1 Administrative Topics

- Show some cool projects.

2 How do objects fit into our memory model?

Let’s look at a simple image manipulation example to see how we represent objects in memory. We will step through code similar to that on Monday. We use the display function from Monday, but write a new function blackDotPixmap to put change the color of four neighboring pixels to black. (Note: This isn’t a good function after which to model yours - it is an easy function to step through!).

So, the code begin by importing the graphics module, then defines the display function, then adds the new function, and finally has the main code:
import graphics

# display Pixmap pm in a window with the given title

def displayPixmap( pm, title ):
    w = pm.getWidth()
    h = pm.getHeight()
    win = graphics.GraphWin(title, w, h)
    img = graphics.Image(graphics.Point(w/2.0, h/2.0), pm)
    img.draw(win)
    win.getMouse() # pause for a click in the window
    win.close()

# Place a 2x2 black dot on the image at (250,50)

def blackDotPixmap( pm ):
    pm.setPixel( 250, 50, (0,0,0) )
    pm.setPixel( 250, 51, (0,0,0) )
    pm.setPixel( 251, 50, (0,0,0) )
    pm.setPixel( 251, 51, (0,0,0) )

# main
pm = graphics.Pixmap('george1.ppm')
blackDotPixmap(pm)
displayPixmap(pm, "George is Cute!")
Let’s step through the code. By the time we get to line 20, there are three entries in the symbol table:
Line 21 statement is an assignment statement. We evaluate the right hand side (by calling the Pixmap function in the graphics module) and a Pixmap is returned.

```
main

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphics</td>
<td>&lt;module&gt;</td>
</tr>
<tr>
<td>displayPixmap</td>
<td>&lt;fcn&gt;</td>
</tr>
<tr>
<td>blackDotPixmap</td>
<td>&lt;fcn&gt;</td>
</tr>
</tbody>
</table>

pm = graphics.Pixmap('george1.ppm')

Pixmap

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>getWidth</td>
<td>&lt;fcn&gt;</td>
</tr>
<tr>
<td>getHeight</td>
<td>&lt;fcn&gt;</td>
</tr>
<tr>
<td>clone</td>
<td>&lt;fcn&gt;</td>
</tr>
</tbody>
</table>
```
We then update the symbol table by adding an entry whose name is pm and whose value is an arrow (in the table) to a Pixmap object (outside the table):
Next, we call `blackDotPixmap` (line 22).

A new symbol table is created. There is an entry automatically added for the parameter. The name is `pm` and the value is an arrow to the same Pixmap object:
Then, line 15 is executed and pixel (250,50) is colored black:
Now line 16 is executed and pixel (250, 51) is colored black.
Now line 17 is executed and pixel (251, 50) is colored black.
Now line 18 is executed and pixel (251, 51) is colored black.
Since both the main code and blackDotPixmap are pointing to the same object, there is no need for negatePixmap to return anything. It is simply manipulating the same pixels main has access to. This is helpful because it cuts down on the amount of memory required and on the time needed to copy data from one memory location to another. So, blackDotPixmap returns None and its symbol table goes away. Control returns to the main code and it is now pointing to the updated Pixmap:

It is this updated Pixmap that is passed in to displayPixmap.

Done with this part of the demo.
What if we don’t want to alter the original pixmap? What if we want to keep it around so that we can apply a different manipulation function? One option is to create two Pixmap objects from the same file. Another is to use the clone method. `pm.clone()` will return a new copy of the Pixmap contained in `pm`.

Suppose we add two lines to the main code from above:

```python
newpm = pm.clone()
displayPixmap(pm, "Really, I mean it!")
```

The right hand side of the assignment statement evaluates to a pointer to a new Pixmap.
And the symbol table is updated to point to both the original Pixmap and a clone:

The final line displays the Pixmap. It has no effect on the main symbol table.