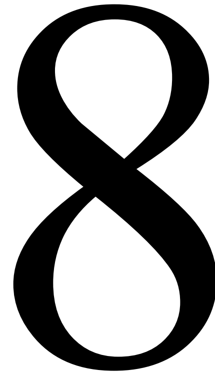


Sights & Sounds

Don't make music for some vast, unseen audience or market or ratings share or even for something as tangible as money. Though it's crucial to make a living, that shouldn't be your inspiration. Do it for yourself.

-Billy Joel

Opposite page: Be My Valentine
Fractal Art by Sven Geier (www.sgeier.net)

We mentioned earlier that the notion of computation these days extends far beyond simple numerical calculations. Writing robot control programs is computation, as is making world population projections. Using devices like iPods you are able to enjoy music, videos, and radio and television shows. Manipulating sounds and images is also the realm of computation and in this chapter we will introduce you to these. You have already seen how, using your robot, you can take pictures of various scenes. You can also take similar images from your digital camera. Using basic computational techniques you have learned so far you will see, in this chapter, how you can do computation on shapes and sound. You will learn to create images using computation. The images you create can be used for visualizing data or even for aesthetic purposes to explore your creative side. We will also present some fundamentals of sound and music and show you how you can also do similar forms of computation using music.

Sights: Drawing[1]

If you have used a computer for any amount of time you must have used some kind of a drawing application. Using a typical drawing application you can draw various shapes, color them etc. You can also generate drawings using drawing commands provided in the Myro library module. In order to draw anything you first need a place to draw it: a canvas or a window. You can create such a window using the command:

```
window myCanvas = GraphWin();
```

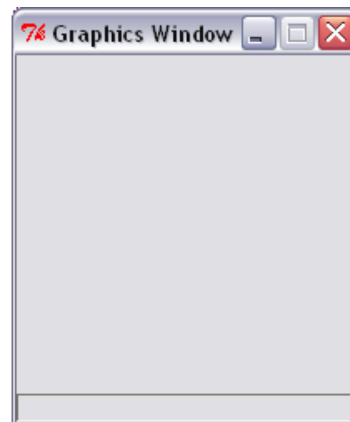
If you execute the command above in a C++ program, you will immediately see a small gray window pop up (see picture on right). This window will be serving as our canvas for creating drawings. By default, the window created by the `GraphWin` command is 200 pixels high and 200 pixels wide and its name is `"Graphics Window"`. Not a very inspiring way to start, but the

[1] Graphics functions and objects are not implemented in the current C++/Myro API. The following description is subject to change.

`GraphWin` command has some other variations as well. First, in order to make the window go away, you can use the command:

```
myCanvas.close();
```

A Graphics window



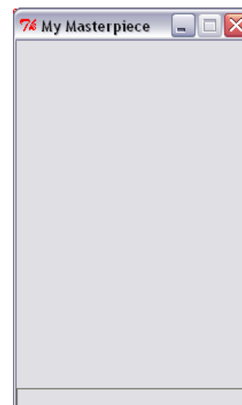
To create a graphics window of any size and a name that you specify, you can use the command below:

```
window myCanvas = GraphWin
  ("My Masterpiece", 200, 300);
```

The command above creates a window named `"My Masterpiece"` that will be 200 pixels wide and 300 pixels tall (see picture on right). You can change the background color of a graphics window as shown below:

```
myCanvas.setBackground("white");
```

My Masterpiece 200x300



You can name any of a number of colors in the command above, ranging from mundane ones like `"red"`, `"blue"`, `"gray"`, `"yellow"`, to more exotic colors ranging from `"AntiqueWhite"` to `"LavenderBlush"` to `"WhiteSmoke"`. Colors can be created in many ways as we will see below. Several thousand color names have been pre-assigned (Google: color names list) that can be used in the command above.

Now that you know how to create a canvas (and make it go away) and even set a background color, it is time to look at what we can draw in it. You can create and draw all kinds of geometrical objects: points, lines, circles, rectangle, and

even text and images. Depending on the type of object you wish to draw, you have to first *create* it and then draw it. Before you can do this though, you should also know the coordinate system of the graphics window.

In a graphics window with width, W and height H (i.e WxH pixels) the pixel (0, 0) is at the top left corner and the pixel (199, 299) will be at the bottom right corner. That is, x-coordinates increase as you go right and y-coordinates increase as you go down.

