Homework

31

Images & Subscripts

We have used pointers and structures to modify images in previous assignments. Now that you have an Image class, we're going to implement a `translate()` member function which uses the overloaded subscript operators to rotate the pixels in the image in both the horizontal and vertical dimension. It will shift each pixel by ***dx*** in the ***x*** dimension, and ***dy*** in the ***y*** dimension. Both ***dx*** and ***dy*** will be arguments, and either may be negative, or, larger than the width or height of the image.

Here's an example using letters in place of pixels.

before translate by (dx=2, dy=-1)

	0	1	2	3	4	5
0	A	B	C	D	E	F
1	G	H	I	J	K	L
2	M	N	O	P	Q	R
3	S	T	U	V	W	X

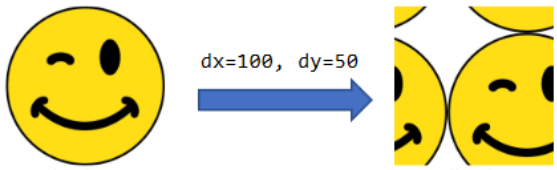
-->

after

	0	1	2	3	4	5
0	K	L	G	H	I	J
1	Q	R	M	N	O	P
2	W	X	S	T	U	V
3	E	F	A	B	C	D

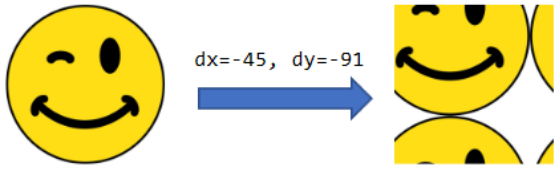
Here the 6x4 image is shifted in the ***x*** dimension by ***2*** and the ***y*** dimension by ***-1***. If you follow the letter ***a***, you can see it has moved to column ***2***. However, since it is in row ***0***, when it is shifted by ***-1***, it wraps around to the bottom row.

Here's a picture with an actual image, so you can see the effect.



smiley-face.png

smiley-face-translate-100-50.png



Step 1: Getting Started

Open **h31.h** and copy the prototypes for the two overloaded operators and the **translate** member function into **h31.cpp**. Then:

1. Remove the semicolons and add a body **{ }** to each one
2. Fully qualify each of the names (**Image::**)
3. Fill in the returned value for both overloaded operators like this:

```
return m_pixels.at(idx);
```

That's all there is too it. You should be able to **make** test **and** see that the overloaded operators now work correctly. Of course, you aren't doing any of the work; you're handing it off to the member functions in the **vector** class.

Step 2: Translate Horizontal

The algorithm to do this efficiently is fairly complex for this point in the course. Instead, we'll adopt the following brute-force algorithm which uses three nested loops.

```
for row from 0 to height
   repeat dx times
      Save the last element in the row in temp
      for col from last to 0
         Copy pixel at col-1 to col
      place temp in the first position in the row
```

This is fairly easy. There are only a few tricky spots.

- At the top of the function, create a reference to the current object by dereferencing the pointer named **this**, passed to every member function.

  ```
  Image& img = *this;
  ```

- The subscript operator uses only a single index. To convert row/column indexes to single index, use the following formula:

  ```
  index = row * width() + col
  ```

- The inner-most loop has to move from right-to left to shift items to the right. If you go the other way, every pixel will be the same.

If you **make test** you'll see that the two positive horizontal tests pass. (**translate(50, 0)** and **translate(385, 0)**). If they don't then ask for help.

Step 3: Translate Vertical

This is a little more complex. The code appears **after** shifting left or right. That means you need another triple-nested loop. Here's the algorithm:

```
lastRow <- width * (height - 1)
repeat dy times
   for col from 0 to width
      temp <- img[lastRow + col]
      for row from height - 1 to 0
         img[row*width + col] <- img[(row-1)*width + col]
      img[col] <- temp
```

Here's what these lines do:

1. **Calculate the address of the last row before** the loop.
2. Write an outer loop that repeats ***dy*** times.
3. Inside that, add another that **visits every column.**
4. Inside that inner loop, **save the pixel** at the current column **in the last row**. At the end of this loop, place the saved pixel **in the top row**.
5. **Between those two lines, shift** the pixels from row to row, column by column. Copy **into the pixel on the current** row and copy **from the pixel on the previous row**. The loop **starts at the last row** and goes toward the top.

Go ahead and **make test** once again; only tests with **negative *dx*** or ***dy*** fail.

Step 4: Handle Negatives

We could check if ***dx*** or ***dy*** is negative, and then use a different loop that copies in the opposite direction (from left to right instead of right to left, for instance). While fine, it's really not needed.

The key is to realize that a ***dx*** of ***-1*** (translate left) can be thought of as the same as a ***dx*** of ***width - 1*** (translate right). If we simply add negative ***dx*** values to ***width*** and add negative ***dy*** values to ***height***, we won't need to change our algorithm at all.

```
if (dx < 0) dx = width() - abs(dx) % width();
else        dx = dx % width();
```

The code for ***dy*** (appearing before the vertical section) is similar. When you fix that and **make test**, all of the tests should pass.

Be sure to **`make submit`** to turn in your code for credit **before the deadline**. As always, if you run into problems, bring your questions to Piazza or come to my office hour.