House Navigation and Cake Baking Lab (C Programming)

Introduction

You are supposed to bake cakes. In order to do so you should be able to navigate through the stores of a house/building collecting ingredients. The house is represented by an 8×8 grid or matrix. The Kitchen is located at grid location (0,0). Your task is to collect baking ingredients using a bag and deliver them to the Kitchen for baking a given type of cake.

Lab Description

In this lab, you will **complete** a C program that simulates a small delivery robot or chef moving through a **house** to gather ingredients for a cake. The **house** is represented by an 8x8 grid with rooms and hallways. Some grid cells are open floor that can be traversed, while others are walls or obstacles that block movement. The Kitchen is located at the top-left corner of the grid (coordinate (0,0)) and where baking is done. The robot starts and ends its journey. Several ingredients (e.g. Flour, Sugar, Eggs, etc.) are placed in different rooms of the house; each ingredient's location on the grid is known. A **recipe** specifies which ingredients are needed (and in what order). The goal is to navigate from the Kitchen to each ingredient in the order specified on the recipe, collect them, and eventually return everything to the Kitchen for baking.

There are two scenarios for the recipes in this lab, which simulate different constraints:

- **Scenario A (Unweighted Recipe):** Ingredients are treated uniformly with no concept of weight or carrying capacity for the robot. The robot can carry all ingredients in one trip if it finds a path through the house. This scenario focuses on basic navigation and ensures you can collect items in order without worrying about load constraints.

- **Scenario B (Weighted Recipe):** Each ingredient is assigned a weight in kilograms. The robot's carrying bag has a limited capacity – specifically **60% of the total recipe weight (rounded up)**. This means the robot will likely **not** be able to carry everything at once, forcing it to make multiple trips. In this scenario, your program must decide when the bag is full enough to return to the Kitchen to drop off ingredients before continuing. This models a realistic constraint (you can't carry an arbitrarily large load) and introduces additional logic for planning multiple trips.

Pedagogical Goals: This lab gives you practice with multi-file C programs, file I/O and parsing, using 2D arrays and data structures to represent a grid, implementing control flow for navigation (loops, conditionals), and handling constraints (like carrying capacity). You will also practice validating input data and thinking about basic pathfinding strategies in a grid. By the end, you should understand how to break a larger problem into functions/modules, how to use a grid (2D array) to represent spatial problems, and how to manage state (like position and weight carried/lifted) throughout the duration of the program.

The reflection questions at the end will prompt you to think about the efficiency of your approach, the importance of input validation, and the benefits of modular program design.

Lab Setup and Provided Files

The lab materials include a number of files and directories. Below is an overview of the provided structure:

- **`house_layout.txt`** – An 8×8 grid of numbers (either `0` or `1`) defining the house's floor plan. Here `1` indicates an open, traversable cell, and `0` indicates a blocked cell (a wall or obstacle). This file essentially maps out where the robot can move.
- **`ingredients_map.txt`** – A list of ingredient locations in the house. Each line of this file contains an ingredient name and its coordinates given as an ordered pair `(row,column)`. For example, a line might be `Flour 5 1` meaning the ingredient "Flour" is located at row 5, column 1 on the grid. The special entry `"Kitchen"` is also included in this file with its coordinates, so you know where the Kitchen is (should be 0 0). You will use this map to look up where each ingredient is located.
- **`recipes/`** – A directory containing recipe files for both scenarios. There are five *Scenario A* recipe files (unweighted) and five *Scenario B* recipe files (weighted). In a Scenario A recipe file, each line contains **only the name** of an ingredient (in the order they should be collected). In a Scenario B recipe file, each line contains an ingredient name followed by an integer weight. For example, a Scenario B recipe line might read `Eggs 2` to indicate "Eggs" with weight 2 (kg). The recipe files are named accordingly

  (for instance, `recipeA1_VanillaCake.txt` vs. `recipeB1_VanillaCake.txt` for the same cake in scenarios A and B).
- **`src/` and `include/`** – These folders contain the C source code (.c files in `src/`) and header files (.h files in `include/`) for the lab. The code is split into multiple modules (files), and **several functions are left unimplemented as TODOs** that you will complete. The skeleton code is provided to handle reading input files, printing output, and some structure of the program, but key parts of the logic are missing (marked with comments like `// TODO:` in the source). You will fill in those parts during the lab.

- **`Makefile`** – A build script with instructions to compile the program. It defines how to produce the executable (likely named **labhouse**) from the source files. You can simply run `make` to compile the program.
- **`README.md`** – (This document) A detailed description of the lab (goals, instructions, tasks).
- **`answers.txt`** – A text file where you will write answers to the reflection questions at the end of the lab. Be sure to fill this in with your responses to **Q1, Q2, Q3** before submission.

Building and Running the Program

To compile the program, navigate to the project directory and run the provided Makefile by typing `make` in the terminal.

```
make                      # compile the program
```

This will produce an executable (for example, called **`labhouse`**).

Once compiled, you can run the program from the command line. The program expects **two command-line arguments**: first the ingredients map.txt file, and second a recipe .txt file of your choice.

For example, to run the lab with the provided ingredients map and the first Vanilla Cake recipe for each scenario, you would use:

```
make                      # compile the program


./labhouse ingredients_map.txt recipes/recipeA1_VanillaCake.txt   # Run ScenarioA

./labhouse ingredients_map.txt recipes/recipeB1_VanillaCake.txt   # Run Scenario B
```

The above commands will run the program for Scenario A (unweighted) and Scenario B (weighted) respectively. Of course, you can try any of the other provided recipe files as well— just specify the path to the desired recipe file. Make sure the `house_layout.txt` and `ingredients_map.txt` files are in the same directory (or provide the correct path) since the program will read those internally as well.

