



4

Sensing From Within

I see dead people.
Cole Sear (played by Haley Joel Osment) in *Sixth Sense*,
M. Night Shyamalan, 1999.

Opposite page: Candle Flame
Photo courtesy of Jon Sullivan (www.pdphoto.org)

Cole Sear in Shyamalan's *Sixth Sense* is not referring to dead bodies lying in front of him (for those who have not seen the movie). The five senses that most humans relate to are: touch, vision, balance, hearing, and taste or smell. In all cases our bodies have special sensory receptors that are placed on various parts of the body to enable sensing. For example the taste receptors are concentrated mostly on the tongue; the touch receptors are most sensitive on hands and the face and least on the back and on limbs although they are present all over the body, etc. Besides the difference in the physiology of each kind of receptors there are also different neuronal pathways and thereby sensing mechanisms built into our bodies. Functionally, we can say that each type of sensory system starts with the receptors which convert the thing they sense into electrochemical signals that are transmitted over neurons. Many of these pathways lead to the cerebral cortex in the brain where they are further *processed* (like, "Whoa, that jalapeno is hot!!"). The perceptual system of an organism refers to the set of sensory receptors, the neuronal pathways, and the processing of perceptual information in the brain. The brain is capable of combining sensory information from different receptors to create richer experiences than those facilitated by the individual receptors.

The perceptual system of any organism includes a set of *external* sensors (also called exteroceptors) and some *internal* sensing mechanisms (*interoceptors* or *proprioception*). Can you touch your belly button in the dark? This is because of proprioception. Your body's sensory system also keeps track of the internal state of your body parts, how they are oriented, etc.

Proprioception

Sensing from within

Get something really delicious to eat, like a cookie, or a piece of chocolate, or candy (whatever you fancy!). Hold it in your right hand, and let your right arm hang naturally on your side. Now close your eyes, real tight, and try to eat the thing you are holding. Piece of cake! (well, whatever you picked to eat :-)

Without fail, you were able to pick it up and bring it to your mouth, right?

Give yourself a *Snickers moment* and enjoy the treat.

Sensing is an essential component of being a robot and every robot comes built with internal as well as external sensors. It is not uncommon, for example, to find sensors that are capable of sensing light, temperature, touch, distance to another object, etc. An example of internal sensing in robots is the measurement of movement relative to the robot's internal frame of reference. Sometimes also called *dead reckoning*, it can be a useful sensing mechanism that you can use to design robot behaviors.

Robots employ electromechanical sensors and there are different types of devices available for sensing the same physical quantity. For example, one common sensor found on many robots is a *proximity* sensor. It detects the distance to an object or an obstacle. Proximity sensors can be made using different technologies: infrared light, sonar, or even laser. Depending upon the type of technology used, their accuracy, performance, as well as cost vary: infrared (IR) is the cheapest, and laser is on the expensive side. Lets us take a look at the perceptual system of your Scribbler robot starting with internal sensors.

Proprioception in the Scribbler

The Scribbler has three useful internal sensory mechanisms: *stall*, *time*, and *battery level*. When your program asks the robot to move it doesn't always imply that the robot is actually physically moving. It could be stuck against a wall, for example. The stall sensor in the Scribbler enables you to detect this. You have already seen how you can use time to control behaviors using the `timeRemaining` and `wait` functions. Also, for most movement commands, you can specify how long you want that movement to take place (for example `forward(1, 2.5)` means full-speed forward for 2.5 seconds). Finally, it is also possible to detect battery power level so that you can detect when it is time to change batteries in the robot.

Time

All computers come built-in with an internal clock. In fact, clocks are so essential to the computers we use today that without them we would not have

computers at all! Your Scribbler robot can use the computer's clock to sense time. It is with the help of this clock that we are able to use time in functions like `timeRemaining`, `wait`, and other movement commands. Just with these facilities it is possible to define interesting automated behaviors.

