



5

Sensing The World

I see all obstacles in my way.
From the song *I can see clearly now*,
Johnny Nash, 1972.

Opposite page: The Senses
Photo courtesy of Blogosphere (cultura.blogosfere.it)

In the previous chapter you learned how proprioception: sensing time, stall, and battery-level can be used in writing simple yet interesting robot behaviors. All robots also come equipped with a suite of external sensors (or exteroceptors) that can sense various things in the environment. Sensing makes the robot *aware* of its environment and can be used to define more intelligent behaviors. Sensing is also related to another important concept in computing: *input*. Computers act on different kinds of information: numbers, text, sounds, images, etc. to produce useful applications. Acquiring information to be processed is generally referred to as input. In this chapter we will also see how other forms of input can be acquired for a program to process. First, let us focus on Scribbler's sensors.

Scribbler Sensors

The Scribbler robot can sense the amount of ambient light, the presence (or absence) of obstacles around it, and also take pictures from its camera. Several devices (or sensors) are located on the Scribbler (see picture on the previous page). Here is a short description of these:

Camera: The camera can take a still picture of whatever the robot is currently “seeing”.

Light: There are three light sensors present on the robot. These are located in the three holes present on the front of the robot. These sensors can detect the levels of brightness (or darkness). These can be used to detect variations in ambience light in a room. Also, using the information acquired from the camera, the Scribbler makes available an alternative set of brightness sensors (left, center, and right).

Proximity: There are two sets of these on the Scribbler: IR Sensors (left and right) on the front; and Obstacle Sensors (left, center, and right) on the Fluke dongle. They can be used to detect objects on the front and on its sides.

Getting to know the sensors

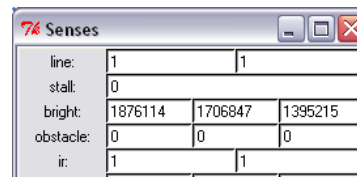
Sensing using the sensors provided in the Scribbler is easy. Once you are familiar with the details of sensor behaviors you will be able to use them in your programs to design interesting creature-like behaviors for your Scribbler. But first, we must spend some time getting to know these sensors; how to access the information reported by them; and what this information looks like. As for the internal sensors, Myro provides several functions that can be used to acquire data from each sensor device. Where multiple sensors are available, you also have the option of obtaining data from all the sensors or selectively from an individual sensor.

Do This: perhaps the best way to get a quick look at the overall behavior of all the sensors is to use the Myro function `senses`:

```
robot.senses();
```

This results in a window (see picture on right) showing all of the sensor values (except the camera) in real time. They are updated every second. You should move the robot around and see how the sensor values change. The window also displays the values of the stall sensor as well as the battery level. The leftmost value in each of the sensor sets (light, IR, obstacle, and bright) is the value of the left sensor, followed by center (if present), and then the right.

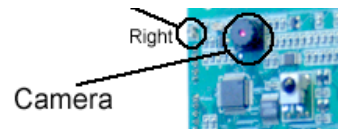
Scribbler Sensors



7% Senses			
line:	1		1
stall:	0		
bright:	1876114	1706847	1395215
obstacle:	0	0	0
ir:	1		1

The Camera

The camera can be used to take pictures of the robot's current view. As you can see, the camera is located on the Fluke dongle. The view of the image taken by the camera will depend on the orientation of the robot



(and the dongle). To take pictures from the camera you can use the `takePicture` command:

```
takePicture()  
takePicture("color")  
takePicture("gray")
```

Takes a picture and returns a picture object. By default, when no parameters are specified, the picture is in color. Using the `"gray"` option, you can get a grayscale picture. Example:

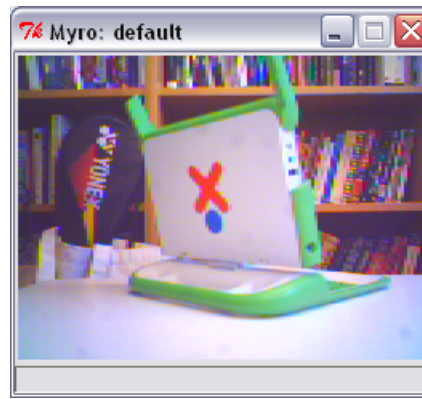
```
picture p = robot.takePicture();  
show(p);
```

Alternately, you can also do:

```
show(robot.takePicture());
```

Once you take a picture from the camera, you can do many things with it. For example, you may want to see if there is a laptop computer present in the picture. Image processing is a vast subfield of computer science and has applications in many areas. Understanding an image is quite complex but something we do quite naturally. For example, in the picture above, we have no problem locating the laptop, the bookcase in the background, and even a case for a badminton racket (or, if you prefer racquet). The camera on the Scribbler is its most complex sensory device that will require a lot of computational effort and energy to use it in designing behaviors. In the simplest case, the camera can serve as your remote "eyes" on the robot. We may not have mentioned this earlier, but the range of the Bluetooth wireless on the robot is 100 meters. In Chapter 9 we will learn several ways of using the pictures. For now, if you take a picture from the camera and would like to save it for later use, you use the Myro command, `savePicture`, as in:

```
savePicture(p, "office-scene.jpg");
```



The file `office-scene.jpg` will be saved in the same folder as your C++ program. You can also use `savePicture` to save a series of pictures from the camera and turn it into an animated “movie” (an animated gif image). This is illustrated in the example below.

