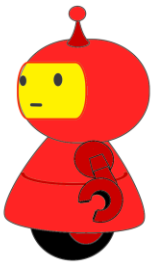
Chapter 2
Karel the Robot

In this chapter, you will get introduced to the fundamentals of programming by working with a little robot, named Karel. Karel lives in a simple world: he can only perform basic actions (such as moving around and picking up items) and receive basic information about his environment (such as the presence of items and walls). Yet, even with this simple vocabulary, you will be able to program some complex and interesting behaviors.

The goal is to get you to start thinking like a programmer before overburdening you with the full complexities of C. Nevertheless, everything you do with Karel is within the C-syntax. The concepts you learn here will therefore directly carry over to the more complex constructs that are covered in later chapters.

2.1 Who is Karel?



In the 1970s, Richard Pattis, then a graduate student at Stanford, created Karel the Robot to teach the fundamentals of programming. The idea was to start with only a simple set of commands that would let students acquire basic programming skills without being bogged down by the complexities of a vast syntax. In its first incarnation, published in 1981, the Karel language was its own standalone programming language, similar to PASCAL. Due to the success of Karel, alternate versions and variants have been developed over the years by various people around the world, some porting Karel to a syntax closer to C or Java or morphing his environment to an island or more exotic settings. The main power of Karel has remained the same, however: teach students how to **think** about programming before worrying about how to write complex code. Karel is still a mainstay in programming courses in universities all around world.

By the way, the name “Karel” is in recognition of the Czech writer Karel Čapek, who wrote the play R.U.R, which coined the word “robot”.

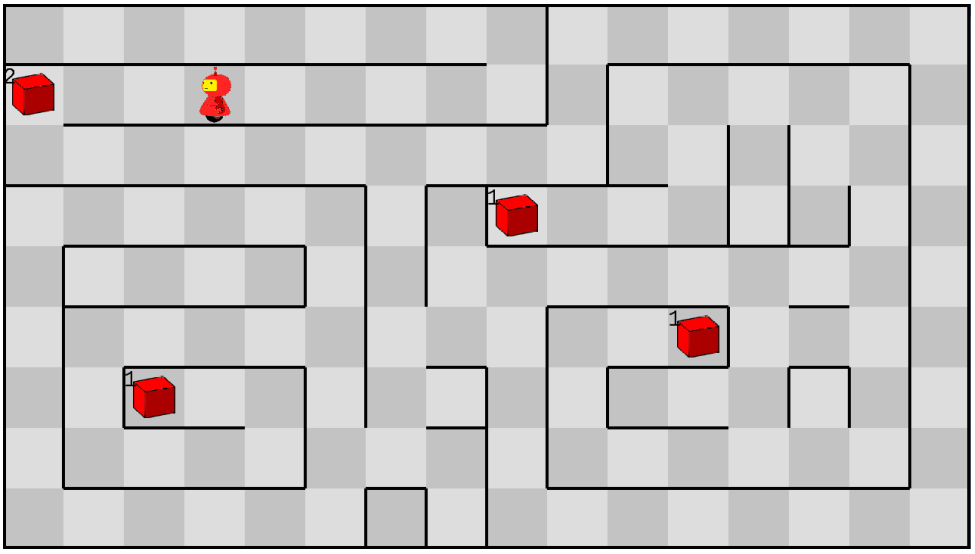
2.2 Karel Overview

2.2.1 Karel's world

In our context, **Karel speaks C**. He lives in a fairly simple rectangular grid world. We will talk about a "map" with "tiles", and coordinates given as rows and columns starting with the bottom left corner being the first row and first column. There are walls that Karel cannot pass through, indicated by solid lines. Note that his world is surrounded by walls along the edges.

There are also "items" (displayed as red cubes). You can think of items as generic things that are relevant to the scenario that he finds himself in: they could represent bricks, packages, or treasure he needs to collect. Karel can walk past items; i.e., they don't block his path. He can also decide to pick them up or deposit them where he wants. For this purpose, he carries a bag with him, which may or may not already have items in it when he starts his adventure.

Even in such a simple world, Karel has responsibilities – he may need to reach a specific part of the map, move items around, or both. Your task will be to program Karel to accomplish his goals in various situations.



