resize

TODO

- open file
- update outfile’s header info
- read infile’s scanline, pixel by pixel
- resize horizontally
- remember padding!
- resize vertically

copy.c

- opens a file
- updates header info for outfile
- reads each scanline, pixel by pixel
- writes each pixel into the output file's scanline

```
cp copy.c resize.c
```

TODO

- ☑ open file
- ☐ update outfile’s header info
- ☐ read infile’s scanline, pixel by pixel
- ☐ resize horizontally
- ☐ remember padding!
- ☐ resize vertically

bitmaps

- just an arrangement of bytes!
- how do we interpret this arrangement?
- `bmp.h`

updating header info

- new bmp → new header info
- what's changing?
  - file size
  - image size
  - width
  - height

BITMAPINFOHEADER

- `biWidth`
  - width of image (in pixels)
    - does not include padding
- `biHeight`
  - height of image (in pixels)

BITMAPINFOHEADER

- biSizeImage
  - total size of image (in bytes)
    - includes pixels and padding

```
bi.biSizeImage =
    ((sizeof(RGBTRIPLE) * bi.biWidth) + padding)
    * abs(bi.biHeight);
```

BITMAPFILEHEADER

- `bfSize`
  - total size of file (in bytes)
  - includes pixels, padding, and headers
- `bf.bfSize = bi.biSizeImage + sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER);`

what’s changing?

old	new
bi.biWidth	bi.biWidth *= n
bi.biHeight	bi.biHeight *= n
bi.biSizeImage	...?
bf.bfSize	...?

TODO

- ☑ open file
- ☑ update outfile’s header info
- ☐ read infile’s scanline, pixel by pixel
- ☐ resize horizontally
- ☐ remember padding!
- ☐ resize vertically

reading files

```
fread(data, size, number, inptr);
```

- `data`: pointer to a struct that will contain the bytes you're reading
- `size`: size of each element to read
  - `sizeof`
- `number`: number of elements to read
- `inptr`: `FILE *` to read from

TODO

- ☑ open file
- ☑ update outfile’s header info
- ☑ read infile’s scanline, pixel by pixel
- ☐ resize horizontally
- ☐ remember padding!
- ☐ resize vertically

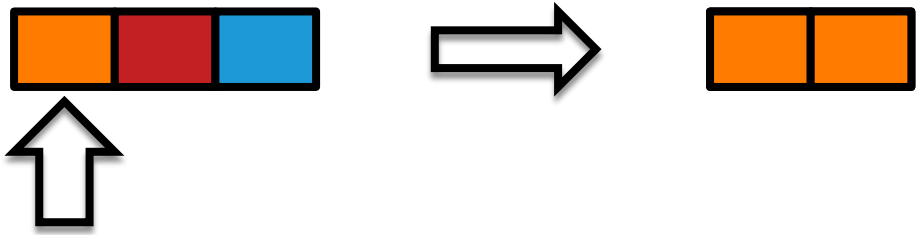
resize horizontally

n = 2

```
for each pixel in row
    write n times
```

resize horizontally

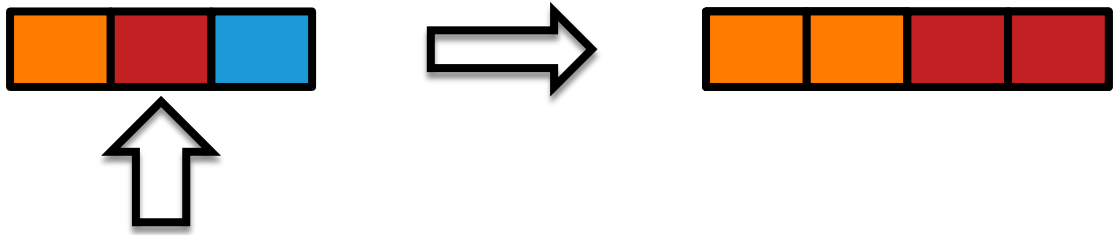
n = 2



```
for each pixel in row
    write n times
```