Do This: First try out all the commands for taking and saving pictures. Make sure that you are comfortable using them. Try taking some grayscale pictures as well. Suppose your robot has ventured into a place where you cannot see it but it is still in communication range with your computer. You would like to be able to look around to see where it is. Perhaps it is in a new room. You can ask the robot to turn around and take several pictures and show you around. You can do this using a combination of `rotate` and `takePicture` commands as shown below:

```
while (timeRemaining(30)) {
    show(robot.takePicture());
    robot.turnLeft(0.5, 0.2);
}
```

That is, take a picture and then turn for 0.2 seconds, repeating the two steps for 30 seconds. If you watch the picture window that pops up, you will see successive pictures of the robot’s views. Try this a few times and see if you can count how many different images you are able to see. Next, change the `takePicture` command to take grayscale images. Can you count how many images it took this time? There is of course an easier way to do this:

```
int N = 0;
while (timeRemaining(30)) {
    show(robot.takePicture());
    robot.turnLeft(0.5, 0.2);
    N = N + 1;
}
cout << N << endl;
```

Now it will output the number of images it takes. You will notice that it is able to take many more grayscale images than color ones. This is because color images have a lot more information in them than grayscale images (see

text on right). A 256x192 color image requires 256x192x3 (= 147, 456) bytes of data whereas a grayscale image requires only 256x192 (= 49,152) bytes. The more data you have to transfer from the robot to the computer, the longer it takes.

You can also save an animated GIF of the images generated by the robot by using the `savePicture` command by accumulating a series of images in a vector. This is shown below:

```
vector<picture> Pics;
while (timeRemaining(30)) {
    picture pic =
        robot.takePicture();
    show(pic);
    Pics.push_back(pic);
    robot.turnLeft(0.5, 0.2);
}
savePicture(Pics,
    "office-movie.gif");
```

First we create an empty vector called, `Pics`.

Then we append each successive picture taken by the camera onto the vector. This is accomplished by `Pics.push_back(pic)`, which inserts `pic` at the back (end) of `Pics`. Once all the images are accumulated, we use `savePicture` to store the entire set as an animated GIF. You will be able to view the animated GIF inside any web browser. Just load the file into a web browser and it will play all the images as a movie.

There are many more interesting ways that one can use images from the camera. In Chapter 9 we will explore images in more detail. For now, let us take a look at Scribbler's other sensors.

Pixels

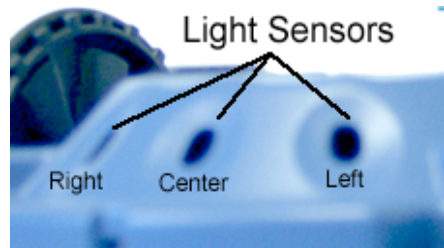
Each image is made up of several tiny picture elements or *pixels*. In a color image, each pixel contains color information which is made up of the amount of red, green, and blue (RGB). Each of these values is in the range 0..255 and hence it takes 3 bytes or 24-bits to store the information contained in a single pixel. A pixel that is colored pure red will have the RGB values (255, 0, 0). A grayscale image, on the other hand only contains the level of gray in a pixel which can be represented in a single byte (or 8-bits) as a number ranging from 0..255 (where 0 is black and 255 is white).

Light Sensing

The following function is available to obtain values of light sensors:

getLight(<POSITION>) Returns the current value in the <POSITION> light sensor. <POSITION> can either be one of "left", "center", "right" or one of the numbers 0, 1, 2. The positions 0, 1, and 2 correspond to the left, center, and right sensors.

Examples:



```
cout << robot.getLight("left") << endl;
```

```
135
```

```
cout << robot.getLight(0) << endl;
```

```
135
```

```
cout << robot.getLight("center") << endl;
```

```
3716
```

```
cout << robot.getLight(1) << endl;
```

```
3716
```

```
cout << robot.getLight("right") << endl;
```

```
75
```

```
cout << robot.getLight(2) << endl;
```

```
75
```

The values being reported by these sensors can be in the range [0..5000] where low values imply bright light and high values imply darkness. The above values were taken in ambient light with one finger completely covering the center sensor. Thus, the darker it is, the higher the value reported. In a way, you could even call it a darkness sensor. Later, we will see how we can easily transform these values in many different ways to affect robot behaviors.

It would be a good idea to use the `senses` function to play around with the light sensors and observe their values. Try to move the robot around to see how the values change. Turn off the lights in the room, or cover the sensors with your fingers, etc.

The camera present on the Fluke dongle can also be used as a kind of brightness sensor. This is done by averaging the brightness values in different zones of the camera image. In a way, you can think of it as a virtual sensor. That is, it doesn't physically exist but is embedded in the functionality of the camera. The function `getBright` is similar to `getLight` in how it can be used to obtain brightness values:

`getBright (<POSITION>)` Returns the current value in the `<POSITION>` light sensor. `<POSITION>` can either be one of "left", "center", "right" or one of the numbers 0, 1, 2. The positions 0, 1, and 2 correspond to the left, center, and right sensors. Examples:

```
cout << robot.getBright("left") << endl;
```

```
2124047
```

```
cout << robot.getBright(0) << endl;
```

```
2124047
```

```
cout << robot.getBright("center") << endl;
```

```
1819625
```

```
cout << robot.getBright(1) << endl;
```

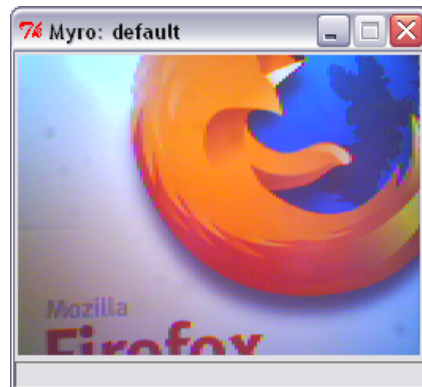
```
1819625
```

```
cout << robot.getBright("right")  
<< endl;
```

```
1471890
```

```
cout << robot.getBright(2) << endl;
```

```
1471890
```