Do This: Design a robot program for the Scribbler to draw a square (say with sides of 6 inches). To accomplish this, you will have to experiment with the movements of the robot and correlate them with time. The two movements you have to pay attention to are the rate at which the robot moves, when it moves in a straight line; and the degree of turn with respect to time. You can write a function for each of these:

```
void travelStraight(double distance) {
    // Travel in a straight line for distance inches
    ...
}

void degreeTurn(double angle) {
    // Spin a total of angle degrees
}
```

That is, figure out by experimentation on your own robot (the results will vary from robot to robot) as to what the correlation is between the distance and the time for a given type of movement above and then use that to define the two functions above. For example, if a robot (hypothetical case) seems to travel at the rate of 25 inches/minute when you issue the command `translate(1.0)`, then to travel 6 inches you will have to translate for a total of $(6*60)/25$ seconds. Try moving your robot forward for varying amounts for time at the same fixed speed. For example try moving the robot forward at speed 0.5 for 3, 4, 5, 6 seconds. Record the distance travelled by the robot for each of those times. You will notice a lot of variation in the distance even for the same set of commands. You may want to average those. Given this data, you can estimate the average amount of time it takes to travel an inch. You can then define `travelStraight` as follows:

```
void travelStraight(double distance) {
    // set up your robot's speed
    double inchesPerSec = <Insert your robot's value here>;

    // Travel in a straight line for distance inches
    robot.forward(1, distance/inchesPerSec);
}
```

Similarly you can also determine the time required for turning a given number of degrees. Try turning the robot at the same speed for varying amounts of time. Experiment how long it takes the robot to turn 360 degrees, 720 degrees, etc. Again, average the data you collect to get the number of degrees per second. Once you have figured out the details use them to write the `degreeTurn` function. Then use the following main program:

```
int main() {
    // Transcribe a square of sides = 6 inches

    connect();
    for (int side = 1; side <= 4; side++) {
        travelStraight(6.0);
        degreeTurn(90.0);
    }
    speak("Look at the beautiful square I made.")
    disconnect();
}
```

Run this program several times. It is unlikely that you will get a perfect square each time. This has to do with the calculations you performed as well with the variation in the robot's motors. They are not precise. Also, it generally takes more power to move from a still stop than to keep moving. Since you have no way of controlling this, at best you can only approximate this type of behavior. Over time, you will also notice that the error will aggregate. This will become evident in doing the exercise below.

Do This: Building on the ideas from the previous exercise, we could further abstract the robot's drawing behavior so that we can ask it to draw any regular

polygon (given the number of sides and length of each side). Write the function:

```
void drawPolygon (int Sides, double Length) {  
    // Draw a regular polygon with Sides number of sides  
    // and each side of length Length.  
}
```

Then, we can write a regular polygon drawing robot program as follows:

```
int main() {  
    // Given the number of sides and the length of each side,  
    // draw a regular polygon  
  
    int nSides;           // number of polygon sides  
    double sideLength;   // length of polygon side  
    connect();  
  
    // First, ask the user for the number of sides and  
    // side length  
    cout << "Given # of sides and side length I will draw"  
        << endl;  
    cout << "a polygon for you. " // string continues...  
        << "Specify side length in inches." << endl;  
  
    cout << "Enter # of sides in the polygon: ";  
    cin >> nSides;  
    cout << "Enter the length of each side: ";  
    cin >> sideLength;  
  
    // Draw the polygon  
    drawPolygon(nSides, sidelength);  
  
    speak("Look! I can draw.");  
    disconnect();  
}
```

To test the program, first try drawing a square of sides 6 inches as in the previous exercise. Then try a triangle, a pentagon, hexagon, etc. Try a polygon with 30 sides of length 0.5 inches. What happens when you give 1 as

the number of sides? What happens when you give zero (0) as the number of sides?

A Slight Detour: Random Walks

One way you can do interesting things with robot drawings is to inject some randomness in how long the robot does something. C++, and most programming languages, typically provide a library function for generating random numbers. Generating random numbers is an interesting process in itself but we will save that discussion for a later time. Random numbers are very useful in all kinds of computer applications, especially games and in simulating real life phenomena. For example, in estimating how many cars might be entering an already crowded highway in the peak of rush hour? Etc. In order to access the random number generating functions in C++ you have to include `cstdlib`, the C++ standard library:

```
#include <cstdlib>
using namespace std;
```

There are lots of features available in this library but we will restrict ourselves with just two, which are described below:

rand() Returns a random unsigned integer between 0 and `RAND_MAX` (inclusive).

RAND_MAX The maximum random number (at least 32767, but usually more than two billion).