Karel in his world

2.2.2 Karel's Skills

When you program Karel, it is useful to know what exactly he can do. The things we will talk about here are specific to Karel and his world -- what actions he can take and how he can perceive his surroundings.

First, let's talk about the **actions** Karel can take, i.e., what he can do. They are listed below:

`move()`	Karel moves one tile forward in the direction he is facing. If he is in front of a wall, he crashes into it and this will end the program.
`turn_left()`	Karel turns in-place counter-clockwise by 90 degrees. He remains on the same tile.
`turn_right()`	Similar to turn_left, but now clockwise by 90 degrees.
`take_item()`	Karel picks up one item from the tile he is standing on. If there is no item there, it will end the program.
`put_item()`	Karel takes one item from his bag and places it on the tile he is standing on. If his bag is empty, it will end the program.
`turn_off()`	Karel turns himself off and checks if he accomplished his end-goal.

Next, let's look at specific **queries** Karel can execute to learn about what is going on in his surroundings. You can think of each of these resulting in a "yes" or "no" answer.

`wall_in_front()`	Is there a wall between Karel and the next tile (in the direction Karel is facing)?
`wall_to_left()`	Similar to wall_in_front(), but now looking to the left (so 90 degrees counterclockwise from direction Karel is facing).
`wall_to_right()`	Similar to wall_to_left(), but now looking to the right.
`item_present()`	Are there one or more items on the tile Karel is standing on?
`bag_empty()`	Is Karel's bag empty (i.e., are there no items in his bag)?
`facing_north()`	Is Karel facing north?
`facing_east()`	Is Karel facing east?
`facing_south()`	Is Karel facing south?
`facing_west()`	Is Karel facing west?

2.3 Writing a Basic Karel Program

2.3.1 Program Structure

A Karel program is essentially a sequence of actions you want him to take, possibly depending on the result of queries. When we are programming Karel, we are doing this within the framework of the C-language. For example, you may have noticed that all the actions and queries ended with "()", which is a result of our C-implementation (this will become clear soon).

However, when we create a complete Karel program, we cannot just write a list of commands. There is a bit of wrapper code that is needed to properly set up the simulation. The structure of this wrapper code is shown in the figure below. Everything you see there is the code framework we need for our C-based karel simulation. Don't worry. At this point, it is enough to just accept that this stuff needs to be there. We will provide you with this wrapper-code. It is not something you need to change or write yourself.

Within this wrapper, you will need to insert your own code to control Karel. In the figure below, **<…> indicates the spot where your code will go.** On the next page is an example of a simple karel program embedded in this wrapper code.

```
#include <karel.h>
int main() {
    karel_setup("settings/settings00.json");
    <...>
}
```

Even though you don't have to modify this wrapper code, let's nevertheless have a quick look, as it illustrates some C-concepts that we will cover later anyway. The first line, starting with `#include`, tells the program to add in functionality that is defined elsewhere (so-called libraries), so we don't have to do everything from scratch. In this case, we are including the `karel.h` library, which is something we build to run the karel simulator.

Next, you find the basis of all C-programs: the main-function. The main function starts with the line `int main()`. This is where execution starts. The { } delineate the extent of this function -- everything between the brackets is executed. The code in this main-function may call on other pieces of code located elsewhere, but we will talk about that later. Right now, the important part is that a C-program always has a main-function and that execution starts there.

The `karel_setup()` function creates our simulation environment: it builds a map and puts Karel in it. All the necessary information is listed in the file specified in parentheses (with its relative path). We will talk a little more about these settings in section 2.5.3 Karel Settings File.

When we are programming Karel, we need to save the code in a file to be compiled. This file will have a **.c** extension as it is essentially C-code; for example, karel_test.c (or whatever name you decide to give your file).

2.3.2 Doing Actions

Now, let's look at a simple Karel program with him performing some actions.

```
#include <karel.h>

int main() {
    karel_setup("settings/settings00.json");

    move();
    turn_right();
    move();
}
```

In this example, provided there are no walls in the way, Karel moves one tile forward, turns 90 degrees to the right and moves again one tile forward. That's it.