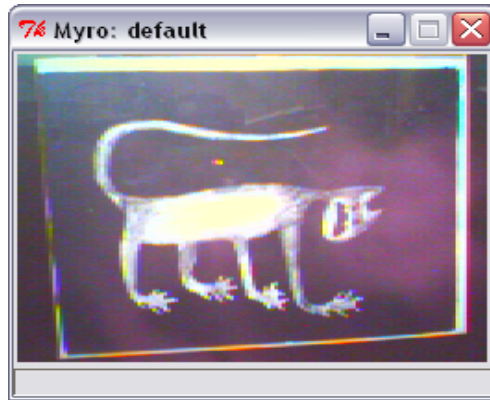
The above values are from the camera image of the Firefox poster (see picture above). The values being reported by these sensors can vary depending on the view of the camera and the resulting brightness levels of the image. But you will notice that higher values imply bright segments and lower values imply darkness. For example, here is another set of values based on the image shown on this page.

```
cout << robot.getBright("left") << ", "  
      << robot.getBright("center") << ", "  
      << robot.getBright("right") << endl;
```

```
1590288, 1736767, 1491282
```

As we can see, a darker image is likely to produce lower brightness values. In the image, the center of the image is brighter than its left or right sections.

It is also important to note the differences in the nature of information being reported by the `getLight` and `getBright` sensors. The first one reports the amount of ambient light being sensed by the robot (including the light above the robot). The second one is an average of the brightness obtained from the image seen from the camera. These can be used in many different ways as we will see later.



Do This: The program shown below uses a normalization function to normalize light sensor values in the range $[0.0..1.0]$ relative to the values of ambient light. Then, the normalized left and right light sensor values are used to drive the left and right motors of the robot.

```
#include "Myro.h"

double Ambient;
// This function normalizes light sensor values to 0.0..1.0
double normalize (int v) {
    if (v > Ambient) {
        v = Ambient;
    }

    return 1.0 - v/Ambient;
}

int main() {
    connect();

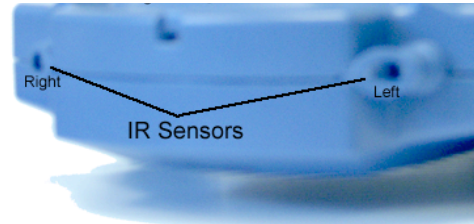
    // record average ambient light values
    Ambient = (robot.getLight("left") +
               robot.getLight("center") +
               robot.getLight("right")) / 3.0;

    // Run the robot for 60 seconds
    while (timeRemaining(60)) {
        int L = robot.getLight("left");
        int R = robot.getLight("right");
        // motors run proportional to light
        robot.motors(normalize(L), normalize(R));
    }
    disconnect();
}
```

Run the above program on your Scribbler robot and observe its behavior. You will need a flashlight to affect better reactions. When the program is running, try to shine the flashlight on one of the light sensors (left or right). Observe the behavior. Do you think the robot is behaving like an insect? Which one? Study the program above carefully. We will also return to the idea of making robots behave like insects in the next chapter.

Proximity Sensing

The Scribbler has two sets of proximity detectors. There are two infrared (IR) sensors on the front of the robot and there are three additional IR obstacle sensors on the Fluke dongle. The following function is available to obtain values of the front IR sensors:



getIR(<POSITION>) Returns the current value in the <POSITION> IR sensor. <POSITION> can either be one of "left" or "right" or one of the numbers 0, 1. The positions 0 and 1 correspond to the left, center, and right sensors.

Examples:

```
cout << robot.getIR("left") << endl;
```

```
1
```

```
cout << robot.getIR(0) << endl;
```

```
1
```

```
cout << robot.getIR("right") << endl;
```

```
0
```

```
cout << robot.getIR(1) << endl;
```

```
0
```

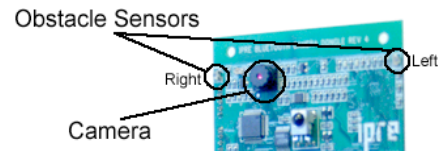
IR sensors return either a 1 or a 0. A value of 1 implies that there is nothing in close proximity of the front of that sensor and a 0 implies that there is something right in front of it. These sensors can be used to detect the presence or absence of obstacles in front of the robot. The left and right IR sensors are places far enough apart that they can be used to detect individual obstacles on either side.

Do This: Run the `senses` function and observe the values of the IR sensors. Place various objects in front of the robot and look at the values of the IR

proximity sensors. Take your notebook and place it in front of the robot about two feet away. Slowly move the notebook closer to the robot. Notice how the value of the IR sensor changes from a 1 to a 0 and then move the notebook away again. Can you figure out how far (near) the obstacle should be before it is detected (or cleared)? Try moving the notebook from side to side. Again notice the values of the IR sensors.

The Fluke dongle has an additional set of obstacle sensors on it. These are also IR sensors but behave very differently in terms of the kinds of values they report. The following function is available to obtain values of the obstacle IR sensors:

getObstacle (<POSITION>) Returns the current value in the <POSITION> IR sensor. <POSITION> can either be one of "left", "center", or "right" or one of the numbers 0, 1, or 2. The positions 0, 1, and 2 correspond to the left, center, and right sensors. Examples:



```
cout << robot.getObstacle("left") << endl;
```

```
1703
```

```
cout << robot.getObstacle(0) << endl;
```

```
1703
```

```
cout << robot.getObstacle("center") << endl;
```

```
1128
```

```
cout << robot.getObstacle(1) << endl;
```

```
1128
```

```
cout << robot.getObstacle("right") << endl;
```

```
142
```

```
cout << robot.getObstacle(2) << endl;
```

```
142
```

The values reported by these sensors range from 0 to 7000. A 0 implies there is nothing in front of the sensor where as a high number implies the presence of an object. The sensors on the sides can be used to detect the presence (or absence of walls on the sides).

