2

Personal Robots

Every Pleo is autonomous. Yes, each one begins life as a newly-hatched baby Camarasaurus, but that's where predictability ends and individuality begins. Like any creature, Pleo feels hunger and fatigue - offset by powerful urges to explore and be nurtured. He'll graze, nap and toddle about on his own -when he feels like it! Pleo dinosaur can change his mind and his mood, just as you do.

From: www.pleoworld.com

Opposite page: Pleo robots
Photo courtesy of UGOBE Inc.

Most people associate the personal computer (aka the PC) revolution with the 1980's but the idea of a personal computer has been around almost as long as computers themselves. Today, on most college campuses, there are more personal computers than people. The goal of One Laptop Per Child (OLPC) Project is to "provide children around the world with new opportunities to explore, experiment, and express themselves" (see www.laptop.org). Personal robots, similarly, were conceived several decades ago. However, the personal robot 'revolution' is still in its infancy. The picture on the previous page shows the Pleo robots that are designed to emulate behaviors of an infant Camarasaurus. The Pleos are marketed mainly as toys or as mechatronic "pets". Robots these days are being used in a variety of situations to perform a diverse range of tasks: like mowing a lawn; vacuuming or scrubbing a floor; entertainment; as companions for elders; etc. The range of applications for robots today is limited only by our imagination! As an example, scientists in Japan have developed a baby seal robot (shown on the opposite page) that is being used for therapeutic purposes for nursing home patients.

Your Scribbler robot is your personal robot. In this case it is being used as an educational robot to learn about robots and computing. As you have already seen, your Scribbler is a rover, a robot that moves around. Such robots have become more prevalent in the last few years and represent a new dimension of robot applications. Roaming robots have been used for mail delivery in large offices and as vacuum cleaners in homes. Robots vary in the ways in which they move about: they can roll about like small vehicles (like the lawn mower, Roomba, Scribbler, etc.), or even ambulate on two, three, or more legs (e.g. Pleo). The Scribbler robot moves on three wheels, two of which are powered. In this chapter, we will get to know the Scribbler in some more detail and also learn about how to use its commands to control its behavior.

The Scribbler Robot: Movements

In the last chapter you were able to use the Scribbler robot through Myro to carry out simple movements. You were able to start the Myro software, connect to the robot, and then were able to make it beep, give it a name, and move it around using a joystick. By inserting a pen in the pen port, the

scribbler is able to trace its path of movements on a piece of paper placed on the ground. It would be a good idea to review all of these tasks to refresh your memory before proceeding to look at some more details about controlling the Scribbler.

The Paro Baby Seal Robot. Photo courtesy of National Institute of Advanced Industrial Science and Technology, Japan (paro.jp).

If you hold the Scribbler in your hand and take a look at it, you will notice that it has three wheels. Two of its wheels (the big ones on either side) are powered by motors. Go ahead turn the wheels and you will feel the resistance of the motors. The third wheel (in the back) is a free wheel that is there for support only. All the movements the Scribbler performs are controlled through the two motor-driven wheels. In Myro, there are several commands to control the movements of the robot. The command that directly controls the two motors is the `motors` command:

```
robot.motors(LEFT, RIGHT);
```

In the command above, `LEFT` and `RIGHT` can be any value in the range `[-1.0...1.0]` and these values control the left and right motors, respectively. Specifying a negative value moves the motors/wheels backwards and positive values move it forward. Thus, the command:

```
robot.motors(1.0, 1.0);
```

will cause the robot to move forward at full speed, and the command:

```
robot.motors(0.0, 1.0);
```

will cause the left motor to stop and the right motor to move forward at full speed resulting in the robot turning left. Thus by giving a combination of left and right motor values, you can control the robot's movements. Myro has also

provided a set of often used movement commands that are easier to remember and use. Some of them are listed below:

```
forward(SPEED)
backward(SPEED)
turnLeft(SPEED)
turnRight(SPEED)
stop()
```

Another version of these commands takes a second argument, an amount of time in seconds:

```
forward(SPEED, SECONDS)
backward(SPEED, SECONDS)
turnLeft(SPEED, SECONDS)
turnRight(SPEED, SECONDS)
```