Here are some sample outputs that demonstrate these two features:

```
cout << rand() << endl;
```

```
16807
```

```
cout << rand() << endl;
```

```
282475249
```



```
cout << rand() << endl;
```

```
1622650073
```

```
cout << rand() << endl;
```

```
984943658
```

```
cout << rand() << endl;
```

```
1144108930
```

Generally, we do not need random numbers in such a wide range. For example, we are more likely to need a random integer in the range 1 to 6 or in the range 0 to 9. This is easy to do with the C++ *modulus operator*, `%`, which gives the remainder after division. For example, `5 % 2` returns 1, `12 % 4` returns 0, and `17 % 10` returns 7. The following examples show the computation of random integers in the range 0 to 9 (inclusive):

```
cout << rand() % 10 << endl;
```

```
2
```

```
cout << rand() % 10 << endl;
```

```
4
```

```
cout << rand() % 10 << endl;
```

```
8
```

```
cout << rand() % 10 << endl;
```

```
3
```

```
cout << rand() % 10 << endl;
```

```
9
```

```
cout << rand() % 10 << endl;
```

```
0
```

Sometimes we want a fractional random number in some range. For example, to get a random number in the range 0.0 to 1.0 (inclusive) we can divide the number returned by `rand()` by `RAND_MAX`, but we must be careful because both these numbers are integers, and if we divide a smaller integer by a larger one, we will always get zero. As explained in Chapter 3, if you want to get a

fractional (floating point) value from a division, then at least one of its arguments must be floating point. One simple way to do this is to add 0.0 (a floating point number) to either the dividend or divisor:

```
cout << rand() / (0.0 + RAND_MAX) << endl;
```

```
0.519416
```

```
cout << rand() / (0.0 + RAND_MAX) << endl;
```

```
0.830965
```

Although this is a standard “trick,” which would be understood by most programmers, it is a bit misleading (Why would you bother to add zero to something?). Fortunately C++ has a more explicit and readable way of accomplishing the same thing. We can *cast* (explicitly convert) either the divisor or the dividend to the required type (`double`) in the following way:

```
cout << (double) rand() / RAND_MAX << endl;
```

```
0.0345721
```

```
cout << rand() / (double) RAND_MAX << endl;
```

```
0.0534616
```

As you can see, using the C++ standard library is easy enough, and similar to using the Myro library for robot commands. Given the `rand()` function, it is entirely up to you how you use it. Look at the program below:

```
int main() {
    int userInput;
    connect();

    // generate 10 random polygons
    for (int poly = 1; poly <= 10; poly++) {
        // generate a random polygon and draw it
        cout << "Place a new color in the pen port and then..."
            << endl;
        cout << "Enter any number: ";
        cin >> userInput;
        sides = 3 + rand() % 6; // random int in {3, ..., 8}
        size = 2 + rand() % 5; // random int in {2, ..., 6}
    }
}
```

```
        drawPolygon(sides, size);
    }

    // generate a random walk of 20 steps
    for (int step = 1; step <= 20; step++) {
        travelStraight((double) rand() / RAND_MAX);
        degreeTurn(rand() % 360);
    }
    disconnect();
}
```

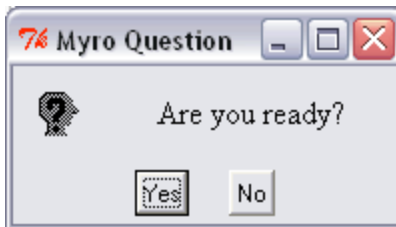
The first loop in the program draws 10 random polygons of sizes ranging from 3 to 8 sides and each side in the range 2 to 6 inches. The second loop carries out a random walk of 20 steps.

Asking Questions?¹

As you can see from above, it is easy to program various kinds of movements into the Scribbler. If there is a pen in the pen port, the Scribbler draws a path. Also in the example above, you can see that we can stop the program temporarily, pretend that we are taking some input and use that as an opportunity to change the pen and then go on. Above, we used the C++ output and input commands `cout <<` and `cin >>` to accomplish this. There is a better way to do this and it uses a function provided in the Myro library:

```
askQuestion("Are you ready?")
```

When this function is executed, a dialog window pops up as shown below:



¹ This section may not reflect the final C++/Myro API.