At this point, it is important to note that actions need to be followed by a "**;**". The reason is that they are "statements" in C (we will formally define statements in a later chapter), which need to be terminated with a semi-colon. Forgetting the semi-colon will result in compilation errors (compilation is the process of preparing our code for execution). It is a mistake every C programmer makes at one point or another.

2.3.3 Using Queries for Conditional Behavior

When defining Karel's behavior, you can also bring in the queries we introduced before, to make him react to what is happening around him. For example:

```
#include <karel.h>

int main() {
    karel_setup("settings/settings00.json");

    move();
    if ( item_present() ) {
        take_item();
        turn_left();
    }
}
```

Note that the query is not followed by a "**;**". It is used here as part of an if-statement, which is another C-construct. If the query is answered with a "yes", the action or list of actions that follow between { } are executed.

```
if (query) {
    action;
    ...
}
```

The if-statement allows you to perform conditional execution. Basically, do an action only if certain conditions are true. An extension is the if-else-statement, where you do one action or set of actions if the query is answered with a “yes” and another set otherwise (the “else”-part).

```
if (query) {
    action;
    ...
}
else {
    alternativeaction;
    ...
}
```

Note that the { } are strictly speaking not necessary if only a single action follows. You can think of the { } as grouping the actions together. An if-statement is essentially the if-part followed by a “block of code”, where a **block of code is either a single action (statement) or a group of actions (statements) between { }**. Something similar will hold true when we talk about loops in upcoming sections.

Also, thus far, we have assumed you want to do actions when a query is answered with “yes”. But what if we want to do something more complex like the action being dependent on two different queries, like doing something when there is an item AND we are facing east? One way to do this is to extend what we already know. The if-statement (i.e., the if( ) and the associated part between { } together) also behaves as (a more complex) action in itself. Whenever we have talked about there being an action done as part of an if-statement, this action itself could be a more complex action, like another if-statement. This is called “nesting”.

For example:

```
#include <karel.h>

int main() {
    karel_setup("settings/settings00.json");
    move();
    if ( item_present() ) {
        if ( facing_east() ) {
            take_item();
            turn_left();
        }
    }
}
```

However, an alternative solution is to perform logical combinations of queries, basically literally writing the “and” when we want two things to both be true at the same time. Well, almost literally; C uses “**&&**” instead of the word “and”.

```
#include <karel.h>

int main() {
    karel_setup("settings/settings00.json");
    move();
    if ( item_present() && facing_east() ) {
        take_item();
        turn_left();
    }
}
```

Other useful keywords to perform logical operations are:

||	“or”	For example:	`if (facing_north() || facing_south())`
!	“not”	For example:	`if ( !wall_in_front() )`

You can also combine these to make it even more complex. Just like with addition and multiplication, one type of operation is always executed before the other (we say “one has precedence”). However, it is easiest to simply use parentheses:

```
if ( !facing_north() || (facing_south() && !wall_in_front()) )
```

2.3.4 Loops

Another thing we would like to be able to do is to execute certain actions repeatedly. The while-loop is reminiscent of the if-statement, but now the actions between { } are repeatedly executed *as long as* the query results in a “yes”.

```
while (query) {
    action;
    ...
}
```

What happens is that when the query is answered with “yes”, the actions are executed one-by-one. Then, we jump back to the query and see if it results in “yes” again. If so, we execute the actions again and jump back to the query, and so on. This process continues until the query is answered with “no”; we then skip the { } block and continue with the code that follows after.

The “repeat”-loop also results in the actions between { } being repeated, but now a fixed number of times, denoted by **n**.[1] In your actual code, you need to replace **n** by the number of times (a whole positive number) you want the actions to be executed.

```
repeat(n) {
    action;
    ...
}
```

For example:

```
#include <karel.h>

int main() {
    karel_setup("settings/settings00.json");
    repeat(3) {
        move();
        turn_right();
    }
    while ( item_present() && !facing_east() ) {
        take_item();
        turn_left();
    }
}
```

As before, the repeat-loop and the while-loop can be considered as more complex actions. This means you could have an if-statement inside a while-loop, which is inside a repeat-loop, etc.