The simplest object that you can create is a *point*. This is done as follows:

```
Point p = Point(100, 50);
```

That is, `p` is an object that is a `Point` whose x-coordinate is at 100 and y-coordinate is at 50. This only creates a `Point` object. In order to draw it, you have to issue the command:

```
p.draw(myCanvas);
```

The syntax of the above command may seem a little strange at first. You saw it briefly when we presented lists in Chapter 5 but we didn't dwell on it. Certainly it is different from what you have seen so far. But if you think about the objects you have seen so far: numbers, strings, etc. Most of them have standard operations defined on them (like +, *, /, etc.). But when you think about geometrical objects, there is no standard notation. Programming languages like C++ provide facilities for modeling any kind of object and the syntax we are using here is standard syntax that can be applied to all kinds of objects. The general form of commands issued on objects is:

```
<object>.<function>(<parameters>)
```

Thus, in the example above, `<object>` is the name `p` which was earlier defined to be a `Point` object, `<function>` is draw, and `<parameters>` is `myCanvas`. The `draw` function requires the graphics window as the parameter. That is, you are asking the point represented by `p` to be drawn in the window specified as its parameter. The `Point` objects have other functions available:

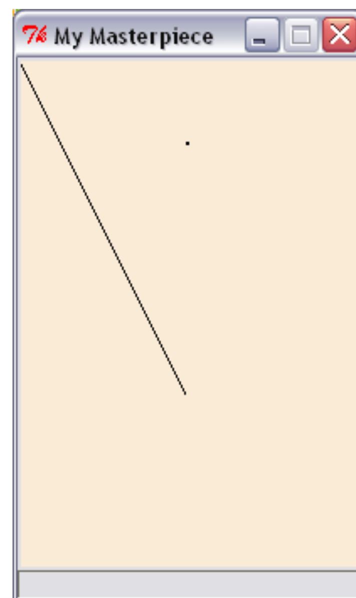
```
cout << p.getX() << endl;
```

```
100
```

```
cout << p.getY() << endl;
```

```
50
```

That is, given a `Point` object, you can get its x- and y-coordinates. Objects are created using their *constructors* like the `Point(x, y)` constructor above. We will use lots of constructors in this section for creating the graphics objects. A line object can be created similar to point objects. A line requires the two end points to be specified. Thus a line from (0, 0) to (100, 200) can be created as:




```
Line L = Line(Point(0,0),
  Point(100,200));
```

And you can draw the line using the same draw command as above:

```
L.draw(myCanvas);
```

The picture on the right shows the two objects we have created and drawn so far. As for `Point`, you can get the values of a line's end points:

```
Point start = L.getP1();
cout << start.getX << endl;
```

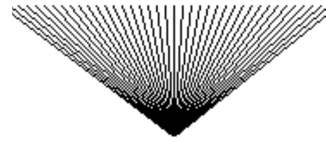
```
0
```

```
Point end = L.getP2();
cout << end.getY() << endl;
```

```
200
```

Here is a small C++ loop that can be used to create and draw several lines:

```
for (int n = 0; n < 200; n = n+5) {
    Line L=Line(Point(n,25),
      Point(100,100));
    L.draw(myCanvas);
}
```



In the loop above (the results are shown on the right), the value of `n` starts at 0 and increases by 5 after each iteration all the way up to but not including 200 (i.e. 195). For each value of `n` a new `Line` object is created with starting co-ordinates `(n, 25)` and end point at `(100, 100)`.

Do This: Try out all the commands introduced so far. Then observe the effects produced by the loop above. Change the increment 5 in the loop above to different values (1, 3, etc.) and observe the effect. Next, try out the following loop:

```
for (int n = 0; n < 200; n = n+5) {
    Line L = Line(Point(n, 25), Point(100, 100));
    L.draw(myCanvas);
    wait(0.3);
    L.undraw();
}
```

The `undraw` function does exactly as the name implies. In the loop above, for each value that `n` takes, a line is created (as above), drawn, and then, after a wait of 0.3 seconds, it is erased. Again, modify the value of the increment and observe the effect. Try also changing the amount of time in the `wait` command.

