Patterns of Behavior Behaviors

In this lesson, you will learn how the various ways of navigating the warehouse environment break down into a common set of sub-behaviors.

A typical task for the inventory robot may be to retrieve the object in aisle five. How can the robot get there? The robot is not yet advanced enough to determine its own path, so it will require human assistance to find a route. For example:

In order to follow the path above, the robot needs a way to orient itself in the warehouse environment. With the irregular spacing between shelves, distance may not be reliable. Instead, the robot must rely on the floor markings.

Landmarks
This robot will rely on floor markings to help it find its way along the path.
Patterns of Behavior  Behaviors (cont.)

Let’s view a typical task for the robot. Suppose the object it needs to get to is at the X on the map below. How can the robot get there?

We could get there by following this blue route …

... or perhaps this yellow one, or this green one?
Patterns of Behavior  Behaviors (cont.)

Let’s focus on the first (blue) path. What does the robot need to do in order to follow this path? The large behavior breaks down nicely into **smaller behaviors**.

The green path can also be broken down easily into smaller behaviors.
Patterns of Behavior \textbf{Behaviors} (cont.)

Perhaps most interestingly, these two paths seem to share some common sub-behaviors...

\begin{itemize}
  \item \textbf{Blue Path}:
    \begin{itemize}
      \item Forward 4 Lines
      \item Turn Left
      \item Forward 3 Lines
      \item Turn Right
      \item Forward 2 Lines
    \end{itemize}
  \item \textbf{Green Path}:
    \begin{itemize}
      \item Forward 6 Lines
      \item Turn Left
      \item Forward 3 Lines
      \item Turn Left
      \item Forward 2 Lines
    \end{itemize}
\end{itemize}

\textit{Shared behaviors}: These sub-behaviors appear in both of the larger behaviors.

\textbf{End of Section}

This repeating of sub-behaviors is no coincidence. The smaller behaviors actually represent common actions in the warehouse environment, and so they will likely show up in any number of tasks there. C languages like ROBOTC include structures called \textit{functions} that are made to capitalize on exactly this kind of patterned reuse of commands to make your code more adaptable, readable, and reusable. In the next few sections, you will learn to build and use a set of functions to allow rapid construction and reorganization of behaviors to get around the warehouse.
In this lesson, you will learn how to create and use functions for two of the simplest behaviors.

In the last video, we identified several key simple behaviors that, when combined, will make up the complex behavior of moving to the destination shown here.

For each of these simple behaviors, we are going to create a function which encapsulates the behavior in a single, reusable package. Declaring a function basically means you’re creating your own command in the language of ROBOTC, so you can already begin to see how powerful this technique will be once you master it....
Patterns of Behavior Creating and Using Functions (cont.)

1. Open ROBOTC and start a new program.

   1. Create new program
      Select File > New to create a new program.

2. Create the familiar task main, but don’t put anything in it yet.

   ```c
   task main()
   {
   }
   ```

   2. Add this code
      Add a task main() {}.

3. At the top of your program, before task main(), make some space for your functions.

   ```c
   task main()
   {
   }
   ```

   3. Create space
      Add a few blank lines above task main where your functions will go.
Patterns of Behavior  Creating and Using Functions  (cont.)

4. Create the basic skeleton of a function called “turnLeft”. “void” is a keyword used to begin the declaration of the function, much like “task” in task main, and similarly, the function includes a pair of curly braces that will contain the commands in the function body.

```c
void turnLeft()
{
  //Commands go here
}

task main()
{
  //Main task commands
}
```

4. Add this code
This code creates a new function called turnLeft(), and leaves a space between the curly braces to put its commands.

5. Place the commands for a left turn behavior between the function’s { } braces. This version of the left turn uses the rotation sensor to determine when the robot has turned far enough. See Sensing > Line Tracking > Line Tracking (Rotation, Pt. 1 and Pt. 2) for a review of this sensor.

```c
void turnLeft()
{
  nMotorEncoder[motorB] = 0;
  while (nMotorEncoder[motorB] < 160)
  {
    motor[motorC] = -50;
    motor[motorB] = 50;
  }
  motor[motorC] = 0;
  motor[motorB] = 0;
}

task main()
{
  //Main task commands
}
```

5. Add this code
Add the commands for a left turn behavior, between the curly braces of the new function.

The function will run the commands between its braces when it is called, just as task main runs the commands between its braces when the main program is run.