Do This: Modify your program that uses the game pad controller (`gamepad()`) so that it also issues the `senses` command to get the real time sensor display. Place your Scribbler on the floor, turn it on, and run your new driver program. Our objective here is to really "get into the robot's mind" and drive it around without ever looking at the robot. Also resist the temptation to take a picture. You can use the information displayed by the sensors to navigate the robot. Try driving it to a dark spot, or the brightest spot in the room. Try driving it so it never hits any objects. Can you detect when it hits something? If it does get stuck, try to maneuver it out of the jam! This exercise will give you a pretty good idea of what the robot senses, how it can use its sensors, and to the range of behaviors it may be capable of. You will find this exercise a little hard to carry out, but it will give you a good idea as to what should go into the *brains* of such robots when you actually try to design them. We will try and revisit this scenario as we build various robot programs.

Also do this: Try out the program below. It is very similar to the program above that used the normalized light sensors.

```
#include "Myro.h"
int main() {
    connect();
    // Run the robot for 60 seconds
    while (timeRemaining(60)) {
        int L = robot.getIR("left");
        int R = robot.getIR("right");
        // motors run proportional to IR values
        robot.motors(R, L);
    }
    disconnect();
}
```

Since the IR sensors report 0 or 1 values, you do not need to normalize them. Also notice that we are putting the left sensor value (L) into the right motor and the right sensor value (R) into the left motor. Run the program and observe the robot's behavior. Keep a notebook handy and try to place it in front of the robot. Also place it slightly on the left or on the right. What happens? Can you summarize what the robot is doing? What happens when you switch the R and L values to the motors?

You can see how simple programs like the ones we have seen above can result in interesting automated control strategies for robots. You can also define completely automated behaviors or even a combination of manual and automated behaviors for robots. In the next chapter we will explore several robot behaviors. First, it is time to learn about vectors in C++.

Vectors and Lists in C++

You have seen above that we used vectors to accumulate a series of pictures from the camera to generate an animated GIF. Vectors are a very useful way of collecting a bunch of information and C++ provides a whole host of useful operations and functions that enable manipulation of vectors. In C++, a vector is a sequence of objects of the same type. The objects could be anything: numbers, letters, strings, images, etc. To use vectors, you need to include their header file:

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

Then an empty vector of integers called N can be defined:

```
vector<int> N;
cout << N << endl;
```

```
{ }
```

(The output of vectors is not built into C++, but is defined in the C++/Myro API.) An empty vector does not contain anything initially, but things can be added to it later. For example,

```
<VECTOR>.push_back(<VALUE>);
```

pushes <VALUE> onto the back (end) of <VECTOR>:

```
N.push_back(7);
N.push_back(14);
cout << N << endl;
```

```
{7, 14}
```

A more convenient way to initialize a vector is:

```
int N_init[] = {7, 14, 17, 20, 27};
vector<int> N (N_init, N_init + 5);
cout << N << endl;
```

```
{7, 14, 17, 20, 27}
```

The first line initializes a C++ “array” (a feature we haven’t discussed) to the five specified numbers. The second line initializes the vector `N` to the values of the array between `N_init` (the location of the first element) and `N_init + 5` (the location after the last). This may seem a bit mysterious (and it is), but it is not too hard to get used to. You can have vectors of any type, for example of `string`s or `doubles`:

```
string Cities_init[] =
    {"New York", "Dar es Salaam", "Moscow"};
vector<string> Cities (Cities_init, Cities_init + 3);

double FN_init[] = {3.14159, 2.718, 42};
vector<double> FamousNumbers (FN_init, FN_init + 3);

cout << Cities << endl;
cout << FamousNumbers << endl;
```



```
{New York, Dar es Salaam, Moscow}
{3.14159, 2.718, 42}
```

C++ provides several useful functions that enable manipulation of vectors. Below, we will show some examples using the variables defined above:

```
cout << N.size() << endl;
```

```
5
```

From the above, you can see that the function `size()` takes a vector and returns the size or the number of objects in the vector. An empty vector has zero objects in it. You can also access individual elements in a vector using the indexing operation (as in `FamousNumbers[0]`). The first element in a list has index 0 and the last element in a list of n elements will have an index $n-1$:

```
cout << FamousNumbers[0] << endl;
```

```
3.14159
```

You can insert the elements from one vector into another. For example, to insert the elements of `N` at the beginning of `FamousNumbers`:

```
FamousNumbers.insert (FamousNumbers.begin(),
    N.begin(), N.end());
cout << FamousNumbers << endl;
```

```
{7, 14, 17, 20, 27, 3.14159, 2.718, 42}
```

The first argument to `insert` specifies where to insert the elements, namely, at the beginning of `FamousNumbers`. The second and third arguments specify the elements to be inserted, namely those from the beginning up to the end of `N`. You can also insert new elements at the end of a vector:

```
N.insert (N.end(), FamousNumbers.begin(),
    FamousNumbers.end());
cout << N << endl;
```

```
{7, 14, 17, 20, 27, 7, 14, 17, 20, 27, 3, 2, 42}
```

The eight values from `FamousNumbers` are inserted before the end of `N`. However, because `N` is an integer vector, the values from `FamousNumbers` are converted from `double` to `int`.

These operations are summarized in more detail at the end of the chapter. The vector types are examples of standard *sequence containers* provided by C++; different types of sequences provide different operations. As we have seen, vectors permit arbitrary elements to be accessed by the indexing operation (denoted by square brackets). Indexing is efficient because the vector elements are stored in contiguous memory locations, but that means that it is inefficient to insert elements anywhere but at the beginning or end of the vector.