You can also draw several geometrical shapes: circles, rectangles, ovals, and polygons. To draw a circle, (or any geometrical shape), you first create it and then draw it:

```
Circle C = Circle(centerPoint, radius);
c.draw(myCanvas);
```

`centerPoint` is a `Point` object and radius is specified in pixels. Thus, to draw a circle centered at (100, 150) with a radius of 30, you would do the following commands:

```
Circle C = Circle(Point(100, 150), 30);
c.draw(myCanvas);
```

Rectangles and ovals are drawn similarly (see details at the end of the chapter). All geometrical objects have many functions in common. For example, you can get the center point of a circle, a rectangle, or an oval by using the command:

```
centerPoint = C.getCenter();
```

By default, all objects are drawn in black. There are several ways to modify or specify colors for objects. For each object you can specify a color for its outline as well as a color to fill it with. For example, to draw a circle centered at (100, 150), radius 30, and outline color red, and fill color yellow:

```
Circle C = Circle(Point(100, 150), 30);
C.draw(myCanvas);
C.setOutline("red");
C.setFill("yellow");
```

By the way, `setFill` and `setOutline` have the same effect on `Point` and `Line` objects (since there is no place to fill any color). Also, the line or the outline drawn is always 1 pixel thick. You can change the thickness by using the command `setWidth(<pixels>)`:

```
C.setWidth(5);
```

The command above changes the width of the circle’s outline to 5 pixels.

Do This: Try out all the commands introduced here. Also, look at the end of the chapter for details on drawing other shapes.

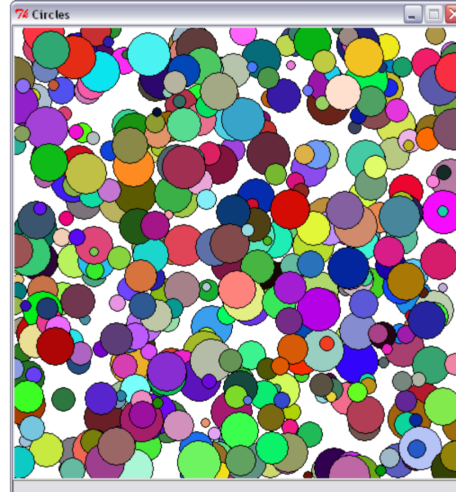
Earlier, we mentioned that several colors have been assigned names that can be used to select colors. You can also create colors of your own choosing by specifying their red, green, and blue values. In Chapter 5 we mentioned that each color is made up of three values: RGB or red, green and blue color values. Each of the these values can be in the range 0..255 and is called a 24-bit color value. In this way of specifying colors, the color with values (255, 255, 255) (that is red = 255, green = 255, and blue = 255) is white; (255, 0, 0) is pure red, (0, 255, 0), is pure blue, (0, 0, 0) is black, (255, 175, 175) is pink, etc. You can have as many as 256x256x256 colors (i.e. over 16 million colors!). Given specific RGB values, you can create a new color by using the command, color_rgb:

```
color_rgb myColor = color_rgb(255, 175, 175);
```

Do This: The program below draws several circles of random sizes with random colors. Try it out and see its outcome. A sample output screen is show on the right. Modify the program to input a number for the number of circles to be drawn. In Chapter 4, Exercise 8, you programmed `randrange(m,n)` to returs a random number in range `[m..n-1]`.

```
// Program to draw a bunch of # random colored circles
#include <cstlib>
#include "Myro.h"
using namespace std;

void makeCircle(double x, double y, double r) {
    // creates a Circle centered at point (x, y) of radius r
    return Circle(Point(x, y), r);
}

color_rgb makeColor() {
    // creates a new color using random RGB values
    int red = randrange(0, 256);
    int green = randrange(0, 256);
    int blue = randrange(0, 256);
```

```
    return color_rgb(red, green,blue);
}

int main() {
    // Create and display a
    // graphics window
    int width = 500;
    int height = 500;
    window myCanvas = GraphWin
      ("Circles",width,height);
    myCanvas.
      setBackground("white");

    // draw a bunch of random
    // circles with random
    // colors.
    int N = 500;
    for (int i = 0;
      I < 100; i++) {
        // pick random center
        // point and radius
        // in the window
        int x = randrange(0,width);
        int y = randrange(0,height);
        int r = randrange(5, 25);
        int c = makeCircle(x, y, r);
        // select a random color
        c.setFill(makeColor());

        c.draw(myCanvas);
    }
    return 0;
}
```



Notice our use of functions to organize the program. From a design perspective, the two functions `makeCircle` and `makeColor` are written differently. This is just for illustration purposes. You could, for instance, define `makeCircle` just like `makeColor` so it doesn't take any parameters and generates the values of x, y, and radius as follows:

```
Circle makeCircle() {
    // creates a Circle centered at point (x, y) of radius r
    int x = randrange(0,width);
    int y = randrange(0,height);
    int r = randrange(5, 25);

    return Circle(Point(x, y), r);
}
```

Unfortunately, as simple as this change seems, the function is not going to work. In order to generate the values of x, and y it needs to know the width and height of the graphics window. But width and height are defined in the function main and are not available or accessible in the function above. This is an issue of *scope* of names in a C++ program: what is the scope of accessibility of a name in a program?