The left turn itself resets the rotation sensor, then turns until the wheel has rotated a set amount, then stops both motors.
Patterns of Behavior  Creating and Using Functions  (cont.)

6. You have created the function turnLeft, and specified the commands that it should run when called (a Rotation Sensor-controlled left turn). To use the function, simply call it by name in the main task.

   ```c
   void turnLeft()
   {
   nMotorEncoder[motorB] = 0;
   while (nMotorEncoder[motorB] < 160)
   {
   motor[motorC] = -50;
   motor[motorB] = 50;
   }
   motor[motorC] = 0;
   motor[motorB] = 0;
   }
   task main()
   {
   turnLeft();
   }
   ```

7. Save your program, download, and run.

6. Add this code
Call the function `turnLeft()` by name, followed by a semicolon, to run it.

7a. Save As
Go to the File menu and select “Save As…”

7b. Name the program
Give this program the name “functionTest”.

7c. Save the program
Press Save to save the program with the new name.
Patterns of Behavior  Creating and Using Functions  (cont.)

Checkpoint
You have created the function turnLeft and told your program to run it in the main task. Does the robot do what you wanted?

Robot running the leftTurn() function
The robot seems to do what we wanted...

7. We said that one of the major advantages of functions was their reusability. Let’s see it in action. Add another left turn, separated from the first one by a 1 second wait.

```c
void turnLeft()
{
    nMotorEncoder[motorB] = 0;
    while(nMotorEncoder[motorB] < 160)
    {
        motor[motorC] = -50;
        motor[motorB] = 50;
    }
    motor[motorC] = 0;
    motor[motorB] = 0;
}

task main()
{
    turnLeft();
    wait1Msec(1000);
    turnLeft();
}
```

6. Add this code
Add another call to turnLeft(), separated from the first one by a 1 second delay.
Patterns of Behavior  Creating and Using Functions  (cont.)

9. Download and run again.

![Robot running two leftTurn() functions](image)
The robot turns once, then waits, and makes a second 90 degree turn.

10. The use of the turnLeft() function to encapsulate the turning behavior in a custom command seems to work well! Now, create one for the right turn, right below the turnLeft() function.

```c
void turnLeft()
{
    nMotorEncoder[motorB] = 0;
    while(nMotorEncoder[motorB] < 160)
    {
        motor[motorC] = -50;
        motor[motorB] = 50;
    }
    motor[motorC] = 0;
    motor[motorB] = 0;
}

void turnRight()
{
    nMotorEncoder[motorC] = 0;
    while(nMotorEncoder[motorC] < 160)
    {
        motor[motorC] = 50;
        motor[motorB] = -50;
    }
    motor[motorC] = 0;
    motor[motorB] = 0;
}
```

10. Add this code
Create a function called turnRight(), below turnLeft(). It should be almost identical, but with a right-turn behavior inside it instead.
Patterns of Behavior  Creating and Using Functions  (cont.)

11. Change the second left turn to a right turn instead. What should the robot do now?

```c
void turnLeft()
{
    nMotorEncoder[motorB] = 0;
    while (nMotorEncoder[motorB] < 160)
    {
        motor[motorC] = -50;
        motor[motorB] = 50;
    }
    motor[motorC] = 0;
    motor[motorB] = 0;
}

void turnRight()
{
    nMotorEncoder[motorC] = 0;
    while (nMotorEncoder[motorC] < 160)
    {
        motor[motorC] = 50;
        motor[motorB] = -50;
    }
    motor[motorC] = 0;
    motor[motorB] = 0;
}

task main()
{
    turnLeft();
    wait1Msec(1000);
    turnRight();
}
```

11. Modify this code
Change the second leftTurn() call to a rightTurn() call instead.
Patterns of Behavior  Creating and Using Functions  (cont.)

12. Download and run again.

*Robot running leftTurn() then rightTurn()*

The robot turns once, then waits, and turns the opposite direction back to the place where it started.

**End of Section**

You now have two of the most common warehouse (and movement, in general) behaviors written as functions. You have also seen the ease with which these functions can be treated as commands in the ROBOTC language to allow their rapid reuse in a situation like the warehouse where they will be seen over and over again.