When you press your mouse on any of the choices (Yes/No), the window disappears and the function returns the name of the key selected by the user as a string. That is, if in the above window you pressed the Yes key, the function will return the value "Yes":

```
cout << askQuestion("Are you ready?") << endl;
```

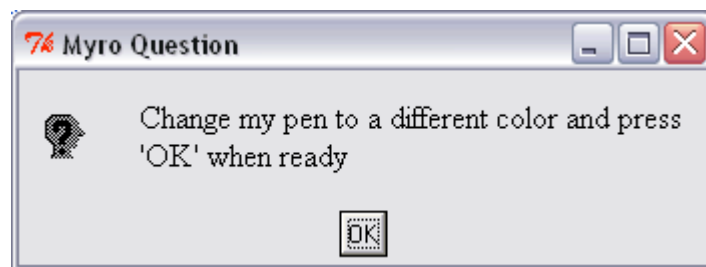
```
Yes
```

The `askQuestion` function can be used in the program above as follows:

```
string userInput = askQuestion ("Change my pen to a different"  
    " color and press 'Yes' when ready.");
```

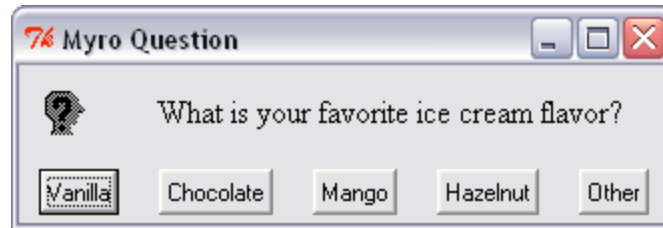
(Notice how to break a long string to make it more readable.) While this is definitely more functional than our previous solution, we can actually do better. For example, what happens when the user presses the No button in the above interaction? One thing you know for sure is that the function will return the string "No". However, the way we are using this function, it really does not matter which key the user presses. `askQuestion` is designed so it can be customized by you so that you can specify how many button choices you want to have in the dialog window as well as what the names of those buttons would be. Here is an illustration of how you would write a better version of the above command:

```
string userInput = askQuestion("Change my pen to a different"  
    " color and press 'OK' when ready", "OK");
```



Now this is certainly better. Notice that the function `askQuestion` can be used with either one parameter or two. If only one parameter is specified, then the default behavior of the function is to offer two button choices: "Yes" and "No". However, using the second parameter you can specify, in a string,² any number of labels that will become the choice buttons. For example,

```
string userInput = askQuestion(  
    "What is your favorite ice cream flavor?",  
    "Vanilla,Chocolate,Mango,Hazelnut,Other");
```



This will be a very handy function to use in many different situations. In the next exercise, try and use this function to become familiar with it.

Do This: Write a Scribbler program of your own that exploits the Scribbler's movements to make random drawings. Make sure you generate drawings with at least three or more colors. Because of random movements, your robot is likely to run into things and get stuck. Help your robot out by picking it up and placing it elsewhere when this happens.

Back to time...

Most programming languages also allow you to access the internal clock to keep track of time, or time elapsed (as in a stop watch), or in any other way you may want to make use of time (as in the case of the `wait`) function. The

² This interface may not represent the final C++/Myro API.

Myro library provides a simple function that can be used to retrieve the current time:

```
cout << currentTime() << endl;
```

```
1169237231.836
```

The value returned by `currentTime` is a number that represents the seconds elapsed since some earlier time, whatever that is. Try issuing the command several times and you will notice that the difference in the values returned by the function represents the real time in seconds. For example:

```
cout << currentTime() << endl;
```

```
1169237351.5580001
```

```
cout << 1169237351.5580001 - 1169237231.836 << endl;
```

```
119.72200012207031
```

That is, 119.722 seconds had elapsed between the two commands above. This provides another way for us to write robot behaviors. So far, we have learned that if you wanted your robot to go forward for 3 seconds, you could either do:

```
robot.forward(1.0, 3.0);
```

OR

```
robot.forward(1.0);  
wait(3.0);  
robot.stop();
```

OR

```
while (timeRemaining(3.0)) {  
    robot.forward(1.0);  
}  
robot.stop();
```

Using the `currentTime` function, there is yet another way to do the same thing:

```
double startTime = currentTime(); // record start time
while (currentTime() - startTime < 3.0) {
    robot.forward(1.0);
}
robot.stop();
```

The above solution uses the internal clock. First, it records the start time. Next it enters the loop which first gets the current time and then checks to see if the difference between the current time and start time is less than 3.0 seconds. If so, the `forward` command is repeated. As soon as the elapsed time gets over 3.0 seconds, the loop terminates. This is another way of using the `while`-loop that you learned in the previous chapter. In the last chapter, you learned that you could write a loop that executed forever as shown below:

```
while (true) {
    <do something>
}
```

The more general form of the `while`-loop is:

```
while (<some condition is true>) {
    <do something>
}
```

That is, you can specify any condition in `<some condition is true>`. The condition is *tested* and if it results in a `true` value, the step(s) specified in `<do something>` is/are performed. The condition is tested again, and so on. In the example above, we use the expression:

```
currentTime() - startTime < 3.0
```

If this condition is true, it implies that the elapsed time since the start is less than 3.0 seconds. If it is false, it implies that more than 3.0 seconds have

elapsed and it results in a `false` value, and the loop stops. Learning about writing such conditions is essential to writing smarter robot programs.