C++ defines the scope of a name in a program *textually* or *lexically*. That is, any name is visible in the text of the program/function *after* it has been defined. Note that the notion of *after* is a textual notion. Moreover, C++ restricts the accessibility of a name to the text of the function in which it is defined. That is, the names width and height are defined inside the function main and hence they are not visible anywhere outside of main. Similarly, the variables `red`, `green`, and `blue` are considered local to the definition of `makeColor` and are not accessible outside of the function, `makeColor`.

So how can `makeCircle`, if you decided it would generate the `x` and `y` values relative to the window size, get access to the width and height of the window? There are two solutions to this. First, you can pass them as parameters. In that case, the definition of `makeCircle` will be:

```
Circle makeCircle(int w, int h) {
    // creates a Circle centered at point (x, y) of radius r
    // such that (x, y) lies within width, w and height, h
    int x = randrange(0,w);
    int y = randrange(0,h);
    int r = randrange(5, 25);
    return Circle(Point(x, y), r);
}
```

Then the way you would use the above function in the main program would be using the command:

```
Circle C = makeCircle(width, height);
```

That is, you pass the values of `width` and `height` to `makeCircle` as parameters. The other way to define `makeCircle` would be exactly as shown in the first instance:

```
Circle makeCircle() {
    // creates a Circle centered at point (x, y) of radius r
    int x = randrange(0,width);
    int y = randrange(0,height);
    int r = randrange(5, 25);

    return Circle(Point(x, y), r);
}
```

However, you would move the definitions of `width` and `height` outside and before the definitions of all the functions:

```
#include <cstlib>
#include "Myro.h"
using namespace std;

int width = 500;
int height = 500;

Circle makeCircle() {
    …

color_rgb makeColor() {
   …

int main() {
   …
```

Since the variables are defined outside of any function and before the definitions of the functions that use them, you can access their values. You

may be wondering at this point, which version is better? Or even, why bother? The first version was just as good. The answer to these questions is similar in a way to writing a paragraph in an essay. You can write a paragraph in many ways. Some versions will be more preferable than others. In programming, the rule of thumb one uses when it comes to the scope of names is: ensure that only the parts of the program that are supposed to have access to a name are allowed access. This is similar to the reason you would not share your password with anyone, or your bank card code, etc. In our second solution, we made the names `width` and `height` *globally* visible to the entire program that follows. This implies that even `makeColor` can have access to them whether it makes it needs it or not.

You may want to argue at this point: what difference does it make if you make those variables visible to `makeColor` as long as you take care not to use them in that function? You are correct, it doesn't. But it puts an extra responsibility on your part to ensure that you will not do so. But what is the guarantee that someone who is modifying your program chooses to?

We used the simple program here to illustrate simple yet potentially hazardous decisions that dot the landscape of programming like land mines. Programs can crash if some of these names are *mishandled* in a program. Worse still, programs do not crash but lead to incorrect results. However, at the level of deciding which variables to make local and which ones to make global, the decisions are very simple and one just needs to exercise safe programming practices. We will conclude this section of graphics with some examples of creating animations.

Any object drawn in the graphics window can be moved using the command `move(dx, dy)`. For example, you move the circle 10 pixels to the right and 5 pixels down you can use the command:

```
C.move(10, 5);
```

Do This: Let us write a program that moves a circle about (randomly) in the graphics window. First, enter this following program and try it out.

```
// Moving circle; Animate a circle...
#include <cstlib>
#include "Myro.h"
using namespace std;

int main() {
    // create and draw the graphics window
    window w = GraphWin("Moving Circle", 500, 500);
    w.setBackground("white");

    // Create a red circle
    Circle c = Circle(Point(250, 250), 50);
    c.setFill("red");
    c.draw(w);

    // Do a simple animation for 200 steps
    for (int i = 0; I < 200; i++) {
        c.move(randrange(-4, 5), randrange(-4, 5));
        wait(0.2);
    }
    return 0;
}
```

