**PA 1 - Karel Explores his World**

In this programming assignment, you will help Karel move around his world. Even with the few actions Karel can do and the limited queries, we are able to solve some interesting problems. You get to learn to think as a programmer without having to learn a lot of C-syntax first. Nevertheless, as our Karel simulator is built in C, everything you see actually fits directly into the C-framework that you will explore throughout this course.

# Summary

You will find the details of the assignment in [Assignment Details](#_dkxv3sjbe19t). There will be [starter code](#_fqoqmmuy51y1) available for you to work from.

The goal is to write the following Karel programs:

| package.c | Pick up an item at a specific location and return to the starting position. | 10 pt |
| --- | --- | --- |
| wall.c | Move north along a wall until you find an opening and follow it south on the other side. | 20 pt |
| doors.c | Move from the SW to the SE corner of the map through openings in walls. | 20 pt |
| missing.c | Same as doors.c, but the end is not marked with an item. | 20 pt |
| maze.c | Navigate through a maze using the right-hand rule. | 30 pt |
| trail.c | (Extra) Same as maze.c, but leave a trail of items to mark the shortest path. | 20 pt |

To submit the work, please follow the [Submission Instructions](#_gc8zso34mxez).

# 

# Assignment Details

## Getting Started

For this assignment, we are again providing you with some starter code. The process is the same as what you did in PA0.

Go to your home directory and use the following command to get the starter code:

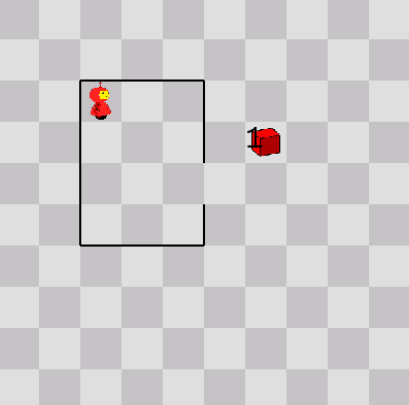
$ getStarter PA1

This will create a subdirectory, called “PA1” with all the starter code. Again, only do this once.

Note that for this programming assignment, you are only allowed to use the actions, queries and programming constructs that were introduced in **Chapter 2. Karel the Robot**. This means that you are NOT allowed to use any variables. If you are yet unfamiliar with variables, don’t worry, we’ll talk about them soon in our course (and you are not allowed to use them here anyway).

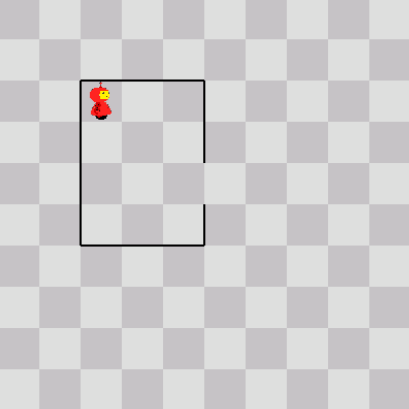
## Karel Retrieves a Package

In this first problem, Karel will start in his little house, exactly as shown below.



**Start position**

A package was delivered to him (in the form of an item in Karel’s world) and is waiting outside (also exactly in the place shown). Your task is to program Karel such that he moves to the item, picks it up and then returns back to the exact tile he started from. So, in the end, everything should look like shown below. Remember, as in all assignments, Karel’s orientation at the end does not matter. When Karel has achieved his goal, he should execute the turn\_off() command. By the way, turn\_off() can be part of an if-statement, be called in a function, etc. You can have it in multiple places in your code. The first one that is encountered is executed and ends the program.



**End position**

Since the environment (size of the map, location of the walls and item) and Karel’s starting position are all fixed, you can “hard-code” your solution. This means that you don’t have to write code that is flexible enough to solve this problem for different variants (e.g., different sizes of his house, different starting positions or locations of the item). Instead, your solution only has to deal with this exact situation.