While it may appear that the solution that specified time in the `forward` command itself seemed simple enough (and it is!), you will soon discover that being able to use the internal clock as shown above provides more versatility and functionality in designing robot behaviors. This, for example is how one could program a vacuum cleaning robot to clean a room for 60 minutes:

```
double startTime = currentTime();
while ((currentTime() - startTime)/60.0 < 60.0) {
    cleanRoom();
}
```

You have now seen how to write robot programs that have behaviors or commands that can be repeated a fixed number of times, or forever, or for a certain duration:

```
// do something N times
for (int step = 1; step <= N; step++) {
    do something...
}

// do something forever
while (true) {
    do something...
}

// do something for some duration
while (timeRemaining(duration)) {
    do something...
}

// do something for some duration
double duration = <some time in seconds>;
double startTime = currentTime();
while (currentTime() - startTime < duration) {
    do something...
}
```


All of the above are useful in different situations. Sometimes it becomes a matter of personal preference.

Writing Conditions

Let us spend some time here to learn about conditions you can write in `while`-loops. The first thing to realize is that all conditions result in either of two values: `true` or `false` (interchangeable with 1 and 0, respectively). These are C++ values, just like numbers. You can use them in many ways. Simple conditions can be written using comparison (or relational) operations: `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), `==` (equal to), and `!=` (not equal to). These operations can be used to compare all kinds of values. Here are some examples:

```
cout << (42 > 23) << endl;
```

```
1
```

```
cout << (42 < 23) << endl;
```

```
0
```

```
cout << (42 == 23) << endl;
```

```
0
```

```
cout << (42 != 23) << endl;
```

```
1
```

```
cout << ((42 + 23) < 100) << endl;
```

```
1
```

```
int a = 10;  
int b = 20;  
int c = 10;  
cout << (a == b) << endl;
```

```
0
```

```
cout << (a == c) << endl;
```

```
1
```

```
cout << (a == a) << endl;
```

```
1
```

```
cout << (true == 1) << endl;
```

```
1
```

```
cout << (false == 1) << endl;
```

```
0
```

The last two examples above also show how the values `true` and `false` are related to 1 and 0. `true` is interchangeable with 1 and 0 is interchangeable with `false`. You can form many useful conditions using the comparison operations and all conditions result in either `true` (or 1) or `false` (or 0). You can also compare other values, like strings, using these operations:

```
cout << ("Hello" == "Good Bye") << endl;
```

```
0
```

```
cout << ("Hello" != "Good Bye") << endl;
```

```
1
```

```
cout << ("Elmore" < "Elvis") << endl;
```

```
1
```

```
cout << ("New York" < "Paris") << endl;
```

```
1
```

```
cout << ("A" < "B") << endl;
```

```
1
```

```
cout << ("a" < "A") << endl;
```

```
0
```

Unicode

Text characters have an internal computer coding or representation that enforces lexicographic ordering. This internal encoding is very important in the design of computers and this is what enables all computers and devices like iPhones etc. to exchange information consistently. All language characters in the world have been assigned a standard computer encoding. This is called *Unicode*.

Study the above examples carefully. Two important things to notice are: strings are compared using alphabetical ordering (i.e. lexicographically). Thus "Elmore" is less than "Elvis" since "m" is less than "v" in those strings ("El" being equal in both). That is also why "New York" is less than "Paris" (since "N" is less than "P"). The second important thing to note is that uppercase letters are less than their equivalent lowercase counterparts ("A" is less than "a"). This is by design (see box on the previous page).