In the next lessons, you will move these two functions from their current location in the test program (functionTest) into the main program, and complete the remaining behaviors.
Patterns of Behavior  Variables and Functions (Part 1)

In this lesson, you will transfer your two turning behaviors into the program from earlier Warehouse activities, and create functions for the remaining behaviors in the program.

The two turning behaviors you’ve created are useful, but disconnected from the rest of the Warehouse program we’ve been working on. Functions only work in the programs they are declared in, and right now, ours are in a test program called functionTest. Let’s start this lesson by moving them into the main program file, and then we’ll work on turning the other behaviors in the program into functions.

1. Highlight and copy the two functions in your “functionTest” program.

```c
void turnLeft()
{
    nMotorEncoder[motorB] = 0;
    while (nMotorEncoder[motorB] < 160)
    {
        motor[motorC] = -50;
        motor[motorB] = 50;
    }
    motor[motorC] = 0;
    motor[motorB] = 0;
}

void turnRight()
{
    nMotorEncoder[motorC] = 0;
    while (nMotorEncoder[motorC] < 160)
    {
        motor[motorC] = 50;
        motor[motorB] = -50;
    }
    motor[motorC] = 0;
    motor[motorB] = 0;
}
```

1a. Highlight code
Highlight both code and all their associated code as shown.

1b. Copy
Select Edit > Copy to put the highlighted code on the clipboard.
2. Open your LineCounter program.

2a. Open and Compile
File > Open and Compile to open up the program LineCounter.

2b. Find LineCounter
Find LineCounter and click on the program previously saved.

2c. Open LineCounter
Press the Open button to open the program.

3. Paste your functions just above the task main code in the LineCounter program.

```c
const tSensors touchSensor = (tSensors) S1;
const tSensors lightSensor = (tSensors) S2;

int lightValue;
int darkValue;
int sumValue;
int thresholdValue;
int countValue = 0;
```

3a. Place cursor here
Place your cursor on the line just above task main so your pasted code will go there.

3b. Paste
Select Edit > Paste to put the copied code into this program.
Patterns of Behavior  Variables and Functions (Part 1) (cont.)

Checkpoint
And just like that, your program has access to both functions. The rest of the task main could use some cleaning via functions, though, so let’s do that next.

The other two functions we’ll need to create are:
• Threshold calculation
• Moving forward for specific numbers of lines

4. Create the structure for the `findThreshold()` function. Put it at the top of the program, above the newly-pasted `turnLeft()` function.

4. Add this code
Add the basic structure for the `findThreshold()` function that we are about to create.

```
const tSensors touchSensor = (tSensors) S1;
const tSensors lightSensor = (tSensors) S2;

void findThreshold()
{
}

void turnLeft()
{
    nMotorEncoder[motorB] = 0;
    while (nMotorEncoder[motorB] < 160)
    {
        motor[motorC] = -50;
        motor[motorB] = 50;
    }
```
5. Highlight all the lines currently in task main that have to do with automatic threshold calculation, and cut them to the clipboard using the Edit > Cut command.

```c
38  task main()
39  {
40
41      int lightValue;
42      int darkValue;
43      int sumValue;
44      int thresholdValue;
45      int countValue = 0;
46      int lastSeen;

47      nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
48      nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;

49      while (SensorValue(touchSensor) == 0)
50      {
51          nxtDisplayStringAt(0, 31, "Read Light Now");
52          lightValue = SensorValue(lightSensor);
53          wait1Msec(1000);
54          while (SensorValue(touchSensor) == 0)
55          {
56              nxtDisplayStringAt(0, 31, "Read Dark Now");
57          }
58          darkValue = SensorValue(lightSensor);
59          sumValue = lightValue + darkValue;
60          thresholdValue = sumValue / 2;
61          ClearTimer(T1);
62          lastSeen = 1;
```
Patterns of Behavior  Variables and Functions (Part 1) (cont.)