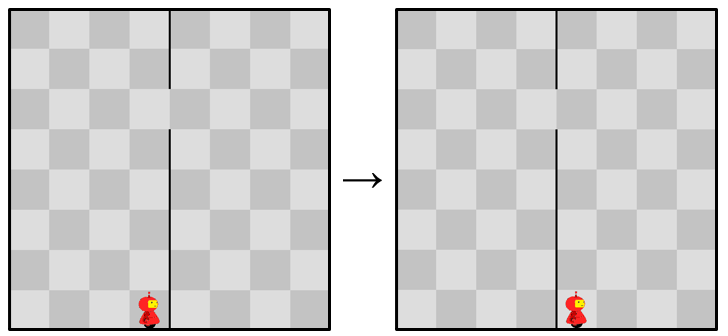
It may still be a good idea to think about using constructs like repeat() and decomposing your problem with functions. It may seem like an unnecessary complication for a problem like this, but it is important to build good habits. Also, when running karel, remember you can speed up the simulation using the arrow keys.

In the starter code, you have the file **package.c**. Write your karel code in this file; it already has all the wrapper code you need. If you need to refresh your memory on how to modify a file, compile a program, etc., just have another look at PA0. Remember, when you run *make*, you do not include the file name extension (i.e., you leave off the ‘.c’ part).

How to submit your code, as well as that of the subsequent parts of this assignment, will be detailed in the [Submission Instructions](#_gc8zso34mxez).

## Karel Follows a Wall

In this problem, Karel finds himself facing east, right in front of a wall, at the south-corner of that wall, as shown below on the left. The wall only has one opening that is exactly one tile wide. However, it is not known how far north the opening is. Karel should keep following the wall north until he finds the opening, go through it and then follow the wall back south on the other end. The goal is to get to the south-corner of the wall on the other side, as shown below on the right.

**Start position End position**

Write your solution in a file called **wall.c**, which is also part of the starter code. Have a look at the code already in that file. It suggests creating a new function “bypass\_the\_wall()”, which is called from main(). If you read the comment, you’ll see that this function actually should perform exactly what we are asking for in this assignment. So basically, you should just implement this function. You should not modify or add any code in the main().

The reason why we are doing it this way is so you get to work with your own functions. But additionally, this function will come in handy for the next problem.

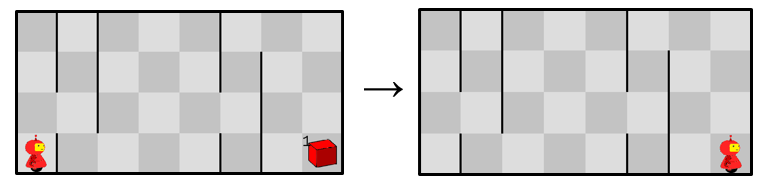
When you are developing your code, keep in mind that you don’t know where the opening is. This means you can no longer hard-code your solution. Karel has to be smarter: he has to figure out when there is an opening in the wall. Your code should work for any position of the opening.

It is good to make sure it works for more than just one example. This means you may want to test out your code for more than one map. This can be done by changing the **settings01\_wall.json** file, which is loaded in the karel\_setup() command and which specifies the map to use. You can find this file in the *settings* subdirectory. Just use your text editor to open the file. The available maps are in the *maps* subdirectory. To load a different map, change the “map” and “goal” attributes in the .json file to the new maps (note: you need to change both, with the “goal” map having the same name as the “map” one, but with “\_end” in the name). For more information on the settings, refer to **Chapter 2. Karel the Robot**. That chapter also explains how you can use our karel map editor to view a map (with the -v option). This is useful when you believe karel accomplished his goal but you are still not getting the “task completed message”. In that case, view the end map to verify what his end state should look like.

That chapter also discusses how you can use our karel map editor to create extra maps for Karel if you want to test out your code more thoroughly. Keep in mind that when we do the grading, we will be using a variety of different new maps to make sure your code works, so try to also test corner cases (e.g., the opening is the first tile, the last tile, etc.). There will always be an opening though.