Besides relational operations you can build more complex conditional expressions using the *logical operations* (also called *Boolean operations*): && (and), || (or), and ! (not). Here are some examples:

```
cout << ((5 < 7) && (8 > 3)) << endl;
```

```
1
```

```
cout << (! ((5 < 7) && (8 > 3))) << endl;
```

```
0
```

```
cout << ((6 > 7) || (3 > 4)) << endl;
```

```
0
```

```
cout << ((6 > 7) || (3 > 2)) << endl;
```

```
1
```

We can define the meaning of logical operators as follows:

- **<expression-1> && <expression-2>**: Such an expression will result in a value `true` only if both `<expression-1>` and `<expression-2>` are `true`. In all other cases (i.e. if either one or both of `<expression-1>` and `<expression-2>` are `false`) it results in a `false`.
- **<expression-1> || <expression-2>**: Such an expression will result in a value `true` if either `<expression-1>` or `<expression-2>` are `true` or if both are `true`. In all other cases (i.e. if both of

<expression-1> **and** <expression-2> are false) it results in a false.

- **! <expression>**: Such an expression will result in a value `true` if <expression> is false or false if <expression> is true). I.e., it flips or complements the value of expression.

These operators can be combined with relational expressions to form arbitrarily complex conditional expressions. In fact, any decision making in your programs boils down to forming the appropriate conditional expressions. The logical operators were invented by the logician George Boole in the mid 19th century. Boolean algebra, named after Boole, defines some simple, yet important laws that govern the behavior of logical operators. Here are some useful ones:

- `(A || true)` is always true.
- `(! (! A))` is just `A`
- `(A || (B && C))` is the same as `((A || B) && (A || C))`
- `(A && (B || C))` is the same as `((A && B) || (A && C))`
- `(! (A || B))` is the same as `((! A) && (! B))`
- `(! (A && B))` is the same as `((! A) || (! B))`

These identities or properties can help you in simplifying conditional expressions. The conditional expressions can be used to write several useful conditions to control the execution of some program statements. These allow you to write conditional repetitions as:

```
while (<some condition is true>) {  
    <do something>  
}
```

Now you can see why the following is a way of saying, “do something forever”:

```
while (true) {  
    <do something>
```

```
}
```

Since the condition is always `true` the statements will be repeated forever. Similarly, in the loop below:

```
while (timeRemaining(duration)) {  
    <do something>  
}
```

As soon as the `duration` is up, the value of the `timeRemaining(duration)` expression will become `false` and the repetition will stop. Controlling the repetitions based on conditions is a powerful idea in computing. We will be using these extensively to control the behaviors of robots.

Sensing Stall

We mentioned in the beginning of this chapter that the Scribbler also has a way of sensing that it is stalled when trying to move. This is done by using the Myro function `getStall`:

`robot.getStall()` Returns `true` if the robot has stalled, `false` otherwise.

You can use this to detect that the robot has stalled and even use it as a condition in a loop to control behavior. For example:

```
while (! robot.getStall()) {  
    <do something>  
}
```

That is, keep doing `<do something>` until the robot has stalled. Thus, you could write a robot behavior that goes forward until it bumps into something, say a wall, and then stops.

```
while (! robot.getStall()) {  
    forward(1.0);  
}  
robot.stop();  
speak("Ouch! I think I bumped into something!");
```

In the above example, as long as the robot is not stalled, `getStall()` will return `false` and hence the robot will keep going forward (since `! false` is `true`). Once it does bump into something, `getStall()` will return `true` and then the robot will stop and speak.

Do This: Write a complete program for the Scribbler to implement the above behavior and then observe the behavior. Show this to some friends who are not in your course. Ask them for their reactions. You will notice that people will tend to ascribe some form of *intelligence* to the robot. That is, your robot is sensing that it is stuck, and when it is, it stops trying to move and even announces that it is stuck by speaking. We will return to this idea of *artificial intelligence* in a later chapter.

Sensing Battery Power Levels

Your Scribbler robot runs on 6 AA batteries. As with any other electronic device, with use, the batteries will ultimately drain and you will need to replace with fresh ones. Myro provides an internal battery-level sensing function, called `getBattery` that returns the current voltage being supplied by the battery. When the battery levels go down, you will get lower and lower voltages causing erratic behavior. The battery voltage levels of your Scribbler will vary between 0.0 and 9.0 volts (0.0 being totally drained). What low means is something you will have to experiment and find out. The best way to do this is to record the battery level when you insert a fresh set of batteries. Then, over time, keep recording the battery levels as you go.

The Scribbler also has some built-in battery-level indicator lights. The red LED on the robot remains lit when the power levels are high (or in the good range). It starts to flash when the battery level runs low. There is also a similar LED on the Fluke dongle. Can you find it? Just wait until the battery levels run low and you will see it flashing.