6. Paste the lines into the \{body\} section of the findThreshold() function.

```
const tSensors touchSensor  = (tSensors) S1;
const tSensors lightSensor  = (tSensors) S2;
void findThreshold()
{
    while (SensorValue(touchSensor)==0)
    {
        nxtDisplayStringAt(0, 31, "Read Light Now");
    }
    lightValue = SensorValue(lightSensor);
    wait1Msec(1000);
    while (SensorValue(touchSensor)==0)
    {
        nxtDisplayStringAt(0, 31, "Read Dark Now");
    }
    darkValue = SensorValue(lightSensor);
    sumValue = lightValue+darkValue;
    thresholdValue = sumValue/2;
}
```

**Checkpoint.** Finding a threshold is now as simple as telling the program to findThreshold();. But first, let’s finish writing the other functions.
Patterns of Behavior Variables and Functions (Part 1) (cont.)

7. Create the structure for the `forward7Lines()` function. Put it just under the `findThreshold()` function, outside its last closing brace.

```c
const tSensors touchSensor = (tSensors) S1;
const tSensors lightSensor = (tSensors) S2;

void findThreshold()
{
    while (SensorValue(touchSensor) == 0)
    {
        nxtDisplayStringAt(0, 31, “Read Light Now”);
    }

    lightValue = SensorValue(lightSensor);
    wait1Msec(1000);

    while (SensorValue(touchSensor) == 0)
    {
        nxtDisplayStringAt(0, 31, “Read Dark Now”);
    }

    darkValue = SensorValue(lightSensor);

    sumValue = lightValue + darkValue;
    thresholdValue = sumValue / 2;
}

void forward7Lines()
{
}

void turnLeft()
{

```
8. Highlight all the lines currently in task main that have to do with moving forward for a given number of lines, and cut them to the clipboard using the Edit > Cut command. Delete the unneeded ClearTimer command that’s still in this portion of the code.

8a. Delete this code
Remove the leftover ClearTimer command that is in this section.

8b. Highlight code
Highlight the code that performs the line counting and forward movement based on lines crossed.

8c. Cut
Select Edit > Cut to remove the highlighted code from the program and put it on the clipboard.
Patterns of Behavior  Variables and Functions (Part 1) (cont.)

9. Paste the lines into the {body} section of the forward7Lines() function.

```c
void forward7Lines()
{
    lastSeen = 1;
    while (countValue < 7)
    {
        if (SensorValue(lightSensor) < thresholdValue)
        {
            motor[motorC]=50;
            motor[motorB]=50;
            if (lastSeen == 1)
            {
                countValue = countValue + 1;
                lastSeen = 0;
            }
        }
        else
        {
            lastSeen = 1;
        }
    }
}

void turnLeft()
```

**Checkpoint**
You now have a function that lets you move forward for 7 lines at a time. Save your program, but don’t download yet.
Patterns of Behavior  Variables and Functions (Part 1)  (cont.)

Your program now has access to two additional behaviors: `findThreshold()` and `forward7Lines()`, which we have just extracted into separate functions. In addition, we have the two behaviors we imported from our test file, `turnLeft()` and `turnRight()`. All that remains now is to tell the task main to run them in the desired order... right?

10. Call the new functions in **task main**. Finding the threshold comes first, followed by the movement forward for 7 lines. We'll wait to see if that works before we put in the turns.

```robotc
86    task main()
87    {
88      int lightValue;
89      int darkValue;
90      int sumValue;
91      int thresholdValue;
92      int countValue = 0;
93      int lastSeen;
94
95      nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
96      nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
97
98      findThreshold();
99      forward7Lines();
100    }
```

**10. Add this code**
Tell your robot to run the `findThreshold()` and `forward7Lines()` functions as part of its main program.

11. Save, download, and run. An error message will appear, indicating that something is not right... let's see if we can find what's going wrong.

**11. Compile and Download**
Compile and download your program, but be ready for unusual results... continue on to the next step.
The problem with your program has to do with a property of variables called **scope**. Scope determines **how broadly applicable a value should be**. The variables in your program are all **declared in task main**. But the actual code that’s trying to use them is outside task main, in **separate functions**. They cannot “see” the variables because they are only accessible within task main. It seems silly to us now that this should be the case, but scope actually plays a vital role in letting functions run without interfering with each other.

```c
void findThreshold()
{
    while (SensorValue(touchSensor)==0)
    {
        nxtDisplayStringAt(0, 31, “Read Light Now”);
    }
    lightValue = SensorValue(lightSensor);
}
```

Nevertheless, for now, we’re going to take a very heavy-handed approach to solving this problem. We’re going to move the variables so that they are visible to all functions and tasks by making them **global**. This has advantages and disadvantages, but for now, we’re going with it.
Patterns of Behavior  Variables and Functions (Part 1) (cont.)