When debugging your code, also feel free to modify the other settings as you see fit. Keep in mind, however, that we will test the correctness of your code using the settings we provided.

## Karel Searches for the Doors

Now, Karel finds himself in a world as shown below on the left. He starts in the SW corner (facing east) and wants to get to the SE corner and pick up his item there (see the end position below). However, there are a number of walls blocking his way. Fortunately for Karel, each wall has a single one-tile-wide opening. And even more fortunate is that he already knows how to move through an opening for a single wall, as you told him how to do that in the previous problem. Now he just has to do it repeatedly. And remember to pick up the item.

**Start position End position**

As in the previous problem, your solution has to be generic. This means it has to work for different configurations of walls and openings in the walls. The following are always true:

* Karel starts off in the SW corner, facing east, and the item is in the SE corner.
* The walls stretch from N to S and are continuous except for a single one-tile-wide opening in each wall.
* The map has an unknown width.
* The spacing between the walls and the number of walls is unknown.
* Karel should execute turn\_off() when he has reached his objective (picked up the item).

Write your solution in a file called “**doors.c**”, which is also part of the starter code.

Initially solving this problem may seem like a daunting task, but let’s tackle it systematically. You may notice that you could reuse the function “bypass\_the\_wall()”, which you wrote to solve the previous problem. Copy that function from the previous problem and paste it in doors.c. Now, you just need you to get him to bypass the wall repeatedly, plus move on to the next wall, until he reaches his end goal. You may also want to create some extra functions for other subtasks (e.g., for moving to the next wall). Creating functions for different sub-tasks, which then get decomposed in yet more functions and so on, is a paradigm called “divide-and-conquer”. It is one of the bedrocks of good programming and engineering.

Note that Karel does not need to find the shortest path to his item. It is fine if he has to backtrack and such. After all, he doesn’t know where the doors are and has to look for them. Again, when you are testing your solution, make sure it works for more than just one instance of the map. We have provided a few map examples and you can again refer to these in the settings .json file.

Finally, we want to suggest to always put a turn\_off() command as the last line of your code. You can have more than one turn\_off(), by the way, and Karel turns himself off (and the simulation ends), when the first turn\_off() is executed. However, the program will crash non-gracefully if the code is all executed and the end of main() is reached without a turn\_off(). So we suggest to always put a turn\_off() as the last line in main() for all karel assignments, even if you expect never to get to that point (for example, when the turn\_off() you want to execute when everything works properly is part of some if-statement somewhere else in your code).

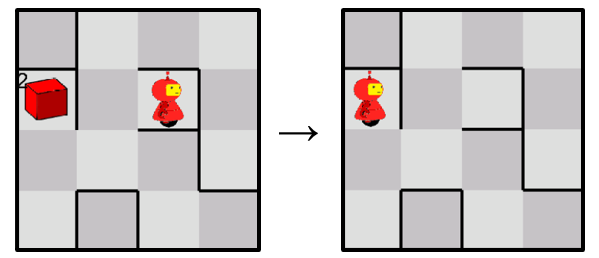
## Karel Finds the Doors but his Item is Missing

For this challenge, we revisit the [Karel Searches for the Doors](#_244ukvr9mmkw) task from before. The basic setup is the same. However, someone has stolen Karel’s item. You need to solve the same problem and Karel should again end up in the SE corner and execute the turn\_off() command at the appropriate time. However, now there is no item there to signal you have reached that specific point (and therefore you also no longer have to pick up the item).

Write your solution in the file **missing.c**, which is part of the starter code.

## Karel Solves a Maze

In this problem, Karel finds himself in an even more challenging environment: a maze. This time there is an extra incentive to make it out though, since at the end there are two packages (items) for him. An example of a maze is shown below, but your solution should again be general enough to work for more than this particular example. However, the maze will not contain any loops. This is known as a maze being “simply connected”.