Usage: The general usage pattern is:

```
./labhouse <ingredients_map_file> <recipe_file> [flags]
```

If you run the program without the correct arguments, it will likely print a usage message or error. Always provide the two required file arguments. The program reads the house layout and ingredient map from the given filenames, then reads the specified recipe and begins the navigation simulation.

Visual and Animation Modes (Optional)

For better insight or debugging, the program can be run in a **visual/verbose mode**. There are optional flags you can add as a third argument to enable these features: - **Visual Mode (-v)** – If you run the program with the -v flag

for example:

```
./labhouse ingredients_map.txt recipes/recipeB1_VanillaCake.txt -v)
```

```
./labhouse ingredients_map.txt recipes/recipeA1_VanillaCake.txt --animate --delay=200
```

```
./labhouse ingredients_map.txt recipes/recipeB2_ChocolateCake.txt --animate --delay=100 --log=run.
```

The program will output a step-by-step visualization of the robot's movement through the house. In this mode, the program may print the 8x8 grid for each move, marking the robot's current position, so you can actually see the path it takes around walls and corridors. This is useful for understanding how your navigation algorithm is working. –

Animation Delay (-a) – You can combine a second flag (for example, -a) to introduce a short delay between steps, making the visual output animate slowly. For instance, running `./labhouse ingredients_map.txt recipes/recipeB1_VanillaCake.txt -v -a` would show the grid updates with a pause (say, half a second) on each move, creating an animation effect. This doesn't change the logic; it only slows the output for easier following of the robot's journey.

Note: These flags are optional and for user experience; they are not required for the core functionality. You can use them to debug your pathfinding or just to watch the robot in action. If you omit them, the program will simply run to completion and print the results without the step-by-step grid display.

Rules and Constraints

Before diving into implementation, it's important to understand the movement rules and scenario constraints that your program must respect:

- **Grid Movement:** The robot can move only in the four cardinal directions: **UP, DOWN, LEFT, RIGHT** (no diagonal moves are allowed). Each move changes the robot's position by one cell in the specified direction.
- **Movement Command (`move(direction, steps)`):** You will implement a function `move(int direction, int steps)` that moves the robot step-by-step in the given direction, the specified number of steps. The move should be executed one step at a time: if any single step is invalid (e.g., hits a wall or goes out of bounds), then the move is

considered a **dead-end** and should stop immediately. Hitting a dead-end means the path you tried is blocked.

- **Dead-End Handling:** If a path to an ingredient is blocked partway (i.e., a dead-end is encountered during movement), the robot must abandon that attempt and **restart from the Kitchen (0,0)**. In other words, whenever you hit an obstacle that prevents reaching the target ingredient via the current route, you should send the robot back to the kitchen and try a different route to that ingredient (for example, take a different path around the obstacle). The program should not consider an ingredient "collected" until a valid path is found.
- **Scenario A (No Bag Limit):** In this scenario, the robot's carrying bag has **no weight limit**. The robot can collect all ingredients in one continuous trip (assuming it can navigate between them) without needing to return to drop off items until the very end. The focus here is purely on finding a path through the house for the given order of ingredients.
- **Scenario B (Limited Bag Capacity):** In this scenario, each ingredient has an associated weight from the recipe file. The robot's bag can only hold so much weight at a time. The **bag capacity is set to 60% of the total weight of all ingredients in the recipe (rounded up)**. This artificial limit ensures that the robot **must make multiple trips** to gather everything (you cannot just load all items at once). The program should keep track of the current bag weight as ingredients are collected. **If adding the next ingredient would exceed the capacity, the robot must return to the Kitchen to drop off the items collected so far**, then go out for the remaining ingredients. Dropping off resets the current carried weight to 0 (but of course the ingredients already delivered remain safely in the Kitchen). The robot may make multiple delivery trips until all ingredients are eventually collected.
- **Path Planning:** You do not need to pre-compute an optimal or shortest path for the whole route; it is acceptable to navigate ingredient by ingredient, always returning to the Kitchen when necessary (especially in Scenario B). However, for each single ingredient, you do need to find *a path* from the current location to that ingredient's location. The program logic should handle trying alternate routes if the first attempt hits a dead-end. The simplest strategy is to attempt one route (say, moving horizontally first then vertically), and if that fails, try a different route (vertically first then horizontally). For more complex layouts, a more systematic search (like a breadth-first search) might be needed, but in this lab a simple trial-and-error as described is sufficient to find paths in the given maps.
- **Starting/Ending Conditions:** The robot always starts at the Kitchen coordinate (0,0) at the beginning of the program. In Scenario B, every time the robot drops off items, it effectively starts again from (0,0) for the next trip. At the very end of the recipe, after collecting the last ingredient, the robot should end up back at the Kitchen (either because it returned with the last items, or the last ingredient happened to be in the Kitchen). This ensures all ingredients are delivered for baking. All grid coordinates in this lab are **0-based indices** (rows and columns 0 through 7), with (0,0) being the topleft cell of the house grid.

Marking Criteria (75 pts)

Your lab solution will be evaluated according to the following breakdown:

- **`move()` function implementation** –15 pts
- **`collect()` function implementation** – 20 pts
- **File parsing (reading the house layout, ingredients map, and recipes)** – 10 pts
- **Multiple-trip logic for Scenario B** – 10 pts
- **Program correctness and code style** – 5 pts
- **Answers to Q1, Q2, Q3 (in answers.txt)** – 5 + 5+ 5pts

Each of the first five items corresponds to a feature or function you must implement in C. "Program correctness" covers overall behavior (does your program produce the expected output, handle edge cases, and avoid crashes) as well as coding style (readability, comments, following the C style guidelines given in class). The three questions in the answers.txt are worth 5 points each, so be sure to provide thoughtful answers. Strive to complete all parts for full credit.