12. Highlight all the lines currently in task main that declare variables needed by the functions, and cut them to the clipboard using the Edit > Cut command.

```
task main()
{
    int lightValue;
    int darkValue;
    int sumValue;
    int thresholdValue;
    int countValue = 0;
    int lastSeen;

    nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
    nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;

    findThreshold();
    forward7Lines();
```
Patterns of Behavior  Variables and Functions (Part 1) (cont.)

13. Paste the lines at the top of the program, outside all the functions, but just below the Motor and Sensors auto-generated lines.

```c
const tSensors touchSensor = (tSensors) S1;
const tSensors lightSensor = (tSensors) S2;

void findThreshold()
{
    while (SensorValue(touchSensor) == 0)
    {
        nxtDisplayStringAt(0, 31, "Read Light Now");
    }
}
```

Checkpoint
All your variables are now declared “globally”, and therefore will be visible to all of the functions and tasks in the program. This will have side effects down the line, but for now, it will get us the result we want.

```c
const tSensors touchSensor = (tSensors) S1;
const tSensors lightSensor = (tSensors) S2;
int lightValue;
int darkValue;
int sumValue;
int thresholdValue;
int countValue = 0;
int lastSeen;

void findThreshold()
{
    while (SensorValue(touchSensor) == 0)
```
Patterns of Behavior Variables and Functions (Part 1) (cont.)

End of Section
Download and run your program. The robot should now run exactly seven lines, then stop, using functions. The result isn’t any different from what you’ve seen before, but you know that under the hood, your program is much more powerful and expandible now, and you are now ready to finish solving the warehouse problem. In the next lesson, you will program the remaining necessary functions for the warehouse.
In this lesson, you will learn to adjust the behaviors to fit the actual path you want to take in the warehouse, and make a few additional refinements as necessary.

And now, let’s return to the path we want to take through the warehouse. The needed behaviors (in addition to finding the threshold, which isn’t shown) are:

![Blue Path](image)

- **Blue Path**
  - Forward 4 Lines
  - Turn Left
  - Forward 3 Lines
  - Turn Right
  - Forward 2 Lines

Our currently programmed behaviors are:

- `findThreshold()`, which finds a threshold
- `forward7Lines()`, which travels forward for 7 lines
- `turnLeft()`, which turns the robot 90 degrees to the left
- `turnRight()`, which turns the robot 90 degrees to the right

It looks like we have a fair number of changes to make, so let’s get started.
1. The current forward-for-lines goes for 7 lines, but the path requires a 4, a 3, and a 2. Start by modifying the 7-line behavior to be a 4-line behavior.

```c
void forward4Lines()
{
  lastSeen = 1;
  while (countValue < 4)
  {
    if (SensorValue(lightSensor) < thresholdValue)
    {
      motor[motorC]=50;
      motor[motorB]=50;
      if (lastSeen == 1)
      {
        countValue = countValue + 1;
        lastSeen = 0;
      }
    }
    else
    {
      lastSeen = 1;
    }
  }
}
```

1a. Modify this code
Change the name of the function to indicate its new behavior: going forward for 4 lines, instead of 7.

1b. Modify this code
The number of line counts in the while loop's (condition) is what determines how many lines the robot watches for. Change this number from 7 lines to 4 lines.
Patterns of Behavior  Variables and Functions (Part 2)  (cont.)

2. Adjust your task main to run the new forward-for-4-lines function, and add all the other behaviors which we will need, even if they haven’t been written yet. Note those for later.

```robotc
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109

task main()
{
    nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;

    findThreshold();
    forward4Lines();
    turnLeft();
    forward3Lines();
    turnRight();
    forward2Lines();
}
```

2a. Modify this code
Change the old forward-for-7-lines command to the new forward-for-4-lines one.

2b. Add this code
Add the appropriate function calls for the remaining behaviors, even the ones where we haven’t written the actual functions yet.