Disposing Batteries

Make sure that you dispose used batteries properly and responsibly. Batteries may contain hazardous materials

You can use battery-level sensing to define behaviors for robots so that they are carried out only when there is sufficient power available. For example:

```
while ((robot.getBattery() >= 5.0) && timeRemaining(duration))
{
    <do something>
}
```

That is, as long as battery power is above 5.0 and the time limit has not exceeded duration, <do something>.

World Population, revisited

The ability to write conditional expressions also enables us to define more sophisticated computations. Recall the world population projection example from previous chapter. Given the population growth rate and the current population, you can now write a program to predict the year when the world's population will increase to beyond a given number, say 9 billion. All you have to do is write a condition-driven repetition that has the following structure:

```
int year = <current year>;
double population = <current population>;
double growthRate = <rate of growth>;

// repeat as long as the population stays below 9000000000
while (population < 9000000000.0) {
    // compute the population for the next year
    year = year + 1;
    population = population * (1+growthRate);
}

cout << "By the year " << year << " the world's population"
    << endl;
cout << "will have exceeded 9 billion." << endl;
```

That is, add population growth in the next year if the population is below 9 billion. Keep repeating this until it exceeds 9 billion.

Do This: Complete the program above and compute the year when the world's population will exceed 9 billion. To make your program more useful make sure you ask the user to input the values of the year, population, growth rate, etc. In fact, you can even ask the user to enter the population limit so you make use the program for any kinds of predictions (8 billion? 10 billion?). How would you change the program so it prints the population projection for a given year, say 2100?

Summary

In this chapter you have learned about proprioception or internal sensory mechanisms. The Scribbler robot has three internal sensory mechanisms: time, stall, and battery-level. You have learned how to sense these quantities and also how to use them in defining automated robot behaviors. You also learned about random number generation and used it to define unpredictable robot behaviors. Later, we will also learn how to use random numbers to write games and to simulate natural phenomena. Sensing can also be used to define conditional repetitive behaviors using conditional expressions in `while`-loops. You learned how to construct and write different kinds of conditions using *relational* and *logical* operations. These will also become valuable in defining behaviors that use external sensory mechanisms and also enable us to write more explicit decision-making behaviors. We will learn about these in the next chapter.

Myro Review

`randomNumber ()`

Returns a random number in the range 0.0 and 1.0. This is an alternative Myro function that works just like `(double) rand() / RAND_MAX`, which uses the `rand ()` function from the C++ `cstdlib` library (see below).

`askQuestion(MESSAGE-STRING)`

A dialog window with `MESSAGE-STRING` is displayed with choices: "Yes" and "No". Returns "Yes" or "No" depending on what the user selects.

`askQuestion(MESSAGE-STRING, LIST-OF-OPTIONS)`³

A dialog window with `MESSAGE-STRING` is displayed with choices indicated in `LIST-OF-OPTIONS`, which is string of labels separated by commas. Returns option string depending on what the user selects.

`currentTime()`

The current time, in seconds from an arbitrary starting point in time, many years ago.

`robot.getStall()`

Returns `true` if the robot is stalled when trying to move, `false` otherwise.

`robot.getBattery()`

Returns the current battery power level (in volts). It can be a `double` number between 0.0 and 9.0 with 0.0 indicating no power and 9.0 being the highest. There are also LED power indicators present on the robot. The robot behavior becomes erratic when batteries run low. It is then time to replace all batteries.

C++ Review

`true, false`

These are Boolean or logical values in C++. C++ also defines `true` as 1 and `false` as 0 and they can be used interchangeably.

`<, <=, >, >=, ==, !=`

These are relational operations in C++. They can be used to compare values. See text for details on these operations.

³ This description may not reflect the final C++/Myro API.

`&&`, `||`, `!`

These are logical operations. They can be used to combine any expression that yields Boolean values.

`%`

The modulus operator. The expression `<dividend> % <divisor>` returns the remainder after dividing integer `<dividend>` by integer `<divisor>`.

`rand()`

Returns a random integer between 0 and `RAND_MAX` (inclusive). This function is a part of the `cstdlib` library in C++.

`RAND_MAX`

A constant, which is the maximum random integer that can be returned by the `rand()` function in C++.