(Remember: even if your code isn't fully working, submit whatever you have by the first deadline to get partial feedback, and always write something for the answers – you can earn points for reasoning even if your code is incomplete.)

Program Structure and Modules

The codebase is divided into multiple modules to organize functionality. Understanding the role of each module will help you know where to implement each part of the lab:

- **`main.c`** – This contains the `main()` function which orchestrates the program. It parses command-line arguments (the file names for the map and recipe), calls the initialization routines to load the data from files, and then drives the overall process of going through the recipe and collecting items. The main loop that iterates over ingredients in the recipe is here. It will call functions like `move()` and `collect()` as needed. It also handles printing final results and messages to the user.
- **`house.c` / `house.h`** – This module manages the house grid and the robot's position. It likely contains a 2D array for the house layout (8x8 int matrix loaded from `house_layout.txt`), and possibly variables for the robot's current row and column. The **movement logic** is implemented here – in particular, the `move(int direction, int steps)` function is defined in this module. It may also include helper functions

  for movement (e.g., checking if a given coordinate is within bounds or if a cell is open) and possibly a function to reset the robot's position to the Kitchen. This module is essentially the navigation engine of your program.
- **`map.c` / `map.h`** (or **`ingredients.c`**) – This module is responsible for reading and storing the ingredient locations from `ingredients_map.txt`. It defines a structure or data type to hold an ingredient (for example, a struct with fields for name and coordinates). When

initialized, it will parse the ingredient map file line by line, create an entry for each ingredient (and the Kitchen), and store them in an array or list. This module likely provides a function to lookup an ingredient's coordinates by name (e.g., `find_ingredient(char *name)` that returns the row and column for that ingredient). This lookup will be used when the program processes the recipe to know where each item is.

- **`recipe.c` / `recipe.h`** – This module handles reading the recipe file and managing the list of ingredients to collect. It will parse the given recipe file: for Scenario A, it might just store a list of ingredient names in order; for Scenario B, it will store a list of (name, weight) pairs. It also likely computes the **total weight** of the recipe as it reads (for Scenario B) so that the bag capacity can be determined. A global or static variable for `bag_capacity` might be set here (or in main after reading). The module could contain logic for the carrying capacity as well. Additionally, **input validation** might be handled here: for example, the `validate_ingredients()` function (ensuring every ingredient in the recipe exists on the house map) could be part of this module. It may also provide an API to get the next ingredient to collect, or to iterate through the recipe list.
- **Other modules:** The lab might contain other small modules or all I/O in one place. For instance, there could be a `util.c` for utility functions (like a custom string trim or something) or the file parsing might be split differently. However, the core concepts remain: you have separation between *data loading* (house layout, ingredient locations, recipe info) and *operations* (moving and collecting). The provided skeleton will have clearly marked places (via TODO comments in each file) where you need to write code. Use the structure as a guide: implement the missing logic in the appropriate file as indicated.

Next, we will walk through each major function and task you need to implement, explaining what it should do and offering guidance on how to do it.

Task 1: Implement `move(direction, steps)` Function

One of the first tasks is to implement the movement function, `move(int direction, int steps)`, in the appropriate module (likely **house.c**). This function is critical: it commands the robot to move in the given direction a certain number of steps on the grid, and handles checking for obstacles.

Function Purpose: `move()` should attempt to move the robot from its current position in a straight line. The `direction` is probably encoded as an integer constant (for example, the code may define `#define UP 0`, `RIGHT 1`, `DOWN 2`, `LEFT 3`, or something similar). The `steps` parameter tells how many grid cells to move in that direction.

What to Implement: You need to update the robot's current position step by step, one cell at a time, for the given number of steps. For each step, do the following:

1. **Calculate the Next Cell:** Based on the current position (`curr_row, curr_col`) and the `direction`, compute the coordinates of the next cell. For example:
2. If moving UP (0), next cell is (`curr_row - 1, curr_col`).
3. If moving RIGHT (1), next cell is (`curr_row, curr_col + 1`).
4. If moving DOWN (2), next cell is (`curr_row + 1, curr_col`).
5. If moving LEFT (3), next cell is (`curr_row, curr_col - 1`). These codes are assumptions; check the actual definitions in the code. The principle is that you change the row or column by ±1 depending on direction.

6. **Check Bounds and Obstacles:** Before "stepping" into the next cell, verify that:

    a. The next cell is within the grid bounds (0 ≤ row < 8 and 0 ≤ col < 8).
    b. The next cell is not a wall/blocked cell. You will use the `house_layout` matrix loaded from file to check this. If `house_layout[next_row][next_col]` is `0`, that means a wall and the move cannot continue. If either condition fails, the step is invalid – you've hit a wall or boundary.

7. **Handle Dead-End:** If an invalid step is encountered, `move()` should **stop immediately** and signal that the move did not complete. How to signal? The function could return an integer or boolean (for example, return 0/false for failure, 1/true for success), or perhaps update a global state or error flag. Check how the skeleton expects it. Most likely, `move()` returns an `int` or `bool` where a false/zero value indicates a dead-end was hit. In that case, you should *not* update the position for that step (i.e., the robot stays where it was before the move), and exit the function early. None of the remaining steps in that move command should be attempted once a dead-end is found.

8. **Make the Move:** If the next cell is valid, update the robot's `curr_row` and `curr_col` to this new cell (the robot actually moves one step). If you are in visual mode, you might also print the grid or the move at this point (e.g., "Moved to (r,c)"). Then continue to the next step in the loop.