Notice, in the above program, that we are moving the circle around randomly in the x- and y-directions. Try changing the range of movements and observe the behavior. Try changing the values so that the circle moves only in the horizontal direction or only in the vertical direction. Also notice that we had to *slow down* the animation by inserting the wait command after every move. Comment the wait command and see what happens. It may appear that nothing did happen but in fact the 200 moves went so quickly that your eyes couldn't even register a single move! Using this as a basis, we can now write a more interesting program. Look at the program below:

```
// Moving circle; Animate a circle...
#include <cstlib>
#include "Myro.h"
using namespace std;
```

```
int main() {
    // create and draw the graphics window
    int winWidth = 500, winHeight = 500;
    window w = GraphWin("Bouncing Circle",
      winWidth, winHeight);
    w.setBackground("white");

    // Create a red circle
    int radius = 25;
    Circle c = Circle(Point(53, 250), radius);
    c.setFill("red");
    c.draw(w);

    // Animate it
    int dx = 3, dy = 3;
    while (timeRemaining(15)) {
        // move the circle
        c.move(dx, dy);

        // make sure it is within bounds
        Point center = c.getCenter();
        int cx = center.getX();
        int cy = center.getY();

        if ((cx+radius >= winWidth) || (cx-radius <= 0))
            dx = -dx;

        if ((cy+radius >= winHeight) || (cy-radius <= 0))
            dy = -dy;

        wait(0.01);
    }
    return 0;
}
```

For 15 seconds, you will see a red circle bouncing around the window. Study the program above to see how we keep the circle inside the window at all times and how the direction of the ball bounce is being changed. Each time you change the direction, make the computer beep:

```
computer.beep(0.005, 440);
```

If you are excited about the possibility of animating a circle, imagine what you can do if you have many circles and other shapes animated. Also, plug in the game pad controller and see if you can control the circle (or any other object) using the game pad controls. This is very similar to controlling your robot. Design an interactive computer game that takes advantage of this new input modality. You can also design multi-user games since you can connect multiple game pad controllers to your computer. See the Reference documentation for details on how to get input from several game pad controllers.

Drawing Text & Images

Like shapes, you can also place text in a graphics window. The idea is the same. You first create the text using the command:

```
Text myText = Text(<anchor point>, <string>);
```

and then draw it. You can specify the type face, size, and style of the text. We will not detail these here. They are summarized in the reference at the end of the text. When we get an opportunity, we will use these features below in other examples.

Images in this system are treated just like any other objects. You can create an image using the `Image` command:

```
Image myPhoto = Image(<centerPoint>, <file name>);
```

You have to have an already prepared image in one of the common image formats (like JPEG, GIF, etc.) and stored in a file (`<file name>`). Once the image object is created, it can be drawn, undrawn, or moved just like other shapes.

What you do with all this new found functionality depends on your creativity. You can use graphics to visualize some data: plotting the growth of world population, for example; or create some art, an interactive game, or even an

animated story. You can combine your robot, the game pad controller, or even sound to enrich the multi-media experience. The exercises at the end of the chapter present some ideas to explore further. Below, we delve into sounds.

Sound

As we have seen, you can have your robot make *beeps* by calling the `beep()` function, like so:

```
robot.beep(1, 440);
```

This command instructs the robot to play a tone at 440 Hz for 1 second. Let us first try and analyze what is in the *440 Hz* tone. First, the letters *Hz* are an abbreviation for *Hertz*. The name itself comes from a German physicist, Heinrich Rudolph Hertz who did pioneering work in the production of electromagnetic waves in the late 19th century. Today, we use **Hertz** (or *Hz*) as a unit for specifying frequencies.

$$1 Hertz = 1 cycle / second$$

The most common use of frequencies these days is in specifying the clock speeds of computer CPU's. For example, a typical PC today runs at clock speeds of a few GigaHertz (or GHz).

$$1 GigaHertz = 10^9 cycles / second$$

Frequencies are related to periodic (or repetitive) motions or vibrations. The time it takes for a motion or vibration to repeat is called its *time period*. Frequency and time period are inversely related. That is the number of cycles or repetitions in a second is called the frequency. Thus 1 Hertz refers to any motion or vibration that repeats every 1 second. In the case of computer clock frequencies then, a computer running at 4 Gigahertz is repeating 4 billion times a second! Other examples of periodic motions include: the earth's rotation on its axis (1 cycle every 24 * 60 * 60 = 86400 seconds or at a frequency of 0.00001157 cycles/second), a typical audio CD spins 400 times a

second, a CD drive on your computer rated at 52x spins the CD at 52 * 400 = 20800 times per second, hummingbirds can flap their wings at frequencies ranging from 20-78 times/second (some can go even as high as 200!).