2.3.5 Turning Karel Off

There is one karel command that needs a little more explanation: `turn_off`(). This will turn Karel off and end the simulation. You should call it when Karel has finished his task. The command also checks if he indeed reached his desired goal, and print out whether or not he did. This end-goal is specified by an end-map, which depicts what Karel’s world should look like after he accomplished his task. Just like the world he starts in, the end-map is also specified in the settings file, as will be detailed in section 2.5.3.

The `turn_off`() command does not need to be the last line of code. For example, it could also be in an if-statement or anywhere else in your program. Basically, whenever you think Karel has reached his objective, you should execute `turn_off`(). Also, you can have more than one `turn_off`() command in your code. Karel turns himself off (and the simulation ends), when the first `turn_off`() is executed.

[1] Note that repeat(n) is not part of the C-standard. It is something we created for you in karel to make your code easier. In the next chapter, you will see how you can achieve this functionality using a for-loop.

However, the program will crash non-gracefully if all the code is executed and the end of `main`() is reached without a `turn_off`(). For safety, we suggest to always put `turn_off`() as the last line in `main`(), even if you expect never to get to that point; for example, when the `turn_off`() you want to execute when things work properly is part of an if-statement somewhere else in your code, as illustrated below.

```
#include <karel.h>

int main() {
    karel_setup("settings/settings00.json");
    move();
    turn_right();
    if ( item_present() )
        turn_off();
    move();
    turn_off();
}
```

2.4 Organizing Your Code

2.4.1 Defining your own Actions with Functions

You may have noticed that Karel's actions and queries all end with double parentheses ( ). This is in fact part of the C-syntax, which our version of karel is based on. Under the hood, our actions and queries are all implemented as what are called **functions** in C. Specifically, they are functions that we have created for you (as part of our karel library) to result in interactions within Karel's world.

You also have the power to create functions yourself. In fact, it is good practice to do so and to decompose your program in bite-sized little actions that you define yourself, to make reasoning about a task easier. For example, you can imagine it would be useful if Karel had an action that had him turn until he was facing north. Such a functionality could have maybe been called `turn_until_north`(). Well, at present, he doesn't know this command. But you could teach him this new action. You just have to define it for him in terms of things he already knows. This boils down to defining your own function.

In the example on the next page, we first define our new function, which specifies what should happen when the function is executed. This is called the function definition. In the `main`(), our new function is then called, much like we called pre-defined functions, such as `move`(). A function call results in the execution of that function. This means that the code in the function

definition, which starts with "`void turn_until_north()`", is not executed immediately. It is only executed when the function is called in `main`(), with the line `turn_until_north();`.

Let's have to closer look at what actually happens. Remember, overall code execution starts with `main`(), so you just start reading from the first line in `main`(). This means that, first, Karel moves one tile; he already knows intrinsically how to do this. Then there is this `turn_until_north()`. This is not something Karel knows by himself. However, luckily, we have defined what it means already, so Karel now knows how to do `turn_until_north`(): it is by executing the code given in the function definition. After that code is done, Karel picks back up where he left off in `main`(), in this case he does another move.

Note that in C, the program always needs to know the details of how a function is called before you can actually call it. Based on what we have learned about functions thus far, this means that inside the **.**c file, the function definition should come before you call that function. In later chapter, we will discuss other options as well. However, for now, make sure you define a function before you call it.

```c
#include <karel.h>

/* Function definition: Our own function to tell Karel to
turn until he is facing north */
void turn_until_north() {
    while ( !facing_north() ) {
        turn_left();
    }
}

int main() {
    karel_setup("settings/settings00.json");
    move();
    turn_until_north();     // Function call:
                            // Here I'm using our new function
    move();
}
```

By the way, in the example above, we showed two ways of writing comments (we marked them in red). Comments help you make sense of your program later and are useful if someone else looks at your code. Karel (and C in general) does nothing with them; he just happily ignores them. One way to write comments is by using "//"; in that case, everything that follows on the same line becomes a comment. Another way is using "/*" and "*/"; in this case, everything between "/*" and "*/" is a comment (even if they are on different lines). The latter approach allows you to have comments span multiple lines.