9. **Complete the Move:** If all steps were executed without hitting a wall, then the move function succeeded in moving the robot the full requested distance. It should return a success indicator (e.g., true). The robot's position will now be the new location after moving those steps.

Important: Because movement is done step-by-step, you effectively ensure that the robot cannot "skip" over walls. Even if asked to move 5 steps to the right, if there is a wall 2 steps to the right, your function will catch it on step 3 and stop.

Hint: Use a loop from 1 to `steps` to process each single-step move. Also, consider using a small table of direction vectors. For example, you could define two arrays `dRow[4] = {1,0,1,0}` and `dCol[4] = {0,1,0,-1}` corresponding to the row/col change for directions

0,1,2,3 respectively. Then for each step do `curr_row += dRow[direction]`, `curr_col += dCol[direction]`, and then check bounds and cell value. This avoids writing a long switchcase.

After implementing `move()`, test it in isolation if possible: try calling `move()` to go right by 1 from (0,0) on an empty layout, or into a wall to see if it detects properly. A robust `move()` is the foundation for the navigation – the higher-level logic will call this repeatedly to navigate corridors.

Task 2: Implement `collect()` Function

The next function to implement is `collect()` (or it might be named `collect_item()` in your code; refer to the skeleton). This function handles the action of picking up an ingredient once the robot has reached the correct location. In the simulation, "collecting" an item likely means updating some state (like the bag's current weight for Scenario B, or marking that ingredient as collected) and printing a confirmation message.

Function Purpose: `collect()` should be called when the robot is currently at the location of the next required ingredient (i.e., the robot's `curr_row, curr_col` matches the coordinate of that ingredient). This function will simulate picking up the item.

What to Implement:

- First, ensure that the robot's current position actually matches the expected ingredient location. In a correct program flow, this should always be true (you would only call `collect()` after successfully navigating to the ingredient). But it's good practice to double-check. If the positions don't match, it means something went wrong in navigation (or programming logic).
- Retrieve the name of the ingredient being collected. The main program likely knows which ingredient is next (perhaps stored in a variable or accessible via the recipe structure). It may pass this info to `collect()` or it might be globally accessible (e.g., a pointer to the current recipe item).
- If Scenario B (weights are used): add the weight of this ingredient to the current bag load. There might be a global or external variable tracking `current_bag_weight` and another for `bag_capacity`. Update `current_bag_weight += item_weight`. If you update the weight, you might also print a message like `"Collected Sugar (3 kg). Current bag weight: 8 kg out of 9 kg."` to inform about the load (this is optional but helpful output).
- If Scenario A: weights aren't considered, so you might skip the weight update. You could still count items if needed (e.g., count how many items picked so far, but that's not necessary unless for output).
- Mark the ingredient as collected. How you do this depends on program design:
- You might remove it from the list of pending items, or
- Set a flag in the ingredient map data (for example, a boolean `collected` field in the ingredient struct),

- Or simply move on without explicitly marking (since you iterate through recipe in order, and won't come back to it). In many cases, no special marking is needed beyond advancing the recipe index, because each ingredient is collected exactly once in the given order.
- Print an output message to the user indicating the collection. For example: `Collected Sugar at (4,7).` or simply `Sugar collected.` The skeleton might already have a printf for this event, or you might add one. Clear feedback is useful.

After `collect()` executes, the robot is holding that item (or has dropped it in the kitchen, depending on if it immediately goes back – see next tasks). The program will then proceed to either the next ingredient or some other action if needed.

Note: In Scenario B, do **not** automatically drop off the item in `collect()`. Dropping off (returning to Kitchen) is a separate decision that happens *before* collecting a new item that would overflow capacity. The `collect()` function itself should just pick up whatever item is at hand. The logic to decide to return to Kitchen will be handled in the main loop (Task 5 below). In other words, `collect()` doesn't concern itself with *when* to return; it only handles picking up the item and updating the carried weight.

Potential Pitfall: Make sure that when you collect an ingredient, you're collecting the correct one. A common error would be mismatching the recipe order and the actual location. This is mitigated by verifying that the current position matches the ingredient's map coordinates (which ideally you did in Task 1 by moving there). Also, ensure you use the right data: for instance, if you maintain an index `current_recipe_index`, use that to identify which ingredient name and weight (if any) to use at this step.

Once `collect()` is implemented, your program will be able to simulate picking up items. Test idea: If possible, manually set the robot's position to a known ingredient coordinate and call `collect()` to see if it prints the correct message and updates weights properly.

Task 3: Implement `validate_ingredients()` Function

The function `validate_ingredients()` is responsible for input validation – it checks that the recipe is valid given the ingredient map. This should be done **after** reading the recipe and the ingredient map, but **before** attempting to navigate and collect items. The purpose is to catch any errors in the input files early (for example, a recipe asking for an ingredient that doesn't exist on the map).

Function Purpose: Ensure that every ingredient listed in the recipe file actually has a corresponding entry (name and coordinates) in the `ingredients_map.txt` data. If any ingredient is missing, that's an error and the program should probably abort with a message rather than attempt a futile search.

What to Implement:

- Iterate over each ingredient name in the loaded recipe list.
- For each name, search through the list/array of ingredients loaded from `ingredients_map.txt` to see if a match is found.
- You might have stored the ingredients in an array of structs (with a name field). Loop through this array and compare strings.
- Alternatively, if you built a dictionary or map structure, you could just do a lookup (but likely for simplicity it's an array and a linear search, since the number of ingredients is small).
- If you find a matching name in the map for the recipe item, great – proceed to the next recipe item.
- If **no match** is found for a recipe item:
- Print an error message to `stderr` or console, for example: `"Error: Ingredient 'Vanilla' in recipe not found on the map."`
- Return an error code (like `false` or `-1`). The main program should handle this by exiting (and possibly informing the user that the input files are inconsistent).
- Also consider the reverse: the recipe might be a subset of items in the house. That's fine – the map can have extra items not used. You generally don't need to warn about that. The main concern is missing ones that are needed.
- If all ingredients validate (found in map), return success (true). The program will then proceed.