Sound is a periodic compression and refraction (or return to its original state) of air (for simplicity, let us assume that the medium is air). One Cycle of a sound comprises one compression and one refraction. Thus, producing a beep at 440 Hz represents 440 complete cycles of compression and refraction. Generally, a human ear is capable for hearing frequencies in the 20 Hz to 20000 Hz (or 20 Kilo Hertz) range. However the capability varies from person to person. Also, many electronic devices are not capable for producing frequencies in that entire range. 20-20KHz is considered hi-fidelity for stereo or home theater audio components. Let us first examine the range of audible sounds the Scribbler can produce. To make a sound out of the Scribbler, you have to give a frequency and the duration (in seconds) that the sound should be played. For example, to play a 440 Hz sound for 0.75 seconds:

```
robot.beep(0.75, 440);
```

The human ear is capable of distinguishing sounds that differ only by a few Hertz (as little as 1 Hz) however this ability varies from person to person. Try the commands:

```
robot.beep(1, 440);
robot.beep(1, 450);
```

Can you distinguish between the two tones? Sometimes it is helpful to place these in a loop so you can repeatedly hear the alternating tones to be able to distinguish between them. The next exercise can help you determine what frequencies you are able to distinguish.

Exercise: Using the example above, try to see how close you can get in distinguishing close frequencies. As suggested, you may want to play the tones alternating for about 5-10 seconds. Start with 440 Hz. Can you hear the difference between 440 and 441? 442? Etc. Once you have established your

range, try another frequency, say 800. Is the distance that you can distinguish the same?

Do This: You can program the Scribbler to create a siren by repeating two different tones (much like in the example above). You will have to experiment with different pairs of frequencies (they may be close together or far apart) to produce a realistic sounding siren. Write your program to play the siren for 15 seconds. The louder the better!

You can also have Myro make a beep directly out of your computer, rather than the robot, with the command:

```
computer.beep(1, 440);
```

Unfortunately, you can't really have the robot and computer play a duet. Why not? Try these commands:

```
robot.beep(1, 440);
computer.beep(1, 440);
robot.beep(1, 880);
computer.beep(1, 880);
robot.beep(1, 440);
computer.beep(1, 440);
```

What happens? Try your solutions to the above exercises by making the sounds on the computer instead of the Scribbler.

Musical Scales

In western music, a *scale* is divided into 12 notes (from 7 major notes: ABCDEFG). Further there are *octaves*. An octave in C comprises of the 12 notes shown below:

```
C C#/Db D D#/Eb E F F#/Gb G G#/Ab A A#/Bb B
```

C# (pronounced "C sharp") is the same tone as Db (pronounced "D flat").

Frequencies corresponding to a specific note, say C, are multiplied (or divided) by 2 to achieve the same note in a higher (or lower) octave. On a piano there are several octaves available on a spread of keys. What is the relationship between these two tones?

```
robot.beep(1, 440);
robot.beep(1, 880);
```

The second tone is exactly one octave the first. To raise a tone by an octave, you simply multiply the frequency by 2. Likewise, to make a tone an octave lower, you divide by 2. Notes indicating an octave can be denoted as follows:

```
C0 C1 C2 C3 C4 C5 C6 C7 C8
```

That is, C0 is the note for C in the lowest (or 0) octave. The fifth octave (numbered 4) is commonly referred to as a middle octave. Thus C4 is the C note in the middle octave. The frequency corresponding to C4 is 261.63 Hz. Try playing it on the Scribbler. Also try C5 (523.25) which is twice the frequency of C4 and C3 (130.815). In common tuning (*equal temperament*) the 12 notes are equidistant. Thus, if the frequency doubles every octave, each successive note is $2^{1/12}$ apart. That is, if C4 is 261.63 Hz, C# (or Db) will be:

$$C\#4/Db4 = 261.63 * 2^{1/12} = 277.18$$

Thus, we can compute all successive note frequencies:

$$D4 = 277.18 * 2^{1/12} = 293.66$$

$$D\#4/Eb = 293.66 * 2^{1/12} = 311.13$$

etc.

The lowest tone that the Scribbler can play is A0 and the highest tone is C8. A0 has a frequency of 27.5 Hz, and C8 has a frequency of 4186 Hz. That's quite a range! Can you hear the entire range?

```
robot.beep(1, 27.5);
robot.beep(1, 4186);
```

Exercise: Write a Scribbler program to play all the 12 notes in an octave using the above computation. You may assume in your program that C0 is 16.35 and then use that to compute all frequencies in a given octave (C4 is 16.35 * 2^4). Your program should input an octave (a number from 0 through 8), produce all the notes in that octave and also printout a frequency chart for each note in that octave.