2.4.2 Coding Style

We also wanted to say a few words about coding style. In previous section, we briefly introduced comments, using // or /* */, to indicate code that is meant for humans to look at and that is ignored by the computer.

Comments are part of good coding practice. As you can imagine, writing “good” code is extremely important, for your own sake as a programmer and for that of the people you will work with on coding projects. Good code does not mean the most compact code. Instead, the two quotes below speak for themselves.

> Programs must be written for people to read, and only incidentally for machines to execute.
>
> *- H. Abelson & G. Sussman ("The Structure and Interpretation of Computer Programs”)*

> Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" Improve the code and then document it to make it even clearer.
>
> *- S. McConnell*

Comments are important to help people understand your code, both when you look at it yourself later or someone else does. Another aspect of this, which we have glossed over thus far, is the use of spaces and indentation as we’ve illustrated in your example programs. The C-compiler is actually completely oblivious to these. They were included only for human-readability.

What is even more important than having code that looks pleasant is to structure it properly. The concept of functions plays a critical role here. They allow you to strategically decompose your program into simpler functionalities. This strategy of **divide-and-conquer** is the key to coding success. Writing well-structured and easy to read code is what makes a good programmer. In this regard, it is important to develop good habits early on. So even when Karel programs may appear easy, try to follow these good coding practices and think about your approach before you start writing code.

2.5 Running Karel Programs

2.5.1 Compiling and Running the Code

Your karel code is written in a file with a **.**c extension. This is the standard extension of files containing C-code.

Once you have finished your karel code, you need to compile it. The easiest way to do this is with a Makefile, which automates the compilation process (for more on Makefiles, see also Appendix B. Compiling Code). To compile, for example karel_test.c, the only thing you need to do is type the following command:

```
$ make karel_test
```

Note that you omit the **.**c extension when you run the "make" on your karel C-file.

The compilation process will result in an executable file, which has the same name as your **.**c file, but without the **.**c extension. In our example, the executable that would be generated is karel_test.

This executable is the actual file you need to run to execute the program. To run it, i.e., to execute the program, type:

```
$ ./karel_test
```

Note that the "**./**" is added to tell Linux to look for the executable in your current directory.

When you execute the program, it prints out some information in the terminal window. Also, it may show you a simulation of Karel moving through his world. Whether or not you want this visual output is a simulation setting you can control. This is done in the settings file, as will be explained in the next section. In short, there are three options: showing full visuals (i.e., a separate graphical window), showing simplified visuals (i.e., ASCII visuals in the terminal window), or not showing any visual output at all.

2.5.2 Controlling the Simulation

There are some additional "actions" you can use for your karel simulation. They are not actually things Karel does, but rather they help you control the simulation and make it is easier to keep track of out what is going on. They are integrated into your code just as regular karel actions.

`pause()`	Karel pauses until you to press a random key (the karel window must be the active window).
`say_text("Hi")`	Karel prints the text between double quotes (the text, `Hi`, is only shown here as an example). Execution also pauses as with the `pause`() command.

`hide()`	Graphics are no longer shown from this point onward. This is useful to speed up the simulation.
`show()`	Graphics are shown again from this point onward. This essentially undoes the `hide()` command.

2.5.3 Karel Settings File

In this section, we will elaborate on the `karel_setup()` command. As mentioned before, it imports the settings to build Karel's environment. In quotes is the name of the settings file (of type **.**json), together with its path (in the example on the next page, we use the settings file settings00.json located in subdirectory settings/). You can modify the settings file to change some of the karel simulation parameters. Alternatively, you can also create a new settings file and then refer to that one in `karel_setup()`.

```
#include <karel.h>

int main() {

    karel_setup("settings/settings00.json");

    move();
    turn_right();
    move();

    turn_off();
}
```

Let's look at the contents of an example settings file. It consists of a list of entries that specify various simulation parameters. The only required entry is the "map", specifying the starting environment. All the others are optional. If they are omitted, their default values are used. If the end goal map is not given, there will be no end goal check.

```
{
   "map": "maps/map00.mf",
   "goal": "maps/map00_end.mf",
   "max_actions": 26,
   "auto_terminate": 0,
   "enable_visuals": 1,
   "items_in_bag_at_start": 0,
   "karel_end_matters": 1,
   "color_scheme": 1
}
```