This function should be called early in `main` after loading the recipe and map. It ensures that subsequent functions (like move and collect) won't run into a scenario where they look for an ingredient coordinate that doesn't exist. It's much easier to handle this check up front than to debug why your navigation is failing later (when the item wasn't even there!).

Additional Thoughts: This is a good place to also check for other potential issues: - If the recipe is empty (no ingredients), you might decide what to do (probably just output "Nothing to collect!" and finish). - If "Kitchen" somehow appeared in the recipe (it shouldn't, but if someone put it in by mistake), you might handle that as an error or simply ignore it since you start there anyway. - If duplicate ingredient names appear in the recipe, is that allowed? Possibly not in a real recipe (you wouldn't list "Eggs" twice, you'd specify quantity), but if it did, and if your program design can't handle collecting the same ingredient twice, you might at least warn. However, this is an edge case likely not present in provided files.

By implementing `validate_ingredients()`, you follow a key programming principle: **"fail fast"** on bad input. This makes your program more robust and easier to use.

Task 4: Parse Input Files (House Layout, Ingredient Map, Recipe)

Parsing the input files is a fundamental step that sets up all the data your program will use. Some of the parsing code might be partially provided, but you are responsible for completing it and storing the data in appropriate structures. This task doesn't correspond to a single function but rather a set of functions or code blocks that read each file.

House Layout (`house_layout.txt`): This file should be read into an 8x8 integer array (e.g., `int house_layout[8][8];`). Each line of the file contains 8 digits (separated by spaces possibly) which are either 0 or 1. You need to open the file, and read it line by line, and within each line, parse 8 integers. This can be done with `fscanf` or by reading the whole line and using `sscanf`/`strtok`. After reading, you'll have the matrix filled with 0s and 1s. This matrix will be used by `move()` to check walls.

For example, the file might look like:

```
1 1 1 1 1 1 1 1
1 0 0 0 1 0 0 1
1 0 1 1 1 1 0 1
1 0 1 0 0 1 0 1
1 1 1 0 1 1 1 1
1 0 1 0 1 0 0 1
1 0 1 1 1 1 0 1
1 1 1 1 1 1 1 1
```

Here `0`s form some walls in the interior. Your parser should populate the 2D array exactly as in the file. Typically, row 0 in the array corresponds to the first line of the file.

Ingredient Map (`ingredients_map.txt`): Each line in this file has an item name followed by two numbers (row and column). You'll need to parse each line: read the string (ingredient name) and the two integers. Consider using `fscanf(file, "%s %d %d", name, &r, &c)` in a loop, or use `fgets` and then `sscanf`. Be mindful that ingredient names might be multiple words (though likely not – "BrownSugar" might be one word, but if spaces appear in names, `%s` will cut at space, which could be an issue. Given the context, names are probably one token without spaces). The Kitchen entry could be something like:

```
Kitchen     0  0
Flour       5  4
Sugar       4  7
Eggs        6  4
Butter      2  2
```

(This is just an example; actual coordinates may differ.)

As you parse, you should create a data structure for each entry. A simple approach: define

```
struct Ingredient {
    char name[50];
    int row;
    int col;
    int weight;
                  };
```

The weight field can be set to 0 or ignored for map entries (since the map itself doesn't contain weight, except Kitchen which has none). You can have an array `Ingredient ingredients[MAX_ITEMS]` and an integer `num_ingredients` count. Append each parsed ingredient to the array. After reading the file, you'll have all ingredient locations accessible. You may also want to store the Kitchen coordinates separately (or find it from this array when needed).

Recipe File (`recipe*.txt`): The parsing here depends on scenario: - For Scenario A (unweighted): each line is just a name. You can read line by line and store the names in order in, say, an array `char *recipe_list[MAX_ITEMS]` or an array of Ingredient structs (re-using the struct but ignoring weight). - For Scenario B (weighted): each line has a name and a weight. Similar to the map, you can parse the name and weight. You might store the recipe as an array of Ingredient structs as well, or parallel arrays (one for names, one for weights). Storing as the same `Ingredient` struct (with the understanding that row/col might not be filled here) could be convenient: you'll match recipe items to map items by name anyway.

After reading the entire recipe, compute the **total recipe weight** (for Scenario B) by summing the weights. Then compute the bag capacity = `ceil(0.6 * total_weight)`[2]. For example, if total weight is 10, 60% is 6, so capacity = 6; if total is 11, 60% is 6.6, round up to 7. If using integers, one way to round up is: `capacity = (total_weight * 6 + 9) / 10;` (this multiplies by 6, divides by 10 with an addition to ensure rounding up for any remainder). Or use a `ceil` function on a float cast. The capacity should be an integer. This value will be used to decide when to force return trips.

Integration: Likely, the skeleton calls these parsing functions in `main()` before doing anything else. For instance:

```
load_house_layout("house_layout.txt");
load_ingredients_map("ingredients_map.txt");
load_recipe(recipe_file); validate_ingredients();
```

It might not separate them all, but conceptually it does this. Your task is to implement inside those functions (or in `main` directly) the actual reading logic as described.

Make sure to handle file opening errors: e.g., if `fopen` returns NULL, print an error `"Could not open file X"` and exit.

After parsing, you should have: - `house_layout[8][8]` filled. - An array/list of `Ingredient` for the map (including Kitchen). - A list of recipe items (with weights if any). - `bag_capacity` (for scenario B) computed. - `current_bag_weight = 0` initialized. - Possibly a variable tracking which scenario mode it is (or you deduce scenario B if any weight > 0 or if weight data was read).