C++ provides several other sequence containers, including the *list*. It is similar to the vector, but is stored differently in memory (in doubly linked, rather than contiguous, locations), and so it provides different, efficient operations. For example, it doesn't allow indexing, but it provides other useful list operations, such as `sort` and `reverse`:

```
int SwankeyZips_init[] = {90210, 33139, 60611, 10036};
list<int> SwankeyZips (SwankeyZips_init, SwankeyZips_init+4);
cout << SwankeyZips << endl;
```

```
[90210, 33139, 60611, 10036]
```

(Again, printing of lists is not built into C++, but is defined in the C++ Myro API.)

```
SwankyZips.sort();
cout << SwankyZips << endl;
```

```
[10036, 33139, 60611, 90210]
```

```
SwankyZips.reverse()
cout << SwankyZips << endl;
```

```
[90210, 60611, 33139, 10036]
```

```
SwankyZips.push_back(19010);
cout << SwankyZips << endl;
```

```
[90210, 60611, 33139, 10036, 19010]
```

`sort` rearranges elements in the list in ascending order. `reverse` reverses the order of elements in the list, and `push_back` appends an element to the back (end) of the list. Some other useful list operations are listed at the end of the chapter.

Both vectors and lists are sequences and hence they can be used to perform repetitions. For example, we can use indexing to get the individual strings in `Cities`:

```
for (int i = 0; i < Cities.size(); i++) {
    cout << Cities[i] << endl;
}
```

```
New York
Dar es Salaam
Moscow
```

However, there is a more general way to iterate through any sequence container, including vectors:

```
vector<string>::const_iterator city;
for (city = Cities.begin(); city != Cities.end(); city++) {
    cout << *city << endl;
}
```

```
New York
Dar es Salaam
Moscow
```

The first line declares `city` to be a *constant iterator* for string vectors; making it `const` means that it can be used to read the vector, but not write into it. The iterator `city` points at consecutive values in the vector `Cities`, from `Cities.begin()` up to `Cities.end()`, and the statements inside the loop are executed once for each value of `city`. The “*” in front of `city` means that we will use the value *pointed to* by the iterator, rather than the iterator itself. (Pointers are discussed in more detail elsewhere.) The same approach can be used for iterating through lists; for example:

```
list<int>::const_iterator zip;
for (zip = SwankeyZipsList.begin();
     zip != SwankeyZipsList.end(); zip++) {
    cout << *zip << " ";
}
```

```
90210 60611 33139 10036 19010
```

Strings are sequences and have properties similar to vectors. That is, the string:

```
string ABC = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

is a sequence of 26 letters. You can write a loop that runs through each individual letter in the string and speaks it out as follows:

```
for (string::const_iterator letter = ABC.begin();
     letter < ABC.end(); letter++) {
    speak( *letter );
}
```

In light of the list operations presented above review some of the sensing examples from earlier in the chapter. We will be using vectors and lists in many examples in the remainder of the text. For now, let us return to the topic of sensing.

Extrasensory Perception?

You have seen many ways of acquiring sensory information using the robot's sensors. In addition to the robot itself, you should be aware that your computer also has several "sensors" or devices to acquire all kinds of data. For example, you have already seen how, using the `cin` input stream, you can input some values into your C++ programs:

```
int N;
cout << "Enter a number: ";
cin >> N;
cout << "You entered " << N << endl;
```

```
Enter a number: 42
You entered 42
```

Indeed, there are other ways you can acquire information into your C++ programs. For example, you can input some data from a file in your folder. In Chapter 1 you also saw how you were able to control your robot using the game pad controller. The game pad was actually plugged into your computer and was acting as an input device. Additionally, your computer is most likely connected to the internet using which you can access many web pages. It is also possible to acquire the content of any web page using the internet. Traditionally, in computer science people refer to this as a process of *input*. Using this view, getting sensory information from the robot is just a form of input. Given that we have at our disposal all of the input facilities provided by the computer, we can just as easily acquire input from any of the modalities and combine them with robot behaviors if we wish. Whether you consider this as *extra sensory perception* or not is a matter of opinion. Regardless, being able to get input from a diverse set of sources can make for some very interesting and useful computer and robot applications.

Game Pad Controllers¹

The game pad controller you used in Chapter 1 is a typical device that provides interaction facilities when playing computer games. These devices have been standardized enough that, just like a computer mouse or a keyboard, you can purchase one from a store and plug it into a USB port of your computer.

Myro provides some very useful input functions that can be used to get input from the game pad controller. Game pads come in all kinds of flavors and configurations with varying numbers of buttons, axes, and other devices on them. In the examples below, we will restrict ourselves to a basic game pad shown in the picture on previous page.



The basic game pad has eight buttons (numbered 1 through 8 in the picture) and an axis controller (see picture on right). The buttons can be pressed or released (on/off) which are represented by 1.0 (for on) and 0.0 (for off). The axis can be pressed in many different orientations represented by a pair of values (for the x-axis and y-axis) that range from -1.0 to 1.0 with [0.0, 0.0] representing no activity on the axis. Two Myro functions are provided to access the values of the buttons and the axis:

```
getGamepad (<device>)
getGamepadNow (<device>)
returns a vector of double values indicating
the status of the specified <device>.
<device> can be "axis" or "button".
```



¹ The game pad controller is not implemented in the current C++/Myro API.

The `getGamepad` function returns only after `<device>` has been used by the user. That is, it waits for the user to press or use that device and then returns the values associated with the device at that instant. `getGamepadNow` does not wait and simply returns the device status right away. Here are some examples:

```
cout << robot.getGamepadNow("axis") << endl;
```

```
{0.0, 0.0}
```

```
cout << robot.getGamepad("axis") << endl;
```

```
{0.0, -1.0}
```

```
cout << robot.getGamepadNow("button") << endl;
```

```
{0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0}
```

Game Pad's Axes



Both `getGamepad` and `getGamepadNow` return the same vector of values: axis values are returned as a `double` vector `{x-axis, y-axis}` (see picture on right for orientation) and the button values are returned as a `double` vector of 0.0's and 1.0's. The first value in the list is the status of button#1, followed by 2, 3, and so on. See picture above for button numbering.

Do This: Modify your game pad controller to try out the game pad commands above and observe the values. Here is another way to better understand the operation of the game pad and the game pad functions:

```
while (timeRemaining(30)) {
    cout << robot.getGamepad("button") << endl;
}
```

Try out different button combinations. What happens when you press more than one button? Repeat the above for axis control and observe the values returned (keep the axes diagram handy for orientation purposes).

The game pad controller can be used for all kinds of interactive purposes, especially for robot control as well as in writing computer games (see Chapter X). Let us write a simple game pad based robot controller. Enter the program below and run it.

