1	1
9	[your answer here]
30	[your answer here]
75	19

## Step 2: Find the pattern

**Guess 1:** Multiply the number of loaves by 5 then divide by 20

**Check:**

Input (loaves)	Output (bags of flour)
1	0.25 (not correct)

Do more guesses and checks until you find an algorithm that produces the output from Step 1.

## Step 3: Write C++ Code

Let's start by translating your algorithm into a single line of code by filling in the blank:

```
1  int flourBags = _____;
```

What you likely found was that you had some *magic numbers* in that formula, like 20. Numbers without an explanation of what they mean are called magic numbers and are [not allowed according to our style guide](#), because someone else reading your code will not know what they represent. The style guide directs us to put these into constants. Fill in the blanks in this revised version of the code that uses constants.

```
1  const double FLOUR_CUPS_PER_LOAF = _____;  
2  const double FLOUR_CUPS_PER_BAG = _____;  
3  // use these constants on the following line:  
4  int flourBags = _____;  
5  // temporary; you'll need to change this later to get pluralization correct  
6  cout << flourBags << " bags of flour" << endl;
```

Put this code in `main()` after the code computing the number of loaves.

## Step 4: Test your code

Run your program, and use the examples you developed in Step 1 to test your program. Remember that the input to your program is the number of **people**, not the number of **loaves**! Here's a table we used to help us test:

Input to program (number of people)	Loaves	Output (bags of flour)
	1	
	9	
	30	
	75	

## Roadmap and Timeline

From here, finish the project by adding one small feature at a time. These were the steps we followed to finish Project 1:

1. Compute number of loaves
  - i. Printed correct one of "loaf" or loaves" in the "You need to make" output line
2. Compute number of bags of flour
  - i. Print correct one of "bag" or "bags" of flour
  - ii. Repeat for all ingredients
3. Compute cost of bags of flour
  - i. Add cost of bags of flour to total and print total
  - ii. Repeat for all ingredients
4. Add header and footer messages
5. Test entire program using sample runs in [Sample Runs section](#)

- i. Use diffchecker to compare program output and sample run (details in Sample Runs section)
  - ii. Debug problems discovered
6. Submitted to autograder
  - i. Debug problems that autograder reveals
7. Read the [style guide](#) and verify code follows it

## Timeline

---

As an approximate timeline, you are on track if by:

- September 6: Your IDE is ready and you have created a project with the starter code
- September 7: The number of loaves and bags of flour (in this tutorial) are working in your IDE, one new ingredient and cost works as expected
- September 9: All ingredients added, program passes sample runs, score greater than 0 on autograder
- September 12: Debugging in progress, passes all sample runs, 80% or higher on autograder
- September 14: Make last submission to autograder for 5% extra credit

## Things to get used to when programming

---

### Spelling and spacing must match exactly

---

Computers pay attention to every little detail, so your program needs to match the sample outputs and the tests on the autograder exactly. This includes spelling, capitalization, and all symbols. The one exception is whitespace: the autograder will *ignore* differences in whitespace, like spaces and newlines. You will need to use a tool like [diffchecker.com](https://www.diffchecker.com) to find the differences in spelling, capitalization, etc., because humans aren't good at spotting them.

### It is normal when things don't work the first time

---

Most of the skill in programming is in debugging and finding problems, since everyone makes them!

### The autograder can find problems you didn't see

---

For this project, the best time to submit to the autograder is as soon as you pass one of the sample runs. In general, you can't assume that because the program works on your computer it will work on the autograder. Many times there are some adjustments that you need to make.

# Specification

Your task for this project is to create a program that creates a shopping list for focaccia bread. Your program will ask the user how many people they need to serve, and will output how much of each ingredient is needed and what the total cost of the ingredients will be.

## Ingredient List

Each loaf of focaccia requires:

- 5 cups flour
- 1 <sup>3</sup>/<sub>4</sub> teaspoons instant yeast
- 1 <sup>7</sup>/<sub>8</sub> teaspoons salt
- 2 tablespoons olive oil
- 2 cups water

## Ingredient Cost and Conversions

Package of ingredient	How to output in shopping list	Cost	Conversion
5 pound bag of flour	1 bag of flour	\$2.69	20 cups in 5 pounds
package of yeast	1 package of yeast	\$0.40	2.25 teaspoons per package
30 gram canister of salt	1 canister of salt	\$0.49	5 grams per teaspoon
500 milliliter bottle of olive oil	1 bottles of olive oil	\$6.39	14.8 millilitres per tablespoon
water	Do not include in shopping list	No cost. Do not include in total	

		cost.	
--	--	-------	--

## Pluralization

When the number of batches and the number of each ingredient is printed in the shopping list, the correct pluralization must be used. For example, "2 bottle of olive oil" is not correct, and needs to be output as "2 bottles of olive oil". As a hint, you can use the `pluralize()` function to help you output the correct forms in your program.

Remember that the plural of "loaf" is "loaves".

## Output Format

Your program's output must match the following template.

First, the program must ask the user how many people they wish to serve, and read in an integer from the user using `cin`. **Each loaf feeds 4 people.**

How many people do you need to serve?