Making Music

Playing songs by frequency is a bit of a pain. Myro contains a set of functions to make this task a bit more abstract.[2] A Myro song is a string of characters composed like so:

```
NOTE1 [NOTE2] WHOLEPART
```

where `[]` means optional. Each of these notes/chords is composed on its own line, or separated by semicolons where:

```
    NOTE1 is either a frequency or a NOTENAME
    NOTE2 is the same, and optional. Use for Chords.
    WHOLEPART is a number representing how much of
              a whole note to play.
```

`NOTENAMES` are case-insensitive strings. Here is an entire scale of `NOTENAMES`:

```
C C#/Db D D#/Eb E F F#/Gb G G#/Ab A A#/Bb B C
```

This is the default octave. It is also the 5th octave, which can also be written as:

```
C5 C#5/Db5 D5 D#5/Eb5 E5 F5 F#5/Gb5 G5 G#5/Ab5 A5 A#5/Bb5 B5
C6
```

[2] These functions are not implemented in the current C++/Myro interface.

The Myro Song Format replicates the keys on the piano, and so goes from A0 to C8. The middle octave on a keyboard is number 4, but we use 5 as the default octave. See http://en.wikipedia.org/wiki/Piano_key_frequencies for additional details. Here is a scale:

```
"C 1; C# 1; D 1; D# 1; E 1; F 1; F# 1; G 1; G# 1; A 1; A# 1; B
1; C 1;"
```

The scale, one octave lower, and played as a polka:

```
"C4 1; C#4 1/2; D4 1/2; D#4 1; E4 1/2; F4 1/2; F#4 1; G4 1/2;
G#4 1/2; A4 1; A#4 1/2; B4 1/2; C4 1;"
```

There are also a few other special note names, including `PAUSE`, `REST`, you can leave the octave number off of the default octave notes if you wish. Use "#" for sharp, and "b" for flat.

`WHOLEPART` can either be a decimal notation, or division. For example:

```
Ab2 .125
```

or

```
Ab2 1/8
```

represents the A flat in the second octave (two below middle).

As an example, try playing the following:

```
c 1
c .5
c .5
c 1
c .5
c .5
e 1
c .5
c .5
```

```
c 2
e 1
e .5
e .5
e 1
e .5
e .5
g 1
e .5
e .5
e 2
```

Do you recognize it??

You may leave blank lines, and comments should begin with a # sign. Lines can also be separated with a semicolon.

Using a song

For the following exercises, you will need to have an object to play the song. Now that you have a song, you probably will want to play it. If your song is in a file, you can read it:[3]

```
song s = readSong(filename);
```

and play it on the robot:

```
robot.playSong(s);
```

or on the computer:

```
computer.playSong(s);
```

You can also use `makeSong(text)` to make a song. For example:

```
song s = makeSong
```

[3] These functions are not implemented in the current C++/Myro interface.

```
("c 1; d 1; e 1; f 1; g 1; a 1; b 1; c7 1;");
```

and then play it as above.

If you want to make it play faster or slower, you could change all of the `WHOLENOTE` numbers. But, if we just want to change the tempo, there is an easier way:

```
robot.playSong(s, .75);
```

The second argument to `playSong` is the duration of a whole note in seconds. Standard tempo plays a whole note in about .5 seconds. Larger numbers will play slower, and smaller numbers play faster.

Summary

You can use the graphics window as a way of visualizing anything. In the graphics window you can draw all kinds of shapes: points, line, circles, rectangles, ovals, polygons, text, and even images. You can also animate these shapes as you please. What you can do with these basic drawing capabilities is limited only by your creativity and your programming ability. You can combine the sights you create with sounds and other interfaces, like the game pad controller, or even your robot. The multimedia functionalities introduced in this chapter can be used in all kinds of situations for creating interesting programs.

Myro Reference

Below, we summarize all of the graphics commands mentioned in this chapter. You are urged to review the reference manual for more graphics functionality.

```
GraphWin()
GraphWin(<title>, <width>, <height>)
```

Returns a graphics `window` object. It creates a graphics window with title,

<title> and dimensions <width> x <height>. If no parameters are specified, the window created is 200x200 pixels.

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

Closes the displayed graphics window <window>.

```
<window>.setBackground(<color>);
```

Sets the background color of the window to be the specified color. <color> can be a named color (Google: color names list), or a new color created using the color_rgb command (see below).

```
color_rgb(<red>, <green>, <blue>)
```

Creates a new RGB color object using the specified <red>, <green>, and <blue> values. The values can be in the range 0..255.

```
Point(<x>, <y>)
```

Creates a Point object at (<x>, <y>) location in the window.

```
<point>.getX()
<point>.getY()
```

Returns the x and y coordinates of the Point object <point>.

```
Line(<start point>, <end point>)
```

Creates a Line object starting at <start point> and ending at <end point>.

```
<line>.getP1()
<line>.getP2()
```

Returns an endpoint (Point object) of a Line object.

```
Circle(<center point>, <radius>)
```