```
#include "Myro.h"
int main() {
    // A simple game pad based robot controller
    connect();
    while (timeRemaining(30)) {
        vector<double> XY = robot.getGamePadNow("axis");
        robot.motors(XY[0], XY[1]);
    }
    disconnect();
}
```

The program above will run for 30 seconds. In that time it will repeatedly sample the values of the axis controller and since those values are in the range -1.0..1.0, it uses them to drive the motors. When you run the above program observe how the robot moves in response to pressing various parts of the axis. Do the motions of the robot correspond to the directions shown in the game pad picture on previous page? Try changing the command from `motors` to `move` (recall that `move` takes two values: translate and rotate). How does it behave with respect to the axes? Try changing the command to `robot.move(-XY[0], -XY[1])`. Observe the behavior.

As you can see from the simple example above, it is easy to combine input from a game pad to control your robot. Can you expand the program above to behave exactly like the `gamepad` controller function you used in Chapter 1? (See Exercise 5).

The World Wide Web and file I/O

If your computer is connected to the internet, you can also save a web page to disk and use it as input to your program. Web pages are written using markup languages like HTML and so when you save the content of a web page you will get the content with the markups included. In this section we will show

you how to access the content of a simple web page and print it out. Later we will see how you could use the information contained in it to do further processing.

Go to a web browser and take a look at the web page:

`http://www.fi.edu/weather/data/jan07.txt`

This web page is hosted by the Franklin Institute of Philadelphia and contains recorded daily weather data for Philadelphia for January 2007. You can navigate from the above address to other pages on the site to look at daily weather data for other dates (the data goes back to 1872!). You will see something like this:

```
January 2007
Day   Max   Min   Liquid   Snow   Depth
1     57   44   1.8    0     0
2     49   40   0      0     0
3     52   35   0      0     0
...   ...   ...   ...     ...   ...
31    31   22   0      0     0
#days 31
Sum   1414 1005 4.18  1.80  1.10
```

Use your browser to save this file to your computer's disk in the same directory where you put your C++ programs. We will use it learn how to do file input/output in C++.

By now you are familiar with using `cin` and `cout` for input/output to the console (keyboard and monitor). They are both examples of *streams* (the *console-input stream* and the *console-output stream*), and input/output to files is also handled by connecting the files to streams. The required definitions are in a standard library called `<fstream>`, which you must `#include` in your program. For example, to read the file `jan07.txt` you would put this declaration in `main`:

```
ifstream in ("jan07.txt");
```

This defines an *input file stream* called `in`, which is connected to the file `jan07.txt`. You can read from this file just as from `cin`; for example, the following will read and print the first word of the file:

```
string word;
in >> word;
cout << word << endl;
```

```
January
```

The operation `in >>`, like `cin >>`, reads “meaningful units” (such as integers, floating point numbers, and words), so it stops when it encounters white space, and skips the white space to get to the next unit. To avoid this, you can use `get()`, which reads a single character from the stream. For example, the following code prints the contents of the file:

```
char ch; // declare a variable to hold a single character
while (!in.eof()) { // continue while not at end of file
    in.get(ch);
    cout << ch;
}
```

Notice that we used the member function `eof()`, which tells us whether we have reached the end of the file. Can you write a program to compute the average of the maximum temperatures recorded in a file such as `jan07.txt`? How about a program to compare two files to see if they are identical?

Writing to an output file is very similar. Consider:

```
ofstream out ("output-file.txt");
...
out << "Here's some output: " << 2+2 << endl;
...
out.close(); // close the file when you are done with it
```

Between the file declaration and the close command, you can use `out <<` just the way you are used to using `cout <<`. C++ provides many more facilities

for doing input and output, including careful control of formatting; you will learn some of them in Chapter 7.

A little more about C++ functions

Before we move on, it would be good to take a little refresher on writing C++ commands/functions. In Chapter 2 we learned that the basic syntax for defining new commands/functions is:

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

The Myro module provides you with several useful functions (`forward`, `turnRight`, etc.) that enable easy control of your robot's basic behaviors. Additionally, using the syntax above, you learned to combine these basic behaviors into more complex behaviors (like `wiggle`, `yoyo`, etc.). By using parameters you can further customize the behavior of functions by providing different values for the parameters (for example, `forward(1.0)` will move the robot faster than `forward(0.5)`). You should also note a crucial difference between the movement commands like `forward`, `turnLeft`, and commands that provide sensory data like `getLight` or `getStall`, etc. The sensory commands always *return* a value whenever they are issued. That is:

```
cout << robot.getLight("left") << endl;
```

```
221
```

```
cout << robot.getStall() << endl;
```

```
0
```

Commands that return a value when they are invoked are called *functions* since they actually behave much like mathematical functions. None of the movement commands return any value, but they are useful in other ways. For instance, they make the robot do something. In any program you typically

need both kinds of functions: those that do something but do not return anything as a result; and those that do something and return a value. You can already see the utility of having these two kinds of functions from the examples you have seen so far. Functions are an integral and critical part of any program and part of learning to be a good programmer is to learn to recognize abstractions that can then be packaged into individual functions (like `drawPolygon`, or `degreeTurn`), which can be used over and over again.

Writing functions that return values

C++ provides a `return`-statement that you can use inside a function to return the results of a function. For example:

```
int triple(int x) {  
    // Returns x*3  
    return x * 3;  
}
```

The word `int` that begins the function definition tells us the type of value returned by `triple`. (As you have seen, definitions of commands, which do not return a value, begin with the word `void`.) The function above can be used just like the ones you have been using:

```
cout << triple(3) << endl;
```

```
9
```

```
cout << triple(5000) << endl;
```

```
15000
```

The general form of a `return`-statement is:

```
return <expression>;
```

That is, the function in which this statement is encountered will return the value of the `<expression>`. Thus, in the example above, the `return`-statement returns the value of the expression $3*x$, as shown in the example

invocations. By giving different values for the parameter `x`, the function simply triples it. This is the idea we used in normalizing light sensor values in the example earlier where we defined the function `normalize` to take in light sensor values and normalize them to the range 0.0..1.0 relative to the observed ambient light values:

```
// This function normalizes light sensor values to 0.0..1.0
double normalize (int v) {
    if (v > Ambient) {
        v = Ambient;
    }

    return 1.0 - v/Ambient;
}
```

In defining the function above, we are also using a new C++ statement: the `if`-statement. This statement enables simple decision making inside computer programs. The simplest form of the `if`-statement has the following structure:

```
if (<CONDITION>) {
    <do something>
    <do something>
    ...
}
```

That is, if the condition specified by `<CONDITION>` is `true` then whatever is specified in the body of the `if`-statement is carried out. In case the `<condition>` is `false`, all the statements under the `if`-command are skipped over.

Functions can have zero or more `return`-statements. Some of the functions you have written, like `wiggle` do not have any. Technically, when a non-void function does not have any `return`-statement that returns a value, the function returns an *undefined value* (that is, an unpredictable and possibly illegal value). This is bad programming and a cause of unpredictable program behavior.

Functions, as you have seen, can be used to package useful computations and can be used over and over again in many situations. Before we conclude this section, let us give you another example of a function. Recall from Chapter 4 the robot behavior that enables the robot to go forward until it hits a wall. One of the program fragments we used to specify this behavior is shown below:

```
while (! getStall()) {
    robot.forward(1);
}
robot.stop();
```

In the above example, we are using the value returned by `getStall` to help us make the decision to continue going forward or stopping. We were fortunate here that the value returned is directly usable in our decision making. Sometimes, you have to do little interpretation of sensor values to figure out what exactly the robot is sensing. You will see that in the case of light sensors. Even though the above statements are easy to read, we can make them even better, by writing a function called `stuck()` as follows:

```
bool stuck() {
    /* Is the robot stalled?
       Returns true if it is and false otherwise. */

    return robot.getStall() == 1;
}
```

The function above is simple enough, since `getStall` already gives us a usable Boolean value (0/false or 1/true). But now if we were to use `stuck` to write the robot behavior, it would read:

```
while (! stuck()) {
    robot.forward(1);
}
robot.stop();
```

As you can see, it reads much better. Programming is a lot like writing in this sense. As in writing, there are several ways of expressing an idea in words.

Some are better and more readable than others. Some are downright poetic. Similarly, in programming, expressing something in a program can be done in many ways, some better and more readable than others. Programming is not all about functionality, there *can* be poetry in the way you write a program.

Summary

In this chapter you have learned all about obtaining sensory data from the robot's perceptual system to do visual sensing (pictures), light sensing, and proximity sensing. The Scribbler provides a rich set of sensors that can be used to design interesting robot behaviors. You also learned that sensing is equivalent to the basic input operation in a computer. You also learned how to get input from a game pad, the World Wide Web, and from data files. Programs can be written to make creative use of the input modalities available to define robot behaviors, computer games, and even processing data. In the rest of the book you will learn how to write programs that make use of these input modalities in many different ways.

Myro Review

```
robot.getBright(<POSITION>)
```

Returns the current value in the <POSITION> light sensor. <POSITION> can either be one of "left", "center", "right" or one of the numbers 0, 1, 2.

```
robot.getGamepad(<device>)  
robot.getGamepadNow(<device>)
```

Returns a double vector indicating the status of the specified <device>. <device> can be "axis" or "button". The `getGamepad` function waits for an event before returning values. `getGamepadNow` immediately returns the current status of the device.

```
robot.getIR(<POSITION>)
```

Returns the current value in the <POSITION> IR sensor. <POSITION> can either be one of "left" or "right" or one of the numbers 0, 1.

```
robot.getLight (<POSITION>)
```

Returns the current value in the <POSITION> light sensor. <POSITION> can either be one of "left", "center", "right" or one of the numbers 0, 1, 2. The positions 0, 1, and 2 correspond to the left, center, and right sensors.

```
robot.getObstacle (<POSITION>)
```

Returns the current value in the <POSITION> IR sensor. <POSITION> can either be one of "left", "center", "right" or one of the numbers 0, 1, or 2.

```
savePicture (<picture>, <file>);  
savePicture (<picture vector>, <file>);
```

Saves the picture in the file specified. The extension of the file should be ".gif" or ".jpg". If the first parameter is a vector of pictures, the file name should have an extension ".gif" and an animated GIF file is created using the pictures provided.

```
robot.senses ();
```

Displays Scribbler's sensor values in a window. The display is updated every second.

```
show (<picture>);
```

Displays the picture in a window. You can click the left mouse anywhere in the window to display the (x, y) and (r, g, b) values of the point in the window's status bar.

```
robot.takePicture ()  
robot.takePicture ("color")  
robot.takePicture ("gray")
```

Takes a picture and returns a `picture` object. When no parameters are specified, the picture is in color.

```
cout << <vector>  
cout << <list>
```

Print vector <vector> or list <list>.

C++ review

```
if (<CONDITION>) {  
    <statement-1>  
    ...  
    <statement-N>  
}
```

If the condition evaluates to `true`, all the statements are performed. Otherwise, all the statements are skipped.