Providing a number for `SECONDS` in the commands above specifies how long that command will be carried out. For example, if you wanted to make your robot traverse a square path, you could issue the following sequence of commands:

```
robot.forward(1, 1);
robot.turnLeft(1, .3);
robot.forward(1, 1);
robot.turnLeft(1, .3);
robot.forward(1, 1);
robot.turnLeft(1, .3);
robot.forward(1, 1);
robot.turnLeft(1, .3);
```

of course, whether you get a square or not will depend on how much the robot turns in 0.3 seconds. There is no direct way to ask the robot to turn exactly 90 degrees, or to move a certain specified distance (say, 2 ½ feet). We will return to this later.

You can also use the following movement commands to translate (i.e. move forward or backward), or rotate (turn right or left):

```
translate(SPEED)
rotate(SPEED)
```

Additionally, you can specify, in a single command, the amount of translation and rotation you wish use:

```
move(TRANSLATE_SPEED, ROTATE_SPEED)
```

In all of these commands, `SPEED` can be a value between `[-1.0...1.0]`.

You can probably tell from the above list that there are a number of redundant commands (i.e. several commands can be specified to result in the same movement). This is by design. You can pick and choose the set of movement commands that appear most convenient to you. It would be a good idea at this point to try out these commands on your robot.

Do This: Edit your driver program to connect to the robot and to try out the following movement commands on your Scribbler:

First make sure you have sufficient room in front of the robot (place it on the floor with a few feet of open space in front of it).

```
robot.motors(1, 1);
wait(5);
robot.motors(0, 0);
```

(The `wait` command waits for a specified number of seconds.) Observe the behavior of the robot. Specifically, notice if it does (or doesn't) move in a straight line after issuing the first command. You can make the robot carry out the same behavior by issuing the following commands:

```
robot.move(1.0, 0.0);
wait(5);
robot.stop();
```

Go ahead and try these. The behavior should be exactly the same. Next, try making the robot go backwards using any of the following commands:

```
robot.motors(-1, -1);
robot.move(-1, 0);
robot.backward(1);
```

Again, notice the behavior closely. In rovers, precise movement, like moving in a straight line, is difficult to achieve. This is because two independent motors control the robot's movements. In order to move the robot forward or backward in a straight line, the two motors would have to issue the exact same amount of power to both wheels. While this technically feasible, there are several other factors than can contribute to a mismatch of wheel rotation. For example, slight differences in the mounting of the wheels, different resistance from the floor on either side, etc. This is not necessarily a bad or undesirable thing in these kinds of robots.
Under similar circumstances even people are unable to move in a precise straight line. To illustrate this point, you can try the experiment shown on right.

For most people, the above experiment will result in a variable movement. Unless you really concentrate hard on walking in a straight line, you are most likely to display similar variability as your Scribbler. Walking in a straight line requires constant feedback and adjustment, something humans are quite adept at doing. This is hard for robots to do. Luckily, roving does not require such precise moments anyway.

Do humans walk straight?

Find a long empty hallway and make sure you have a friend with you to help with this. Stand in the center of the hallway and mark your spot. Looking straight ahead, walk about 10-15 paces without looking at the floor. Stop, mark your spot and see if you walked in a straight line.

Next, go back to the original starting spot and do the same exercise with your eyes closed. Make sure your friend is there to warn you in case you are about to run into an object or a wall. Again, note your spot and see if you walked in a straight line.

Do This: Review all of the other movement commands listed above and try them out on your Scribbler. Again, note the behavior of the robot from each of these commands.