Creates a Circle object centered at <center point> with radius <radius> pixels.

```
Rectangle(<point1>, <point2>)
```

Creates a Rectangle object with opposite corners located at <point1> and <point2>.

```
Oval(<point1>, <point2>)
```
Creates an `Oval` object in the bounding box defined by the corner points `<point1>` and `<point2>`.

```
Polygon(<point1>, <point2>, <point3>,…)
Polygon([<point1>, <point2>, …])
```
Creates a `Polygon` object with the given points as vertices.

```
Text(<anchor point>, <string>)
```
Creates a `Text` object anchored (bottom-left corner of text) at `<anchor point>`. The text itself is defined by `<string>`.

```
Image(<centerPoint>, <file name>)
```
Creates an `Image` centered at `<center point>` from the image file `<file name>`. The image can be in GIF, JPEG, or PNG format.

All of the graphics objects respond to the following commands:

```
<object>.draw(<window>);
```
Draws the `<object>` in the specified graphics window `<window>`.

```
<object>.undraw();
```
Undraws `<object>`.

```
<object>.getCenter()
```
Returns the center point of the `<object>`.

```
<line-object>.getP1()
<line-object>.getP2()
```
Returns the end points of the `<line-object>`.

```
<object>.setOutline(<color>);
<object>.setFill(<color>);
```
Sets the outline and the fill color of the `<object>` to the specified `<color>`.

```
<object>.setWidth(<pixels>);
```
Sets the thickness of the outline of the `<object>` to `<pixels>`.

```
<object>.move(<dx>, <dy>);
```
Moves the object <dx>, <dy> from its current position.

The following sound-related functions were presented in this chapter.

```
<robot/computer object>.beep(<seconds>, <frequency>);
<robot/computer object>.beep(<seconds>, <f1>, <f2>);
```
Makes the robot or computer beep for <seconds> time at frequency specified. You can either specify a single frequency <frequency> or a mix of two: <f1> and <f2>.

```
robot.playSong(<song>) ;
```
Plays the <song> on the robot.

```
readSong(<filename>)
Reads a song object from <filename>.
```

```
song2text(song)
```
Converts a song to text format.

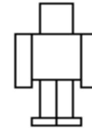
```
makeSong(<text>)
text2song(<text>)
```
Converts <text> to a song format.

C++ reference

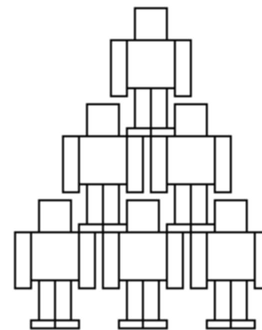
In this chapter we presented informal *scope rules* for names in C++ programs. While these can get fairly complicated, for our purposes you need to know the distinction between a *local name* that is local within the scope of a function versus a *global name* defined outside of the function. The text ordering defines what is accessible.

Exercises

1. Using the graphics commands learned in this chapter, write a program to generate a seasonal drawing: winter, summer, beach, or a festival scene.



2. Write a program that has a function `drawRobot(x, y, w)` such that it draws the figure of a robot shown on the right. The robot is anchored at (`x`, `y`) and is drawn in the window, `w`. Also notice that the robot is entirely made up of rectangles (8 of them).



3. Using the `drawRobot` function from the previous exercise, draw a pyramid of robot as shown on the right.

4. Suppose that each robot from the previous exercise is colored using one of the colors: red, blue, green, and yellow. Each time a robot is drawn, it uses the next color in the sequence. Once the sequence is exhausted, it recycles. Write a program to draw the above pyramid so robots appear in these colors as they are drawn. You may decide to modify `drawRobot` to have an additional parameter: `drawRobot(x, y, c, w)` or you can write it so that `drawRobot` decides which color it chooses. Complete both versions and compare the resulting programs. Discuss the merits and/or pitfalls of these versions with friends. [Hint: Use a vector of color names.]

5. The beep command can play two tones simultaneously. In this exercise we will demonstrate an interesting phenomena. First, Try the following commands:

```
robot.beep(2, 660);
robot.beep(2, 665);
```

You may or may not be able to tell the two tones apart. Next, try playing the two tones together using the command:

```
robot.beep(2, 660, 665);
```

You will hear pulsating beats. This phenomenon called *beating* is very common when you play pure tone together. Explain why this happens.

6. Do a web search on *fractals* and write programs to draw some simple fractals. For example, *Koch Snowflakes*, *Sierpinski Triangles*, *Mandelbrot Sets*, *Julia Sets*, etc.