```
return <expression>;
```

Can be used inside any function to return the result of the function.

```
<TYPE> <NAME> [] = { <VALUES> };
```

Declares a C++ array called `<NAME>` with elements of type `<TYPE>`. The array is initialized to the specified `<VALUES>`.

```
char <NAME>;
```

Declares character variable `<NAME>` (able to hold one character).

Vectors and Lists:

```
#include <vector>  
#include <list>
```

Includes definitions for C++ vectors or lists, which are both examples sequence containers.

```
vector< <TYPE> > <NAME>;
```

Defines an initially empty vector called `<NAME>` with elements of type `<TYPE>`.

```
list< <TYPE> > <NAME>;
```

Defines an initially empty list called `<NAME>` with elements of type `<TYPE>`.

```
vector< <TYPE> > <NAME> (<ARRAY START>, <ARRAY END>);  
list< <TYPE> > <NAME> (<ARRAY START>, <ARRAY END>);
```

Defines a vector or list called `<NAME>` with elements of type `<TYPE>` initialized with values from location `<ARRAY START>` up to, but not including, `<ARRAY END>`. These may be of the form `<ARRAY NAME> + <CONSTANT>`.

```
<seq>.size()
```

Returns the current size (length) of the vector, list, or string `<seq>`.

```
<vector>[i]
```

```
<string>[i]
```

Returns the `i`th element in the `<vector>` or `<string>`. Indexing starts from 0. (Strings are a lot like vectors of characters.)

```
<seq>.push_back(<value>);
```

Appends the `<value>` at the back (end) of vector, list, or string `<seq>`.

```
<list>.push_front(<value>);
```

Appends the `<value>` at the front (beginning) of `<list>`.

```
<list>.sort();
```

Sorts the `<list>` in ascending order.

```
<list>.reverse();
```

Reverses the elements in the list.

```
<seq>.begin()
```

Returns an iterator referring to the first element of the vector, list, or string

```
<seq>.
```

```
<seq>.end()
```

Returns an iterator referring the last element of the vector, list, or string

```
<seq>.
```

```
vector< <TYPE> >::const_iterator <NAME>;
```

```
vector< <TYPE> >::iterator <NAME>;
```

```
list< <TYPE> >::const_iterator <NAME>;
```

```
list< <TYPE> >::iterator <NAME>;
```

```
string::const_iterator <NAME>;
```

```
string::iterator <NAME>;
```

Declares an iterator called `<NAME>`, which can refer to locations in a vector/list with elements of type `<TYPE>` or in a string. A `const` (constant) iterator does not allow the elements of the vector or list to be modified, whereas a non-`const` iterator does.

```
<iterator>++
```

Increments `<iterator>` to refer to the next element of a vector, list, or string.

```
* <iterator>
```

Returns the value stored in the vector, list, or string element referred to by `<iterator>`.

```
<seq>.insert( <it1>, <it2-begin>, <it2-end> );
```

Inserts elements from one vector, list, or string into the vector, list, or string `<seq>`. The new elements are inserted at a location specified by iterator `<it1>`. For example, use `<seq>.begin()` to insert at the beginning of `<seq>` or use `<seq>.end()` to insert at its end. The elements to be inserted are specified by iterators `<it2-begin>` and `<it2-end>`. For example, to insert all the elements from vector, list, or string `<seq2>`, use `<seq2>.begin()` and `<seq2>.end()`.

```
for ( <it> = <seq>.begin(); <it> != <seq>.end(); <it>++) {  
    ... STATEMENTS using <it> or *<it> ...  
}
```

Executes the loop body with iterator `<it>` referring to the elements of vector or list `<seq>` from its beginning up to its end.

Streams:

```
#include <fstream>
```

Includes definitions for C++ streams for file input/output.

```
ifstream <name> (<string>);
```

Declares `<name>` to be an input file stream connected to file name `<string>`.

```
ofstream <name> (<string>);
```

Declares <name> to be an output file stream connected to file name <string>.

```
<ifstream> >> <var>;
```

Reads input value from stream <ifstream> and stores it in variable <var>.

```
<ofstream> << <expression>
```

Outputs values of <expression> to output stream <ofstream>.

```
<stream>.close();
```

Closes (input or output) file stream <stream>, thus completing operation on it. You should close files when you are done with them.

```
<ifstream>.get (<chvar>);
```

Puts the next character from input stream <ifstream> into character variable <chvar>.

```
<ifstream>.eof()
```

Returns `true` if input stream <ifstream> is at the end of the file, and `false` if not.

Exercises

1. The numbers assigned to the variable `FamousNumbers` in this chapter all have names. Can you find them? What other famous numbers do you know?
2. Besides text, the `speak` command can also vocalize numbers. Try `speak(42)` and also `speak(3.1419)`. Try some really large whole numbers, like `speak(4537130980)`. How is it vocalized? Can you find out the limits of numerical vocalization?
3. The Fluke camera returns pictures that are 256x192 (= 49,152) pixels. Typical digital cameras are often characterized by the number of pixels they use for images. For example, 8 megapixels. What is a megapixel? Do some web search to find out the answer.
4. All images taken and saved using the Fluke camera can also be displayed in a web page. In your favorite browser, use the Open File option (under the File menu) and navigate to a saved picture from the Scribbler. Select it and view it in the browser window. Try it with an animated GIF image file.
5. Modify the game pad input program from this chapter to make the axis controller behave so that when the top of the axis is pressed, the robot goes forward and when the bottom of the axis is pressed, it goes backward.
6. Write a program that reads in the names of two files, and then compares the two files character-by-character to see if they are identical.
7. Write a program that reads in a monthly weather report, such as `jan07.txt`, and computes the average maximum temperature and average minimum temperature for the month.