Defining New Commands

Trying out simple commands by modifying `Driver.cpp` is a nice way to get to know your robot's basic features. We will continue to use this each time we want to try out something new. However, making a robot carry out more complex behaviors requires several series of commands. Having to type these over and over in a control program can get tedious. C++ provides a convenient way to package a series of commands into a brand new command called a *function*. For example, if we wanted the Scribbler to move forward and then move backward (like a yoyo), we can define a new command (function) called `yoyo` as follows:

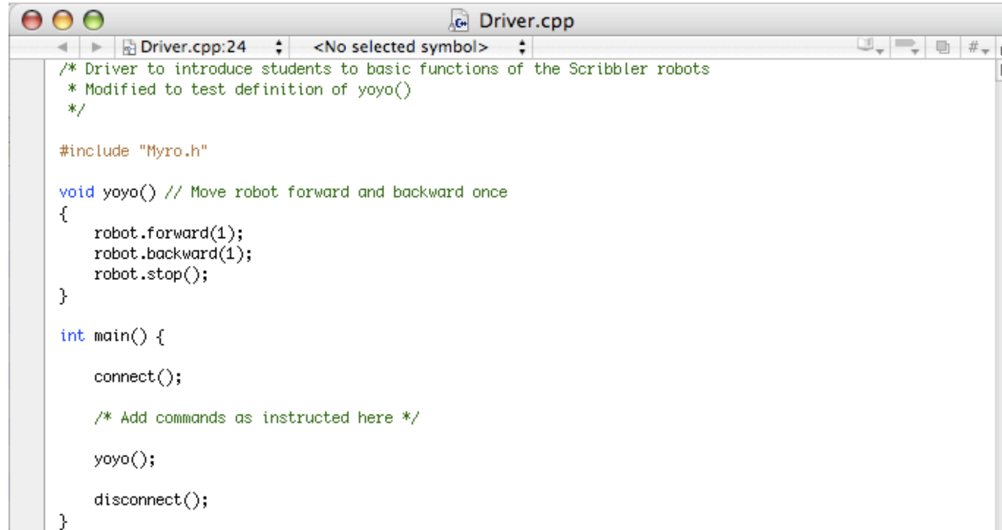
```
void yoyo() // Move robot forward and backward once
{
    robot.forward(1);
    robot.backward(1);
    robot.stop();
}
```

The first line defines the *name* of the new command/function to be `yoyo`. (We will explain later the exact significance of the word `void`, which indicates that we are defining a command.) The left and right curly braces (`{}`) mark the beginning and end of the function's *body*. The lines in between are slightly indented and contain the commands that make up the `yoyo` behavior. That is, to act like a yoyo, move forward and then backward and then stop. The indentation is important and is used in all programming languages to improve readability. We will have more to say about this later.

Put the definition of `yoyo` into `Driver.cpp` before the definition of `main`. After the new command has been defined, you can try it by entering the command into the usual place in `main` as shown below:

```
yoyo();
```

After you have done this editing, your `Driver.cpp` file should look like the following figure:



Do This: If you have your Scribbler ready, go ahead and try out the new definition above by modifying your driver to include the definition and command `yoyo()` as shown above.

Observe the robot's behavior when you run the control program. You may need to run the program several times. The robot momentarily moves and then stops. If you look closely, you will notice that it does move forward and backwards.

In C++, you can define new commands (functions) by using the syntax as shown above. Note also that defining a new command doesn't mean that the commands inside it get carried out. You have to explicitly issue the new command to do this. This is useful because it gives you the ability to use the command over and over again (for example, writing `yoyo()` more than once). Issuing the new function like this in C++ is called, *invocation*. Upon invocation, all the commands that make up the function's definition are executed in the sequence in which they are listed in the definition.

How can we make the robot's `yoyo` behavior more pronounced? That is, make it move forward for, say 1 second, and then backwards for 1 second, and then stop? You can use the `SECONDS` option in forward and backward movement commands as shown below:

```
void yoyo() /* Move robot
forward and backward for 1
sec. */
{
    robot.forward(1,1);
    robot.backward(1,1);
}
```

Scribbler Tip:

Remember that your Scribbler runs on batteries and with time they will get drained. When the batteries start to run low, the Scribbler may exhibit erratic movements. Eventually it stops responding. When the batteries start to run low, the Scribbler's red LED light starts to blink. This is your signal to replace the batteries.