- "**map**" and "**goal**" specify map files (again, with their path). These map files (extension **.**mf) contain all the information about Karel's environment. The "map" encodes the starting environment; i.e., where all the walls and items are, and where Karel starts. The "goal" encodes a similar map, but this time depicting what everything should look like when the task is completed. This is used in the `turnoff`() command to check if Karel accomplished his goal. If you want to create your own maps for Karel, you can do so with a separate map editor program, as explained in section 2.6. The goal map can be omitted from the settings, but in that case no check is performed whether the end goal was reached.
- "**max_actions**" specifies the maximum number of actions Karel can take to reach his objective. A value of -1 signifies an infinite value. This setting is included to avoid getting stuck in an infinite loop somewhere or to eliminate solutions that are too inefficient. The default (i.e., the value used when this entry is omitted) is -1.
- "**auto_terminate**" encodes whether Karel figures out automatically if he has reached his end objective (1) or if you need to include this explicitly in your code by using turn_off() at the appropriate time (0). Typically, this will be set to 0 and you will need to use turn_off(). The default is 0.
- "**enable_visuals**" allows you to control whether to show the visual output of Karel moving through his world, and how this is done. This is explained in detail in the next section. The default is 1.
- "**items_in_bag_at_start**" specifies how many items Karel has in his bag at the beginning of the simulation. A value of -1 signifies an infinite number of items. The default is -1.
- "**karel_end_matters**" sets whether at the end, when checking if he accomplished his task, Karel's end position and/or orientation matter. The options are: (0) neither matter; (1) only his position matters, but not his orientation; (2) both his position and orientation matter. The default is 1.
- "**color_scheme**" specifies the color scheme to use when displaying the maps. The options are 0 (plain gray), 1 (light checkerboard) or 2 (dark checkerboard). The default is 1.

2.5.4 Enabling Visual Output or Not

The use of visual output is controlled by setting the "enable_visuals" entry in the settings file. The following options are supported.

Full Graphics

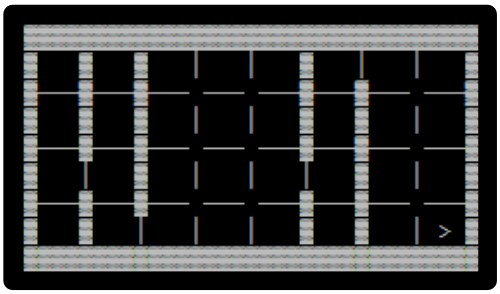
"enable_visuals" is set to an integer value greater than 0. In this case, a new window will pop up with a simulation of Karel (this will look like the Karel examples we've shown throughout this chapter). Selecting higher values of "enable_visuals" (between 1 and 20) results in faster simulation speeds. You can also speed up the simulation or slow it down by pressing the right and left arrow keys.

To close the simulation window after Karel is done, press the ‘x’ on the top right of the karel simulation window, or go to your terminal (where you typed in the command to run the executable) and press control-c. There you will also see a message indicating whether Karel reached his objective or not. If you need to interrupt your program midway through, you can click the ‘x’ on the top right of the karel window.

Note that for this visual simulation to work, the simulator needs to be able to open a new window. This has implications if you want to work remotely. If you use plain ssh, opening a new window is not supported. In this case, simulator will automatically default back to the simplified graphics which will be explained next. Possible solutions are either using X-forwarding with ssh or a vnc.

Simplified Graphics

“enable_visuals” is set to an integer value less than 0. This option runs karel with a simplified output (using ASCII graphics) in the terminal, as shown below. The benefit is that it does not require any external windows, and therefore works when logging in remotely in way that does not support new windows (e.g., when using simple ssh). It is a good way to at least have some feedback on what is happening when running karel remotely, without the overhead of full visual interactivity. As before, values of “enable_visuals” that are greater in absolute value (between -1 and -20) result in faster simulation speeds. You can interrupt a simulation by going into the terminal window and pressing control-c.