Checkpoint
Our functions are in place to perform each of the behaviors we identified in our initial plan. Three of them, forward4Lines, turnLeft, and turnRight are already written. Let’s finish up the others.
Patterns of Behavior  Variables and Functions (Part 2) (cont.)

3. The remaining two behaviors, forward3Lines() and forward2Lines() are very close relatives of the existing forward4Lines(). Copy the forward4Lines() function, and paste two copies of it, which we will turn into the 3-line and 2-line behaviors in the next step.

```c
void forward4Lines()
{
  lastSeen = 1;
  while (countValue < 4)
  {
    if (SensorValue(lightSensor) < thresholdValue)
    {
      motor[motorC]=50;
      motor[motorB]=50;
      if (lastSeen == 1)
      {
        countValue = countValue + 1;
        lastSeen = 0;
      }
    }
    else
    {
      lastSeen = 1;
    }
  }
}
```

3a. Highlight code
Highlight the forward4Lines() function, including its curly braces and everything between them.

3b. Copy
Select Edit > Copy to put the highlighted code on the clipboard.
Patterns of Behavior Variables and Functions (Part 2) (cont.)

4. Paste two copies of the behavior right after the original.

```c
if (SensorValue(lightSensor) < thresholdValue)
{
    motor[motorC] = 50;
    motor[motorB] = 50;
    if (lastSeen == 1)
    {
        countValue = countValue + 1;
        lastSeen = 0;
    }
}
else
{
    lastSeen = 1;
}
}
```

4a. Place cursor here
Place your cursor on the line below the `forward4Lines()` declaration so your pasted code will go there.

4b. Paste
Select Edit > Paste to put the copied code into this program.

4c. Paste again
Paste a second copy of the same code right after the first one.
5. You have three copies of the same behavior. Change two of them to 3-line and 2-line versions.

```c
void forward4Lines()
{
    lastSeen = 1;
    while (countValue < 4)
    {
        if (SensorValue(lightSensor) < thresholdValue)
        {
            motor[motorC]=50;
            motor[motorB]=50;
            if (lastSeen == 1)
            {
                countValue = countValue + 1;
                lastSeen = 0;
            }
        }
        else
        {
            lastSeen = 1;
        }
    }
}

void forward3Lines()
{
    lastSeen = 1;
    while (countValue < 3)
    {
        if (SensorValue(lightSensor) < thresholdValue)
        {
            motor[motorC]=50;
            motor[motorB]=50;
            if (lastSeen == 1)
            {
                countValue = countValue + 1;
                lastSeen = 0;
            }
        }
        else
        {
            lastSeen = 1;
        }
    }
}

void forward2Lines()
{
    lastSeen = 1;
    while (countValue < 2)
    {
        if (SensorValue(lightSensor) < thresholdValue)
        {
            motor[motorC]=50;
            motor[motorB]=50;
            if (lastSeen == 1)
            {
                countValue = countValue + 1;
                lastSeen = 0;
            }
        }
        else
        {
            lastSeen = 1;
        }
    }
}
```

```c
Modify this code
Change the second forward-for-lines behavior to do 3 lines, and the third behavior to do 2 lines.
```
Patterns of Behavior  Variables and Functions (Part 2) (cont.)

Checkpoint
Save, download, and run. The robot will scoot along for 4 lines and turn, just as planned... and then, mysteriously, stop.

The program has a few bugs. This is normal, programs seldom work perfectly on the first try, especially after making big changes like the ones we just did. Continue on to begin fixing them!
6. The problem seems to have occurred in the `forward3Lines()` function, but remember that errors in this function will need to be corrected in its two twins as well. It turns out there are two things keeping this robot from moving on.

```c
void forward3Lines()
{
  lastSeen = 1;
  while (countValue < 3)
  {
  if (SensorValue(lightSensor) < thresholdValue)
    {
      motor[motorC]=50;
      motor[motorB]=50;
    }  
  if (lastSeen == 1)
}
```

**Missing reset**
An ugly side effect of using global variables is that they are shared between functions even when you don't want them to be.

This means that `countValue` is still 4 from the 4-line command that is run earlier in the program. The while loop will immediately kick out without running any additional lines!