Finally, call `validate_ingredients(recipe, map)`. Implemented in Task 3, it will ensure all recipe items have a matching map entry. Only proceed if it returns true.

This task is mostly about correctly reading and storing data, which might not be glamorous but is absolutely necessary for the rest to work. Test the parsing by printing the loaded data (just during development). For example, after loading the map, loop and print all ingredients and coordinates to confirm. Similarly for the recipe.

Task 5: Implement Multiple-Trip Logic (Scenario B)

This part of the lab deals with the planning required in Scenario B, where the robot can't carry everything at once. The skeleton of the program likely handles Scenario A and B slightly differently. You need to fill in the logic that makes the robot return to Kitchen at the appropriate times when carrying capacity is exceeded.

Understanding the Requirement: We have a `bag_capacity` (computed earlier) which is a fraction of total recipe weight[2]. We also track `current_bag_weight` as the robot collects items. In Scenario B, before attempting to go for the *next* ingredient, the program should check: if I add that next item's weight to my current load, will it exceed `bag_capacity`? If yes, then I should **pause and return to Kitchen now, to unload, before picking that item.**

Essentially, the robot will make multiple trips. For example, if the recipe items are [Flour(5), Sugar(3), Butter(4), Eggs(2)] and capacity is 9: the robot could carry Flour and Sugar (total 8) in the first trip, but adding Butter (4) would make 12 which is >9, so it should go home after Sugar, drop off (current_bag_weight resets to 0), then go out for Butter and maybe Eggs in the next trip.

What to Implement: This logic likely resides in the main loop where you iterate over recipe ingredients. Here's how to approach it:

```
current_bag_weight = 0;
trips = 1;
for each ingredient in the recipe list:
   weight = (scenarioB ? ingredient.weight : 0);
   if scenarioB:
      if current_bag_weight + weight > bag_capacity:
         // Need to drop off before collecting this item
```

```
        // 1. Go back to Kitchen
        navigate from current position back to (0,0)
        // (you can implement this by determining how to move back:
        // one simple way: calculate difference in row and col,
        // and call move() in the opposite directions accordingly.)

        // 2. Print a message about returning (optional)
        printf("Returned to Kitchen to drop off items. (Trip %d complete) \n", trips);

        // 3. Increment trip counter
        trips += 1;

        // 4. Reset current bag weight
        current_bag_weight = 0;

    // Now proceed to collect the next ingredient
    determine ingredient’s coordinates (r, c) from map

    // navigate from current position (which is either Kitchen if we just dropped off,
    // or the last item's location if we didn’t drop) to (r, c):
    plan a path: e.g., move horizontally diff and vertically diff, try alternate if dead-end.

    if navigation fails totally:
      print "Ingredient X at (r,c) is unreachable!" and break/exit.
    else:
      collect(item);  // pick it up, which also adds weight if scenarioB

      if scenarioB:
        current_bag_weight += weight;
      // current position is now (r,c) after collecting
```

The pseudocode above outlines where to insert the check. Essentially, **before** you go for an item, if it doesn’t fit, you go home first.

Navigating back to Kitchen: This is something you also need to implement. A simple method is to reuse the `move()` function. Suppose you are at position `(x,y)` and need to return to `(0,0)`. You could move vertically and horizontally back to origin: - If `x > 0`, call `move(UP, x)` to go up x steps (this should bring row to 0 if unobstructed). - If `y > 0`, call `move(LEFT, y)` to go left y steps (this brings col to 0). This will attempt a direct straight path. However, if there are walls, a direct straight move might not work (the same pathfinding challenge exists in reverse). In many house layouts, going straight back may work if the path you came through is open, but if not, you might need to find an alternate path to return as well. In practice, since you got to `(x,y)` in the first place, presumably you can retrace that route back. A robust approach would be to store the path taken to get there and then replay it in reverse. But that's an overcomplication for our level. We can assume for simplicity that moving back in a straight line is possible because likely the house layout allows a direct or at least one-turn path (or you can just attempt similar trial-and-error as forward). If a direct move back hits a dead-end, you can try going one axis at a time in swapped order (horizontal then vertical vs vertical then horizontal). It's rare that you'd need more complex routing to go back if you could come in, since at least one of those orders should succeed unless the path is truly winding.

Once you successfully return to (0,0), update the current position to Kitchen and carry on.

Incrementing Trip Count: Keeping track of `trips` is optional but nice for output or answers (maybe one question asks how many trips for a given recipe). You can increment each time you do a drop-off. The first trip starts when you leave the kitchen initially, so perhaps initialize `trips = 1` at start, and increment when you drop off mid-way. At the very end, `trips` will count how many times you actually went out from the kitchen. If you had to drop off multiple times, trips will be >1.

Output messaging: It's helpful to log when a drop-off happens. For example:

```
Bag capacity exceeded if adding Butter (4). Returning to Kitchen to drop off c
urrent      items.
Dropped off ingredients. (Trip 1 complete, 2 items delivered.)
```

Then when you pick Butter and Eggs in second trip:

```
Picked                                up                                 Butter.
Picked                                 up                                  Eggs.
Returned to Kitchen with remaining items. (Trip 2 complete)
```

Finally:

```
All ingredients collected in 2 trips.
```

The exact wording is up to you; ensure it's clear. The skeleton might have some messages predefined (check for any printf or puts in the code around where capacity is handled).

After implementing this logic, test on a Scenario B recipe where you know multiple trips are needed. For instance, if one recipe's total weight is 10 and capacity is 6, and items weights are [4,3,3] in order: it should take at least 2 trips (4+3 =7 >6 so drop after first maybe, etc.).