The same behavior can also be accomplished by using the command `wait`, which is used as shown below:

```
wait(SECONDS);
```

where `SECONDS` specifies the amount of time the robot waits before moving on to the next command. (Note that `wait` is not a command *to the robot*, and therefore it is not preceded by "`robot.`".) In effect, the robot continues to do whatever it had been asked to do just prior to the wait command for the amount of time specified in the wait command. That is, if the robot was asked to move forward and then asked to wait for 1 second, it will move forward for 1 second before applying the command that follows the wait. Here is the complete definition of `yoyo` that uses the `wait` command:

```
void yoyo() // Move robot forward and backward for 1 sec.
{
    robot.forward(1);
    wait(1);
    robot.backward(1);
    wait(1);
    robot.stop();
}
```

Do This: Go ahead and try out the new definitions exactly as above and issue the command to the scribbler. What do you observe? In both cases you should see the robot move forward for 1 second followed by a backward movement for 1 second and then stop.

Adding Parameters to Commands

Take a look at the definition of the `yoyo` function above and you will notice the use of parentheses, `()`, both when defining the function as well as when using it. You have also used other functions earlier with parentheses in them and probably can guess their purpose. Commands or functions can specify certain *parameters* (or values) by placing them within parentheses. For example, all of the movement commands, with the exception of `stop` have one or more numbers that you specify to indicate the speed of the movement. The number of seconds you want the robot to wait can be specified as a parameter in the invocation of the `wait` command. Similarly, you could have chosen to specify the speed of the forward and backward movement in the `yoyo` command, or the amount of time to wait. Below, we show three definitions of the `yoyo` command that make use of parameters:

```
void yoyo1(double speed)
{
    robot.forward(speed, 1);
    robot.backward(speed, 1);
}

void yoyo2(double waitTime)
{
    robot.forward(1, waitTime);
    robot.backward(1, waitTime);
}

void yoyo3(double speed, double waitTime)
{
    robot.forward(speed, waitTime);
    robot.backward(speed, waitTime);
}
```

In the first definition, `yoyo1`, we specify the speed of the forward or backward movement as a parameter. The word `double` means that the following parameter can be a fractional number, such as 0.5. (The exact significance of the word `double` will not concern us now.) Using this definition, you can control the speed of movement with each invocation. For example, if you wanted to move at half speed, you can issue the command:

```
yoyo1(0.5);
```

Similarly, in the definition of `yoyo2` we have parameterized the wait time. In the last case, we have parameterized both speed and wait time. For example, if we wanted the robot to move at half speed and for 1 ½ seconds each time, we would use the command:

```
yoyo3(0.5, 1.5);
```

This way, we can customize individual commands with different values resulting in different variations on the yoyo behavior. Notice in all of the definitions above that we did not have to use the `stop()` command at all. Why?

Saving New Commands in Files

As you can imagine, while working with different behaviors for the robot, you are likely to end up with a large collection of new functions. It would make sense then that you do not have to type in the definitions over and over again. C++ enables you to define new functions and store them in files in a folder on your computer, which can then be easily used over and over again. Let us illustrate this by defining two behaviors: a parameterized yoyo behavior and a wiggle behavior that makes the robot wiggle left and right. The two definitions are given below:

```
// File: moves.h
// Purpose: Two useful robot commands to try out as a file.

void yoyo(double speed, double waitTime)
{
    robot.forward(speed);
    wait(waitTime);
    robot.backward(speed);
    wait(waitTime);
    robot.stop();
}

void wiggle(double speed, double waitTime)
{
    robot.rotate(-speed);
    wait(waitTime);
    robot.rotate(speed);
    wait(waitTime);
    robot.stop();
}
```