**Move only on dark?**
We didn’t really give this much thought, but this bug has been here the whole time. The robot only starts moving if it’s seeing dark, because it doesn’t reach the motor commands otherwise.

```c
void forward3Lines()
{
  lastSeen = 1;
  countValue = 0;
  while (countValue < 3)
  {
    motor[motorC]=50;
    motor[motorB]=50;
    if (SensorValue(lightSensor) < thresholdValue)
    {
    
    }
}
```

**6a. Add this code**
This line resets the value of `countValue` to a fresh count of 0 lines for this new movement.

**6b. Modify this code**
Move the motor lines out of the “dark” portion of the code and put them just outside the if-else statement so they run regardless of whether the robot is seeing light or dark.

**6c. Repeat**
Make the same changes in the 4-line and 2-line versions of the function.
Variables and Functions

Patterns of Behavior  Variables and Functions (Part 2)  (cont.)

Checkpoint
That should solve the stopping problem. Now, let’s look at one other issue that you may have seen.

Clearance
The robot is clearly biased toward the side of the corridor. If it proceeds along this path, it will hit the wall.

7. The robot needs to back up a little before each turn. Add the appropriate movement code to both turning functions.

```c
void turnLeft()
{
    nMotorEncoder[motorB] = 0;
    while (nMotorEncoder[motorB] < -100)
    {
        motor[motorC] = -50;
        motor[motorB] = -50;
    }
    nMotorEncoder[motorB] = 0;
    while (nMotorEncoder[motorB] < 160)
    {
        motor[motorC] = -50;
        motor[motorB] = 50;
    }
    motor[motorC] = 0;
    motor[motorB] = 0;
}
```

7a. Add this code
Make the robot back a little away from the line before turning.

```c
void turnRight()
{
    nMotorEncoder[motorB] = 0;
    while (nMotorEncoder[motorB] < -100)
    {
        motor[motorC] = -50;
        motor[motorB] = -50;
    }
    nMotorEncoder[motorB] = 0;
    while (nMotorEncoder[motorB] < 160)
    {
        motor[motorC] = -50;
        motor[motorB] = 50;
    }
    motor[motorC] = 0;
    motor[motorB] = 0;
}
```

7b. Repeat
Make the same change in the right turn function.
Patterns of Behavior Variables and Functions (Part 2) (cont.)

End of Section
Save, download, and run your program. At long last, the robot should complete its path from start to finish.

So why did we go through all this extra trouble to write functions instead of just putting all the code in the main task? Ask yourself for a moment what changes it would take to your program to use the green or yellow paths instead: the simplicity of reuse will speak for itself... to use the yellow path, all you would have to do is switch two lines in task main!

```
findThreshold();
forward4Lines();
turnLeft();
forward3Lines();
turnRight();
forward2Lines();

findThreshold();
forward6Lines();
turnLeft();
forward3Lines();
turnLeft();
turnLeft();
forward2Lines();

findThreshold();
forward2Lines();
turnLeft();
forward3Lines();
turnRight();
forward4Lines();
```
In this lesson, you will learn how to use functions with parameters to expand their reusability beyond the level of simple copy-and-paste.

There's still one thing about these functions that could stand to be improved. As it is right now, you have to write a new function every time you want to go a different distance. There is a better way. Consider first, what the actual difference in code is between the three functions below:

```c
void forward4Lines()
{
    countValue = 0;
    lastSeen = 1;
    while (countValue < 4)
    {
        motor[motorC]=50;
        motor[motorB]=50;
        if (SensorValue(lightSensor) < thresholdValue)
        {
            if (lastSeen == 1)
            {
                countValue = countValue + 1;
                lastSeen = 0;
            }
        }
        else
        {
            lastSeen = 1;
        }
    }
}

void forward3Lines()
{
    countValue = 0;
    lastSeen = 1;
    while (countValue < 3)
    {
        motor[motorC]=50;
        motor[motorB]=50;
        if (SensorValue(lightSensor) < thresholdValue)
        {
            if (lastSeen == 1)
            {
                countValue = countValue + 1;
                lastSeen = 0;
            }
        }
        else
        {
            lastSeen = 1;
        }
    }
}

void forward2Lines()
{
    countValue = 0;
    lastSeen = 1;
    while (countValue < 2)
    {
        motor[motorC]=50;
        motor[motorB]=50;
        if (SensorValue(lightSensor) < thresholdValue)
        {
            if (lastSeen == 1)
            {
                countValue = countValue + 1;
                lastSeen = 0;
            }
        }
        else
        {
            lastSeen = 1;
        }
    }
}
```

The answer: **one number**.