Then, it should print the number of batches to make and the shopping list. You must make an integer number of batches and purchase an integer number of packages of each ingredient. The parts of the template that will change each run of your program are in bold red.

```
1  You need to make: 7 loaves of focaccia
2
3  Shopping List for Focaccia Bread
4  -----
5  2 bags of flour
6  6 packages of yeast
7  3 canisters of salt
8  1 bottle of olive oil
9
10 Total expected cost of ingredients: $15.64
11
12 Have a great party!
```

If the number of cents in the total expected cost is a multiple of 10, do not print the trailing zeros in the decimal. For example, instead of `$4.10`, you must print `$4.1` and instead of `$10.00`, you must print `$10`. You should not need to do anything for this to happen – if you use the default settings for `cout`, your code will pass this automatically.

# Sample Runs

## Diffchecker

Your program's output must *exactly match* the specification, including spelling, symbols, and capitalization. The only way to know whether the output is an exact match is to use a tool called a *diffchecker*, which shows you where differences are. The staff's favorite website for this is [diffchecker.com](https://www.diffchecker.com). Paste the output of a sample run below into "Original Text" and your program's output into "Changed Text", then click "Find Difference" to highlight any differences. Note: the autograder will show its own diffchecker - [diffchecker.com](https://www.diffchecker.com) is helpful for comparing the output on your computer to the expected output below, not when viewing the feedback on the autograder.

The input to the program is given in **bold red** in these sample runs.

## Sample Run 1

```
1  How many people do you need to serve? 1
2
3
4  You need to make: 1 loaf of focaccia
5
6  Shopping List for Focaccia Bread
7  -----
8  1 bag of flour
9  1 package of yeast
10 1 canister of salt
11 1 bottle of olive oil
12
13 Total expected cost of ingredients: $9.97
14
15 Have a great party!
```

## Sample Run 2

```
1  How many people do you need to serve? 5
2
3
4  You need to make: 2 loaves of focaccia
5
6  Shopping List for Focaccia Bread
7  -----
```

```
8  1 bag of flour
9  2 packages of yeast
10 1 canister of salt
11 1 bottle of olive oil
12
13 Total expected cost of ingredients: $10.37
14
15 Have a great party!
```

## Sample Run 3

```
1  How many people do you need to serve? 40
2
3
4  You need to make: 10 loaves of focaccia
5
6  Shopping List for Focaccia Bread
7  -----
8  3 bags of flour
9  8 packages of yeast
10 4 canisters of salt
11 1 bottle of olive oil
12
13 Total expected cost of ingredients: $19.62
14
15 Have a great party!
```

## Sample Run 4

```
1  How many people do you need to serve? 90
2
3
4  You need to make: 23 loaves of focaccia
5
6  Shopping List for Focaccia Bread
7  -----
8  6 bags of flour
9  18 packages of yeast
10 8 canisters of salt
11 2 bottles of olive oil
12
13 Total expected cost of ingredients: $40.04
14
```



15 Have a great party!

## Sample Run 5

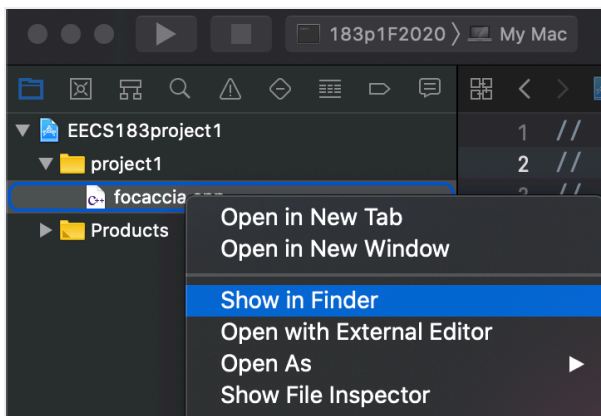
```
1  How many people do you need to serve? 2000
2
3
4  You need to make: 500 loaves of focaccia
5
6  Shopping List for Focaccia Bread
7  -----
8  125 bags of flour
9  389 packages of yeast
10 157 canisters of salt
11 30 bottles of olive oil
12
13 Total expected cost of ingredients: $760.48
14
15 Have a great party!
```

## How to Submit

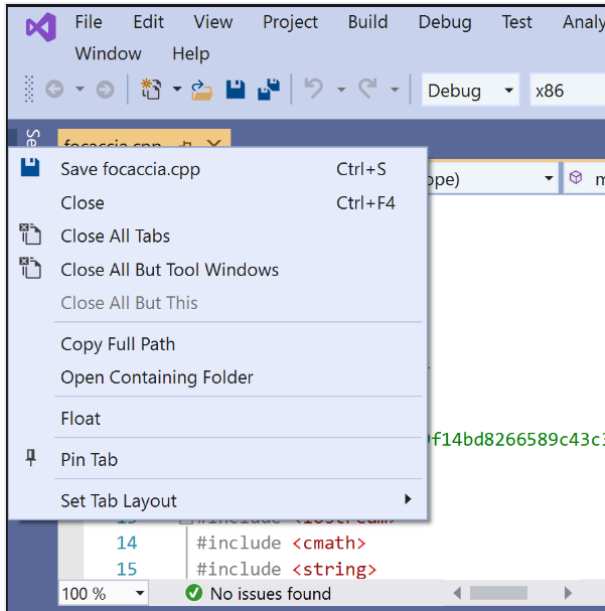
When you're ready to submit, visit the [autograder](#).