Simplified ASCII graphics

Side Note: It is also possible to compile your karel program with the MODE flag set to ‘a’ (for ascii). For example:

```
$ make karel_test  MODE=a
```

This creates an executable as before. However, if visuals are enabled, they are always the simplified ones. The advantage of compiling your program this way is that it uses a simplified karel library that works on systems where the advanced imaging library isn’t even installed.

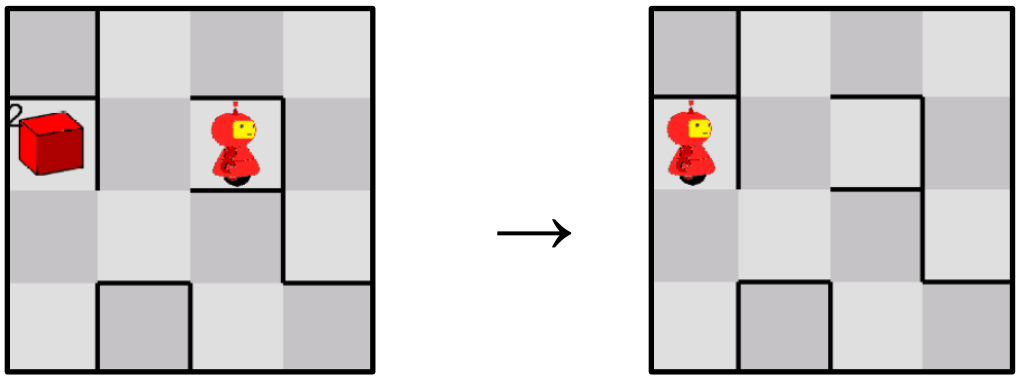
No Graphics

"enable_visuals" is set to 0. If you do not want to work with visual outputs at all, you can disable them altogether. In this case, the simulation runs in the background and you only get a report whether Karel succeeded in his task or not (assuming the code finished because Karel executed the `turn_off`() command). As before, pressing control-c interrupts the simulation.

2.6 Creating Maps for Karel

2.6.1 Map Editor

Map files contain all the information the simulator needs to create Karel's environment. As discussed earlier, there are two such maps: one representing the start state (encoded as "map" in the settings file) and one representing the end state (encoded as "goal" in the settings file). These maps show all the walls, the items and Karel's position. For example:



Start state **End state**

You can view existing maps or create your own maps with our karel map editor. To run the map editor, it needs to be able to open a new window (i.e., you cannot login remotely using simple ssh).

To view an existing map with name <map_name> (e.g., one that we created for you), you can use the following command:

```
$ kareledit -m <map_name>
```

This will open the map in a separate window. Since you are in view-only mode, you cannot make any changes to this map. To exit, simply close the window.

With the same editor, you can also create maps yourself. To create a blank map of size <x_dim> by <y_dim> with name <map_name> use the command (dimensions are constrained to numbers between 1 and 99):

$ kareledit <x_dim> <y_dim> <map_name>

This will open the new blank map in a separate window. You can then add different things to this map using the map editor commands (see 2.6.2). Make sure you use the save command (press q) to actually store the changes to the map file.

For example, this command opens the map editor with a blank map of size 10 tiles by 5 tiles, which upon saving will create map_test.mf (we use extension **.**mf for karel map files):

$ kareledit 10 5 map_test.mf

You can also modify an existing map. If you save it, it will overwrite the earlier version.

$ kareledit -m <map_name>

Finally, there is an option to edit an existing map and save it under a different name. This is useful to create an end map for an existing start map.

$ kareledit -e <old_map_name> <new_map_name>

2.6.2 Map Editor Commands

When you have a map open in the map editor (either a new one or one you are editing), you use your mouse to hover over the tiles (make sure your window is active; you may have to click on it). The commands listed below act on the current tile and allow you to modify the map. Note that you must press ‘q’ to save the map. You can then close the window. If you close without pressing ‘q’, your changes will not be saved.

w	Add or delete a wall on the north side of the current tile
a	Add or delete a wall on the west side of the current tile
s	Add or delete a wall on the south side of the current tile
d	Add or delete a wall on the east side of the current tile
k	Place Karel on the current tile
r	Rotate Karel
Left mouse click	Add an item on the current tile
Right mouse click	Delete an item from the current tile
q	Save the current map