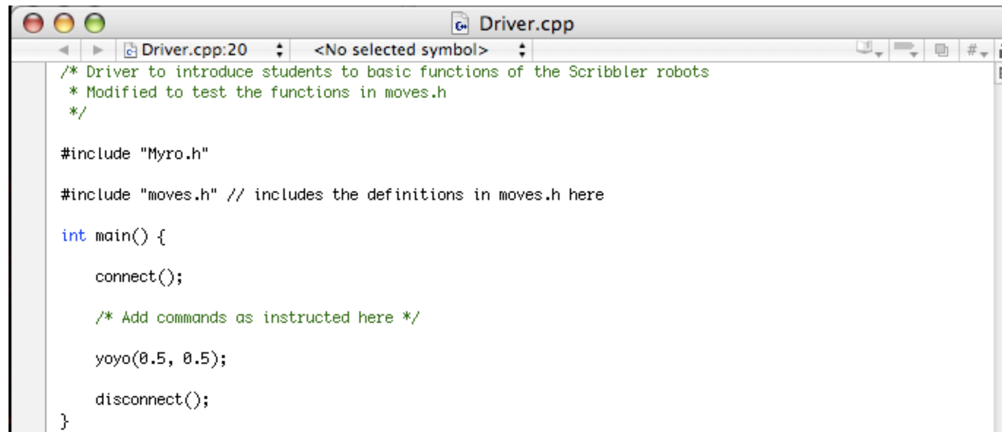
All lines beginning with the "`//`" characters are called comments. These are simply annotations that help us understand and document programs in C++. You can place these comments anywhere, including right after a command. The `//` signs clearly mark the beginning of the comment and anything following it on that line is not interpreted as a command by the computer. C++ also permits multi-line comments, which extend from the characters "`/*`" to the next occurrence of "`*/`". You have already seen several examples of this style of comments. Comments are quite useful and we will make liberal use of them in all our programs.

Do This: To store the `yoyo` and `wiggle` behaviors in a file, you can use your program editor to enter the text containing the two definitions and then save them in a file (let's call it `moves.h`) in your Myro folder (same place you have the `Driver.cpp` file). All files containing C++ definitions of this kind will end with the filename extension `.h` (for *header* file) and you should make sure they are always saved in the same folder as the `Driver.cpp` file. This will make it easy for you as well as the C++ compiler to locate your files when you use them.

Once you have created the file, there is a simple way you can use it. In `Driver.cpp`, just enter the directive:

```
#include "moves.h"
```

in front of the definition of `main`, where the two new commands will be used. For example, the following shows how to use the `yoyo` function after including the `moves` module:



As you can see from above, accessing the commands defined in a file is similar to accessing the capabilities of Myro in the `Myro.h` file. This is a nice feature of C++. In C++, you are encouraged to extend the capabilities of any system by defining your own functions, storing them in files and then using them by including them. Thus including definitions from the `moves.h` file is no different than including definitions from the `Myro.h` file. The directive

```
#include "Myro.h"
```

includes everything in the `Myro.h` file, just as though you had typed it at that place in the driver program. Everything defined in the `Myro.h` library is listed and documented in the C++ Myro Reference Manual. The nice thing that this facility provides is that you can now define your own set of commands that

extend the basic commands available in Myro to customize the behavior of your robot. We will be making use of this over and over again in this course.

Functions as Building Blocks

Now that you have learned how to define new commands using existing ones, it is time to discuss a little more C++. The basic syntax for defining a C++ function takes the form:

```
void <FUNCTION NAME>(<PARAMETERS>)
{
    <SOMETHING>
    ...
    <SOMETHING>
}
```

That is, to define a new function, start by using the word `void` followed by the name of the function (`<FUCTION NAME>`) followed by `<PARAMETERS>` enclosed in parenthesis followed by a left curly brace (`{`). (You will learn later that function definitions do not have to begin with `void`.) This line is followed by the commands that make up the function definition (`<SOMETHING> ... <SOMETHING>`). A right curly brace (`}`) completes the definition. Usually each command is placed on a separate line, and all lines that make up the definition should be indented (aligned) the same amount. The number of spaces that make up the indentation is not that important as long as they are all the same. This may seem a bit awkward and too restricting at first, but you will soon see the value of it, for it makes the definition(s) more readable. For example, look at the following definitions for the `yoyo` function:

```
void yoyo(double speed, double waitTime) {
    robot.forward(speed);
      wait(waitTime);
    robot.backward(speed);
      wait(waitTime);
  robot.stop(); }
```

```
void yoyo(double speed, double waitTime)
   { robot.forward(speed); wait(waitTime);
   robot.backward(speed); wait(waitTime);
   robot.stop(); }
```

Some program editors and *integrated design environments* (IDEs) help you in making your indentations consistent by automatically indenting the next line like the previous.