Click "Choose files to upload" and navigate to your file. The file you submit to the autograder **MUST** be called `focaccia.cpp`.

If you're using Xcode and don't know how where exactly `focaccia.cpp` is located on your disk, right-click (or click while holding down the Control key) on the file in the *Navigator area* on the left side of the Xcode window and choose **Show in Finder**.



If you're using Visual Studio and would like to know where `focaccia.cpp` is on your disk, right-click on `focaccia.cpp` in the tab above the Code pane and choose **Open Containing Folder**.



Once you've selected the correct file, click **Submit** to submit to the Autograder. The autograder will tell you if you did not select a file of the correct name.

## Style Guide

Your code must follow [the EECS 183 style guide, linked here](#).

## Project 1 Style Rubric

### Top Comment

Must have name, username, program name, and project description at the top of the file.

### Readability violations: -1 for each of the following categories

Category	Possible Violations
Indentations	<ul style="list-style-type: none"> <li>Not using a consistent number of spaces for each level of code indentation</li> <li>Not indenting lines at all <ul style="list-style-type: none"> <li>Failing to indent the blocks of code inside curly braces</li> </ul> </li> </ul>

Spacing	<ul style="list-style-type: none"> <li>• Not putting a space around operators (e.g. 5*7 instead of 5 * 7 or count=0; instead of count = 0;)</li> <li>• Not putting a space before and after the stream insertion and extraction operators (e.g cout&lt;&lt;loaves; instead of cout &lt;&lt; loaves;)</li> <li>• Spaces around parentheses <ul style="list-style-type: none"> <li>◦ Function headers and calls should not have a space between the function name and opening parenthesis.</li> </ul> <pre>// good pluralize("person", "people", people); // bad pluralize ( "person" ,"people" , people );</pre> </li> </ul>
Bracing (Scope)	<ul style="list-style-type: none"> <li>• Using a mix of Egyptian-style and hanging braces <ul style="list-style-type: none"> <li>◦ Egyptian-style: '{' at the end of a statement</li> <li>◦ Hanging: '{' on its own line</li> </ul> </li> <li>• Braces should always be used for conditionals, loops, and functions.</li> </ul>
Variables	<ul style="list-style-type: none"> <li>• Variable names not meaningful</li> <li>• Inconsistent variable naming style (camelCase vs. snake_case), excluding const variables, which are always SNAKE_CASE</li> <li>• Not using all uppercase SNAKE_CASE for const variable names</li> <li>• Using variable types that do not make sense in context</li> </ul>
Line Limit	<ul style="list-style-type: none"> <li>• Going over 80 characters on a line <ul style="list-style-type: none"> <li>◦ Includes lines of comments and lines of code</li> </ul> </li> </ul>
Statements	<ul style="list-style-type: none"> <li>• More than one statement on a single line <ul style="list-style-type: none"> <li>◦ A statement ends in a semicolon</li> </ul> </li> </ul>
Comments	<ul style="list-style-type: none"> <li>• Commenting on the end of a line of code <pre>// A comment should be placed before a line of code int count = 0; // not on the same line as the code</pre> </li> <li>• Insufficient comments or excessive comments <ul style="list-style-type: none"> <li>◦ Code should be thoroughly commented such that lines' functionality is apparent from comments alone or from quickly glancing at code</li> <li>◦ Example of appropriate comment: <pre>// calculate the area of the circle double area = pi * radius * radius;</pre> </li> </ul> </li> </ul>

- Example of excessive comments:  

```
// declare variable for area
double area;
// calculate area using the equation:
// area = Pi * r^2
double area = pi * radius * radius;
```
- Unneeded comments left in the code, such as:  

```
// your code goes here
// FIXED
```
- Commented out code, such as:  

```
// double area = 3.141527 * radius * radius;
double area = pi * radius * radius;
```

## Coding quality: -2 for each of the following categories

Category	Possible Violations
Global Variables	Global variables not declared as const
Magic Numbers	<ul style="list-style-type: none"> <li>• Integer and/or double literals used such that it is unclear to a reasonable reader what the literals represent <ul style="list-style-type: none"> <li>◦ 0, 1, and 2 never count as magic numbers</li> </ul> </li> </ul>
Egregious Code	<ul style="list-style-type: none"> <li>• Logic that is clearly too involved or incorrect <ul style="list-style-type: none"> <li>◦ e.g. instead of basing numbers on conversions, writing: <pre>1  if (loaves == 1) { 2      //do something 3  } else if (loaves == 2) { 4      //do something else 5  } // and so on</pre> </li> </ul> </li> </ul>
Function Misuse	Not calling pluralize() when appropriate