**Start position End position**

Your goal is to write a program that gets Karel out of such a maze. The following are always true:

* The width of the corridors in the maze is always exactly one tile.
* The maze is simply connected (i.e., it has no loops).
* The exit is marked by two items (and there are no items anywhere else in the maze). The goal is to pick up these two items. As soon as you have picked up the two items, you should execute the turn\_off() command.
* Karel will start at a random spot in the maze, facing a random direction.

A good solution strategy for this kind of maze is known as the “right hand rule” or “wall following”. The easiest way to understand this strategy is by imagining you are stuck in such a maze in absolute darkness, with only your hands to feel your way out (this is analogous to Karel only being able to sense walls in his immediate vicinity, and a common occurrence for many practical robots with short-range distance sensors). In such a scenario, you can imagine touching the right wall with your hand and just start walking always touching the right wall -- hence the name wall following. Because of the [special structure of the maze](https://www.youtube.com/watch?v=IIBwiGrUgzc), you will always reach the exit. Before you start coding, think about the structure of your algorithm first. What should be Karel’s decision for each step he takes?

Your code should go in the file **maze.c**, which is again part of the starter code. As before, your solution should work not just for one specific maze, but for all variants that satisfy the constraints given before. There are a few different maps provided for you.

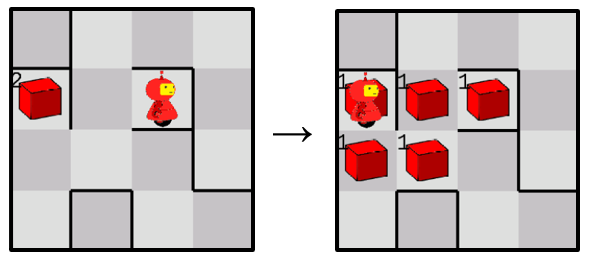
## Extra Challenge: Karel Leaves a Trail of Breadcrumbs

[This part is for extra credit] Extra challenges are optional. For this optional challenge, we go back to [Karel Solves a Maze](#_2hzgvpxff0mp). Your solution should go in the file **trail.c**.

The basic setup is identical to the original maze problem. However, Karel also has to leave a trail of breadcrumbs (i.e., items) for anyone following him. This trail should indicate the *shortest* path from the starting point to the exit, which is not necessarily the path Karel took himself to get there. One way to think about it is that you drag a string behind you (as in the legend of [Theseus and the minotaur](https://www.greekmyths-greekmythology.com/myth-of-theseus-and-minotaur/)), but shorten it each time you retrace your steps, such that in the end what you are left with is the shortest path. Karel has an infinite number of items in his bag.

The end state should be as follows:

* Karel standing on the finish tile (which again was marked by two items at the start of this challenge).
* Exactly one item on each tile that is part of the shortest path, including one item on the start tile and one item on the finish tile. (BTW, this is why we used two items to mark the maze end, to not confuse it with the path you are building).



**Start position End position**

Again, try your solution with different maze examples. This problem is a little bit more challenging. Think of a good strategy first, before you try to implement. Again, use divide-and-conquer to map sub-tasks to functions.

# 

# Submission Instructions

For submitting your assignment, we will again follow the procedure outlined in the **PA Submission, Grading and Regrade Procedures** document. Don’t forget that you need to ssh into the ieng6 servers.

Also, make sure you delete or comment out all your pause() or say\_text() commands. Your code should be able to run without being halted.

You run the submit script in the directory where your .c files are located (this should be PA1 if you followed the directions in this document).

$ submit PA1

This script will submit all the .c files specified in this assignment. So even if you didn’t do the optional assignment, it will still be submitted. That is not a problem; it will just be our own starter code that will be submitted if you didn’t modify it yet.

Remember that running the script again will overwrite your previous submission. So you could run the script as soon as you have some of the problems done, and then run it again and again as you finish more of the assignment.

Remember, you can use the **checkSubmit** command to verify your submission.