Another feature built into some IDEs and program editors, which improves readability of C++ programs, is the use of color highlighting. Notice in the above examples (where we use screen shots from an IDE) that pieces of your program appear in different colors. For example, the word `void` in a function definition appears in blue, and the name of your function, `yoyo`, appears in black. Other colors are also used in different situations, look out for them. This particular IDE displays all C++ words (like `void`) in blue and all names defined by you (like `yoyo`) in black. What do you think the other colors indicate?

The idea of defining new functions by using existing functions is very powerful and central to computing. By defining the function `yoyo` as a new function using the existing functions (`forward, backward, wait, stop`) you have *abstracted* a new behavior for your robot. You can define further higher-level functions that use `yoyo` if you want. Thus, functions serve as basic building blocks in defining various robot behaviors, much like the idea of using building blocks to build bigger structures. As an example, consider defining a new behavior for your robot: one that makes it behave like a yoyo twice, followed by wiggling twice. You can do this by defining a new function as follows:

```
void dance()
{
    yoyo(0.5, 0.5);
    yoyo(0.5, 0.5);
    wiggle(0.5, 1);
    wiggle(0.5, 1);
}
```

Do This: Go ahead and add the `dance` function to your `moves.h` file. Put it after the definitions of `yoyo` and `wiggle`, since in C++ we have to define things before we use them. Try the `dance` command on the robot. Now you have a very simple behavior that makes the robot do a little shuffle dance.

Guided by Automated Controls

Earlier we agreed that a robot is a "mechanism guided by automated controls". You can see that by defining functions that carry out more complex movements, you can create modules for many different kinds of behaviors. The modules make up the programs you write, and when they are invoked on the robot, the robot carries out the specified behavior. This is the beginning of being able to define automated controls for a robot. As you learn more about the robot's capabilities and how to access them via functions, you can design and define many kinds of automated behaviors.

Summary

In this chapter, you have learned several commands that make a robot move in different ways. You also learned how to define new commands by defining new C++ functions. Functions serve as basic building blocks in computing and defining new and more complex robot behaviors. C++ has specific syntax rules for writing definitions. You also learned how to save all your function definitions in a file and then using them as a module by including it. While you have learned some very simple robot commands, you have also learned some important concepts in computing that enable the building of more complex behaviors. While the concepts themselves are simple enough, they represent a very powerful and fundamental mechanism employed in almost all software development. In later chapters, we will provide more details about writing functions and also how to structure parameters that customize

individual function invocations. Make sure you do some or all of the exercises in this chapter to review these concepts.

Myro Review

`robot.backward(SPEED);`
Move backwards at `SPEED` (value in the range -1.0…1.0).

`robot.backward(SPEED,SECONDS);`
Move backwards at `SPEED` (value in the range -1.0…1.0) for a time given in `SECONDS`, then stop.

`robot.forward(SPEED);`
Move forward at `SPEED` (value in the range -1.0..1.0).

`robot.forward(SPEED,TIME);`
Move forward at `SPEED` (value in the range -1.0…1.0) for a time given in seconds, then stop.

`robot.motors(LEFT,RIGHT);`
Turn the left motor at `LEFT` speed and right motor at `RIGHT` speed (value in the range -1.0…1.0).

`robot.move(TRANSLATE, ROTATE);`
Move at the `TRANSLATE` and `ROTATE` speeds (value in the range -1.0…1.0).

`robot.rotate(SPEED);`
Rotates at `SPEED` (value in the range -1.0…1.0). Negative values rotate right (clockwise) and positive values rotate left (counter-clockwise).

`robot.stop();`
Stops the robot.

`robot.translate(SPEED);`
Move in a straight line at `SPEED` (value in the range -1.0…1.0). Negative

values specify backward movement and positive values specify forward movement.