Example Run and Output

To illustrate how all these pieces come together, let's walk through a sample execution for each scenario after and before implementation. Suppose the house layout and ingredient map are set up such that:

- Kitchen at (0,0)
- Flour at (5,1)
- Sugar at (4,7)
- Butter at (2,2)
- Eggs at (6,3)

And we have a **Vanilla Cake** recipe: - Scenario A file: just the names in order:

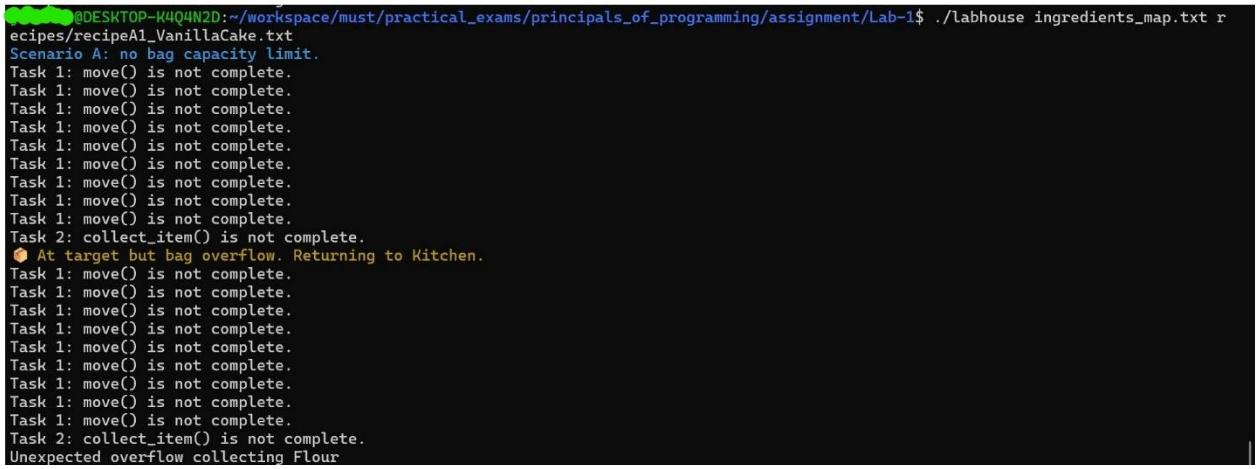
```
Flour
Sugar
Butter Eggs
```

- Scenario B file: names with weights (in kg, for example):

```
Flour 5
Sugar 3
Butter 4
Eggs 2
```

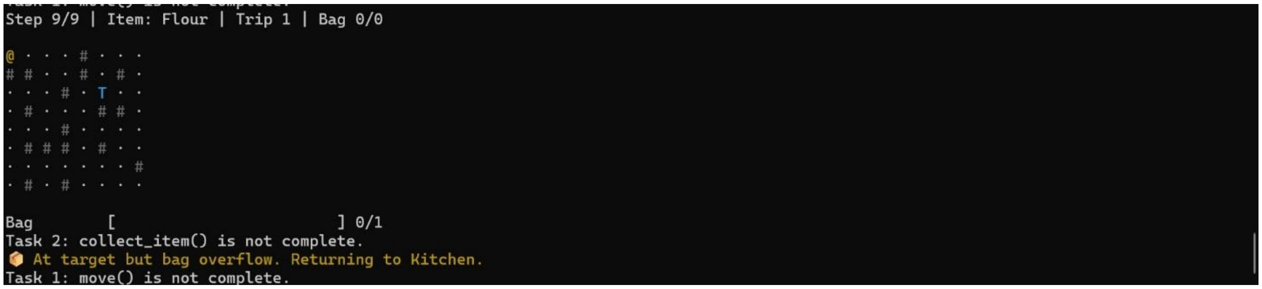
(Total weight = 14, so capacity = ceil(0.6*14) = ceil(8.4) = 9) **Sample**

Output for Scenario A without To/dos implemented :



```
[redacted]@DESKTOP-K4Q4N2D:~/workspace/must/practical_exams/principals_of_programming/assignment/Lab-1$ ./labhouse ingredients_map.txt r
ecipes/recipeA1_VanillaCake.txt
Scenario A: no bag capacity limit.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 2: collect_item() is not complete.
📦 At target but bag overflow. Returning to Kitchen.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 1: move() is not complete.
Task 2: collect_item() is not complete.
Unexpected overflow collecting Flour
```

Output for Scenario A run with Visual flags without ToDos implemented :



```
Task 1: move() is not complete.
Step 9/9 | Item: Flour | Trip 1 | Bag 0/0

@ · · · # · · ·
# # · · # · # ·
· · · # · T · ·
· # · · · # # ·
· · · # · · · ·
· # # # · # · ·
· · · · · · · #
· # · # · · · ·

Bag       [                         ] 0/1
Task 2: collect_item() is not complete.
📦 At target but bag overflow. Returning to Kitchen.
Task 1: move() is not complete.
```

Output for Scenario A after implementation:

```
Starting at Kitchen (0,0).
Moving to Flour at (5,1)... Collected
Flour.
Moving to Sugar at (4,7)...
Collected Sugar.
Moving to Butter at (2,2)...
Collected Butter.
Moving to Eggs at (6,3)...
Collected Eggs.
Returned to Kitchen (0,0) with all ingredients.

All ingredients collected! Cake baked successfully.
```

In this scenario, the robot never needed to drop off until the end, so it went through all ingredients in one continuous route (the order given). The output shows each move/collection and final success.