Exercises

1. Write a robot program to make your Scribbler draw a five point star. [Hint: Each vertex in the star has an interior angle of 36 degrees.]
2. Experiment with Scribbler movement commands and learn how to make it transcribe a path of any given radius. Write a program to draw a circle of any input diameter.
3. Write a program to draw other shapes: the outline of a house, a stadium, or create art by inserting pens of different colors. Write the program so that the robot stops and asks you for a new color.
4. If you had an open rectangular lawn (with no trees or obstructions in it) you could use a Zanboni like strategy to mow the lawn. Start at one end of the lawn, mow the entire length of it along the longest side, turn around and mow the entire length again, next to the previously mowed area, etc. until you are done. Write a program for your Scribbler to implement this strategy (make the Scribbler draw its path as it goes).

5. Enhance the random drawing program from this chapter to make use of speech. Make the robot, as it is carrying out random movements, to speak out what it is doing. As a result you will have a robot artist that you have created!

6. Rewrite your program from the previous exercise so that the random behavior using each different pen is carried out for 30 seconds.

7. The Myro library also provides a function called, `randomNumber()` that returns a random number in the range 0.0 and 1.0. This is similar to the result of `(double) rand() / RAND_MAX`, which uses the `cstdlib` library function `rand()` that was introduced in this chapter. You can use either based on your own preference. You will have to import the appropriate library depending on the function you choose to use. Experiment with both to convince yourself that these two are equivalent.

8. You need only the function `rand()` to generate random integers in any range. For example, you can get a random integer between 1 and 5 (inclusive) as shown below:

```
int randomValue = 1 + rand() % 5;
```

Given that `rand() % 5` returns an integer between 0 and 4 (inclusive), the above expression will assign a random value between 1 and 5 (inclusive) to `randomValue`. Based on this example, write a new function called `randRange(A, B)` that returns a random integer between A and B (inclusive):

```
int randRange (int A, int B) {  
    // generate a random integer between A..B (inclusive)  
}
```

9. You can also use `rand()` to generate floating point numbers in any required range. For example, the following will generate a random double in the range -1.0 to 1.0 (inclusive):

```
double randomValue = -1.0 + 2.0 * rand() / RAND_MAX;
```

Based on this example, write a new function called `randInterval(A, B)` that returns a random `double` number between `A` and `B`:

```
double randInterval (double A, double B) {  
    // generate a random floating point number in (A, B).  
}
```

10.⁴ What kinds of things can your robot talk about? You have already seen how to make the robot/computer speak a given sentence or phrase. But the robot can also “talk” about other things, like the time or the weather.

One way to get the current time and date is to import another C++ library called `ctime`:

```
#include <ctime>
```

The time library provides functions called `time` and `localtime` that work as follows. The invocation `time(NULL)` returns a `time_t` object, which represents the current date and time (usually as the number of seconds from some fixed date many years ago). It can be saved in a variable by:

```
time_t now = time(NULL);
```

The `localtime` function takes a pointer to a `time_t` object and returns a pointer to a structure `tm` with the following `int` fields:

- `tm_year`: years since 1900
- `tm_month`: month (0–11)
- `tm_mday`: day (1–31)
- `tm_hour`: hour (0–23)
- `tm_min`: minute (0–59)
- `tm_sec`: seconds (0–59)

⁴ This exercise uses *structures* and *pointers*, C++ features that have not been discussed so far. Do not attempt this exercise unless you know about them from another source or want to learn about them on your own!

- `tm_wday`: weekday (0–6)
- `tm_yday`: day of the year (0–365)
- `tm_isdst`: whether it is using daylight savings time or not (>0 if DST, 0 if not DST, <0 if unknown)

For example, to print out the current year you could write:

```
cout << "The year is "  
  << localtime(&now)->tm_year + 1900  
  << endl;
```

```
2008
```

There are several other useful functions in the `ctime` library. For example `asctime` takes a pointer to a `tm` structure and returns a string containing a human-readable date and time:

```
cout << "Local time is " << asctime(localtime(&now)) << endl;
```

```
Local time is Tue Nov 11 14:14:52 2008
```

Write a C++ program that speaks out the current date and time.

11. Write a function `printTime` that takes an `int` in the range 0 to 23 representing the hour and an `int` in the range 0 to 59 representing the minute, and that prints out the time in 12-hour format with AM or PM appended. For example:

```
printTime (8, 30);
```

```
8:30 AM
```

```
printTime (16, 30);
```

```
4:30 PM
```

```
printTime (12, 15);
```

```
12:15 PM
```

```
printTime (0, 15);
```

```
12:15 AM
```