`robot.turnLeft(SPEED);`
Turn left at `SPEED` (value in the range -1.0…1.0)

`robot.turnLeft(SPEED,SECONDS);`
Turn left at `SPEED` (value in the range -1.0..1.0) for a time given in seconds, then stops.

`robot.turnRight(SPEED);`
Turn right at `SPEED` (value in the range -1.0..1.0)

`robot.turnRight(SPEED,SECONDS);`
Turn right at `SPEED` (value in the range -1.0..1.0) for a time given in seconds, then stops.

`wait(TIME)`
Pause for the given amount of `TIME` seconds. `TIME` can be a decimal number.

C++ Review

`#include "FILE NAME"`
Includes the contents of the file named `FILE NAME` just as though it had been typed at this location in the current file. The file is sought in the same directory as your other C++ programs.

```
void <FUNCTION NAME>(<PARAMETERS>)
{
    <SOMETHING>
    ...
    <SOMETHING>
}
```

Defines a new command/function named `<FUNCTION NAME>`. A function name should always begin with a letter and can be followed by any sequence of letters, numbers, or underscores (_), and not contain any spaces. Try to choose names that appropriately describe the command being defined.

Exercises

1. Compare the robot's movements in the commands `turnLeft(1)`, `turnRight(1)` and `rotate(1)` and `rotate(-1)`. Closely observe the robot's behavior and then also try the motor commands:

```
robot.motors(-0.5, 0.5);
robot.motors(0.5, -0.5);
robot.motors(0, 0.5);
robot.motors(0.5, 0);
```

Do you notice any difference in the turning behaviors? The `rotate` commands make the robot turn with a radius equivalent to the width of the robot (distance between the two left and right wheels). The `turn` command causes the robot to spin in the same place.

2. Insert a pen in the scribbler's pen port and then issue it commands to go forward for 1 or more seconds and then backwards for the same amount. Does the robot travel the same distance? Does it traverse the same trajectory? Record your observations.

3. Measure the length of the line drawn by the robot in Exercise 2. Write a function `travel(DISTANCE)` to make the robot travel the given DISTANCE. You may use inches or centimeters as your units. Test the function on the robot a few times to see how accurate the line is.

4. Suppose you wanted to turn/spin your robot a given amount, say 90 degrees. Before you try this on your robot, do it yourself. That is, stand in one spot, draw a line dividing your two feet, and then turn 90 degrees. If you have no way of measuring, your turns will only be approximate. You can study the behavior of your robot similarly by issuing it turn/spin commands and making them wait a certain amount. Try and estimate the wait time required to turn 90 degrees (you will have to fix the speed) and write a function to turn that amount. Using this function, write a behavior for your robot to transcribe a square on the floor (you can insert a pen to see how the square turns out).

5. Generalize the wait time obtained in Exercise 3 and write a function called `degreeTurn(DEGREES)`. Each time it is called, it will make the robot turn the specified degrees. Use this function to write a set of instructions to draw a square.

6. Using the functions travel and `degreeTurn`, write a function to draw the Bluetooth logo (See Chapter 1, Exercise 9).

7. Choreograph a simple dance routine for your robot and define functions to carry it out. Make sure you divide the tasks into re-usable moves and as much as possible parameterize the moves so they can be used in customized ways in different steps. Use the building block idea to build more and more complex series of dance moves. Make sure the routine lasts for at least several seconds and it includes at least two repetitions of the entire sequence. You may also make use of the beep command you learned from the last section to incorporate some sounds in your choreography.

8. Record a video of your robot dance and then dub it with a soundtrack of your choosing. Use whatever video editing software is accessible to you. Post the video online on sites like YouTube to share with friends.

9. Lawn mower robots and even vacuuming robots can use specific *choreographed* movements to ensure that they provide full coverage of the area to be serviced. Assuming that the area to be mowed or cleaned is rectangular and without any obstructions, can you design a behavior for your Scribbler to provide full coverage of the area? Describe it in writing. [Hint: Think about how you would mow/vacuum yourself.]