Sample Output for Scenario B:

```
Starting at Kitchen (0,0). Bag capacity is 9.
Next ingredient: Flour (5) at (5,1)
Moving to Flour... collected Flour. Current load = 5/9.
Next ingredient: Sugar (3) at (4,7)
Moving to Sugar... collected Sugar. Current load = 8/9.
Next ingredient: Butter (4) at (2,2)
Cannot carry Butter (4) now; bag would exceed capacity (8+4 > 9).
Returning to Kitchen to drop off items. Dropped off all items.
(Trip 1 completed) Current load = 0/9.
Next ingredient: Butter (4) at (2,2)
Moving to Butter... collected Butter. Current load = 4/9.
```

```
Next ingredient: Eggs (2) at (6,3)
Moving to Eggs... collected Eggs. Current load = 6/9.

All ingredients collected over 2 trips! Ready to bake in the Kitchen.
```

Here you can see the program decided to return to Kitchen before collecting Butter, because at that point carrying Butter on top of Flour + Sugar would exceed capacity. It completed the first trip (delivered Flour and Sugar), then went out again for Butter and Eggs. The messages track the bag load and trips.

Note: The actual output format may vary; this is just an illustrative example. Your implementation can combine messages or format them differently, but it should convey the key events: movement, collection, drop-offs, and completion.

Reflection Questions (to answer in `answers.txt`)

Now that you have implemented the lab, take some time to think about the following conceptual questions in the answers.txt. Provide your answers in the **answers.txt** file. Each answer should be a few sentences that explain your reasoning

- **Q1.** The navigation strategy in this lab is relatively simple (trying moves in a certain order and restarting on dead-ends). In a larger grid or more complex maze, this approach might not always find a path efficiently. **How effective is your pathfinding strategy, and what is its limitation?** If the grid were much bigger or had more obstacles, how would the complexity of finding a path grow? Can you think of an alternative algorithm that would ensure finding a path if one exists (name it and describe its advantage)?

- **Q2.** Why do we perform ingredient validation (checking the recipe against the map) before attempting to traverse the house? Discuss what could go wrong if we skipped the `validate_ingredients()` step. In general, **why is input validation important in programming**, especially when dealing with external data like files?

- **Q3.** This program is organized into multiple modules (files) such as separate components for reading input, handling movement, and managing recipe logic. **What are the benefits of this modular design?** Consider aspects like code readability, reusability, and ease of debugging. Provide an example from this lab where separating functionality into a distinct function or module made the implementation easier or clearer.

Please write your answers under clearly labeled sections (e.g., “Q1.”, “Q2.”, “Q3.”) in the answers.txt file. Aim for concise yet complete answers, citing specifics from your implementation or experience completing the lab where appropriate.

Submission Guidelines

By the end of the lab, you should have the following ready to submit:

- **Source Code:** All updated `.c` and `.h` files with your implemented functions (move, collect, validate_ingredients, parsing logic, etc.). Make sure you do **not** change the existing file names or function signatures provided in the skeleton, as the grading scripts may depend on them. Implement your code in the places marked by the TODO comments and follow the style guidelines provided (meaningful variable names, proper indentation, comments where needed).
- **`answers.txt`:** Completed with your answers to Q1, Q2, and Q3. Use clear explanations in your own words.

Before submission, test your program with multiple recipes (both Scenario A and B) to ensure it works for all cases. Verify that the output is correct (ingredients are collected in order, drop-offs happen at the right times for Scenario B, and the final messages make sense). Also, double-check that your answers.txt addresses all parts of each question.

Each group is required to work independently on this laboratory using GitHub for version control. Follow the steps below to clone, implement, and submit your work in a structured and traceable manner.

Instructor Repository

A complete lab skeleton has been provided in the instructor's GitHub repository. It includes:

- Full README.md with instructions
- All source files with clearly marked TODO implementation tasks
- Input files: house_layout.txt, ingredients_map.txt, and recipe files
- A Makefile for compilation
- An answers.txt template for written responses

Student Group Workflow

Each group must follow these steps:

1. **Fork the Instructor Repository**
   - Go to the instructor's GitHub repository.
   - Click the **"Fork"** button to create an independent copy of the lab under your group's GitHub account or organization.
2. **Clone the Forked Repository**

   - Navigate to your group’s forked repository.
   - Click **Code → HTTPS** or **SSH** to copy the repository link.
   - Clone it to your local machine:
   - git clone https://github.com/your-group-name/H.git
   - cd Lab1

3. **Implement the Lab**
   - All code-related tasks are located in the src/ directory.
   - Read and follow the instructions marked as TODO in the source files (e.g., move(), collect()).
   - Implement only the designated sections. Do not rename files or modify the folder structure.
   - Optionally, create a separate working branch for better organization:

   ```
   git checkout -b groupX-implementation
   ```

4. **Commit and Push Your Work**
   - Commit changes incrementally with meaningful messages:

   ```
   git add .
   git commit -m "Implemented move() and tested parsing"
   git push origin main
   ```

5. **Submit Your Final Repository**
   - Once complete, submit the link to your group’s GitHub repository to the course lecturer.
   - Example submission:

     Group 3 Submission: https://github.com/group3username/Lab-1

Guidelines and Policies

- Each group’s fork is independent and does not affect the instructor’s repository or other students’ work.
- Sharing, copying, or reusing code across groups is strictly prohibited. Plagiarism checks will be conducted, and violations will be penalized.
- Ensure that your repository history reflects meaningful contributions and teamwork.

Do not include any extraneous debugging prints (especially large grid dumps) in the final submission – if you used the visual flags for debugging, ensure the default run (without `-v`) prints only the required information.

Finally, **do not rename any provided files or functions**. Keep the project structure as given. When you are satisfied, submit the entire project folder (or as instructed by your TA/instructor). Good luck, and enjoy your cake-baking adventure in C![1]