The difference: These three huge functions differ only in one place: a single number that they use to check how many lines they should run for.
Patterns of Behavior  Variables and Functions (Part 3)  (cont.)

We need to take advantage of this somehow. We can do it using a feature of functions called **parameters**. A parameter is a “placeholder value” that you can use in a function’s **declaration** to stand for a value that you will specify in the function **call**. Because you call a function separately every time you want it to run, this means you can specify a different value for the placeholder parameter every time!

\[
\text{while (countValue < (your value here!))}
\]

1. Save your program as “warehouseParameters”.

2. Delete two of your forward-for-lines functions.

   ```
   void forward3Lines()
   {
     countValue = 0;
     lastSeen = 1;
     while (countValue < 3)
   }
   ```

   ```
   void forward2Lines()
   {
     countValue = 0;
     lastSeen = 1;
     while (countValue < 2)
   }
   ```

   **2a. Delete these functions**
   
   Delete both the `forward3Lines()` and `forward2Lines()` functions.
   
   Make sure you catch all the code inside them, and the closing braces at the ends.

   **7b. Name the program**
   
   Give this program the name “warehouseParameters”.

   **7c. Save the program**
   
   Press Save to save the program with the new name.
Patterns of Behavior

Variables and Functions (Part 3) (cont.)

3. Modify your remaining forward-for-lines function to be a general-purpose parameter version.

```c
void forwardLines(int numLines)
{
    lastSeen = 1;
    countValue = 0;

    while (countValue < numLines)
    {
        if (SensorValue(lightSensor) < thresholdValue)
        {
            motor[motorC]=50;
            motor[motorB]=50;
            if (lastSeen == 1)
            {
                countValue = countValue + 1;
                lastSeen = 0;
            }
        }
    }
}
```

3a. Modify this code
Rename the remaining function to have a more general name.

3b. Modify this code
Creating a parameter looks a lot like a variable declaration, placed between the parentheses that follow the function name.

The parameter “numLines” is created here as an integer, and can be used as a placeholder anywhere in the function {body}.

Its value is not specified here at all. It will (and must) be provided by the task that calls this function.

3c. Modify this code
Put the placeholder parameter “numLines” here in place of the value that we want to be able to fill in.

Placeholding using Parameters

Parameters are like temporary placeholder variables that give the programmer the ability to “substitute” a value inside the function, without actually rewriting the function each time. They require attention in two places: the function declaration, and the function call.

Function declaration:
In the function declaration, the presence of a parameter is announced by declaring it, variable-style, between the (parentheses) following the function name. The parameter can then be used like a value in the rest of the function.

```c
void forwardLines(int numLines)
```

(continued on next page...)
Patterns of Behavior  Variables and Functions (Part 3) (cont.)

4. Modify your main task to take advantage of the new parameter.

    nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
    nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
    findThreshold();
    forwardLines(4);
    turnLeft();
    forwardLines(3);
    turnRight();
    forwardLines(2);
}

Checkpoint. Visualize the substitution that is happening with your parameter.

    task main()
    {
        ...
        ...
        forwardLines(4);
    }

    void forwardLines(int numLines)
    {
        lastSeen = 1;
        while (countValue < numLines)
        {
        }

Function call:
The value of the parameter is specified separately each time the function is run. A value is included in the (parentheses) following the function name when called, and becomes the value of the placeholder in the function’s {body} code!

forwardLines(4);

Parameter supplied
The numeric value 4 will take the place of “numLines”
End of Section
Save, download, and run your program. The robot should complete its path from start to finish.

Take a moment to reflect on what you have done here. You haven’t solved a simple problem using complex tools. You’ve solved a whole family of problems, and created easy-to-use tools that will make it simple to follow any of the paths your robot might need to take through the warehouse.

Your robot is beginning to reach a higher level. You are no longer limited to simply performing single tasks. Your programs, through the use of sensor information and the reuse of their own code in parameterized functions, are beginning to solve the actual problems that underlie the tasks, instead of just the single cases. This approach is many times more powerful, and your understanding of it marks your entry into the real world of programming. Congratulations.