resize horizontally

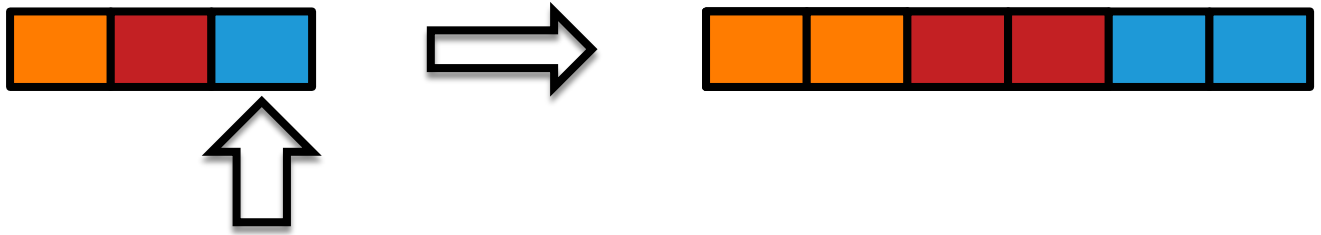
n = 2



```
for each pixel in row
    write n times
```

resize horizontally

n = 2



```
for each pixel in row
    write n times
```

writing files

```
fwrite(data, size, number, outptr);
```

- `data`: pointer to the struct that contains the bytes you're reading from
- `size`
- `number`
- `outptr`: `FILE *` to write to

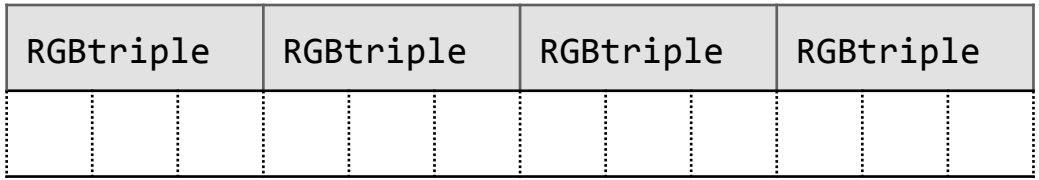
TODO

- ☑ open file
- ☑ update outfile’s header’s info
- ☑ read infile’s scanline, pixel by pixel
- ☑ resize horizontally
- ☐ remember padding!
- ☐ resize vertically

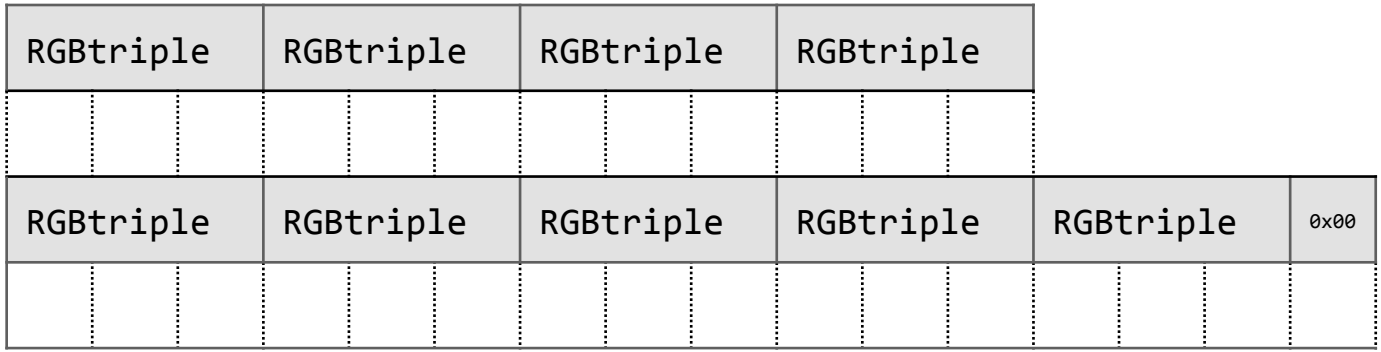
padding

- each pixel is 3 bytes
- length of each scanline must be a multiple of 4 bytes
- if the number of pixels isn't a multiple of 4, we need "padding"
  - padding is just zeros (`0x00`)

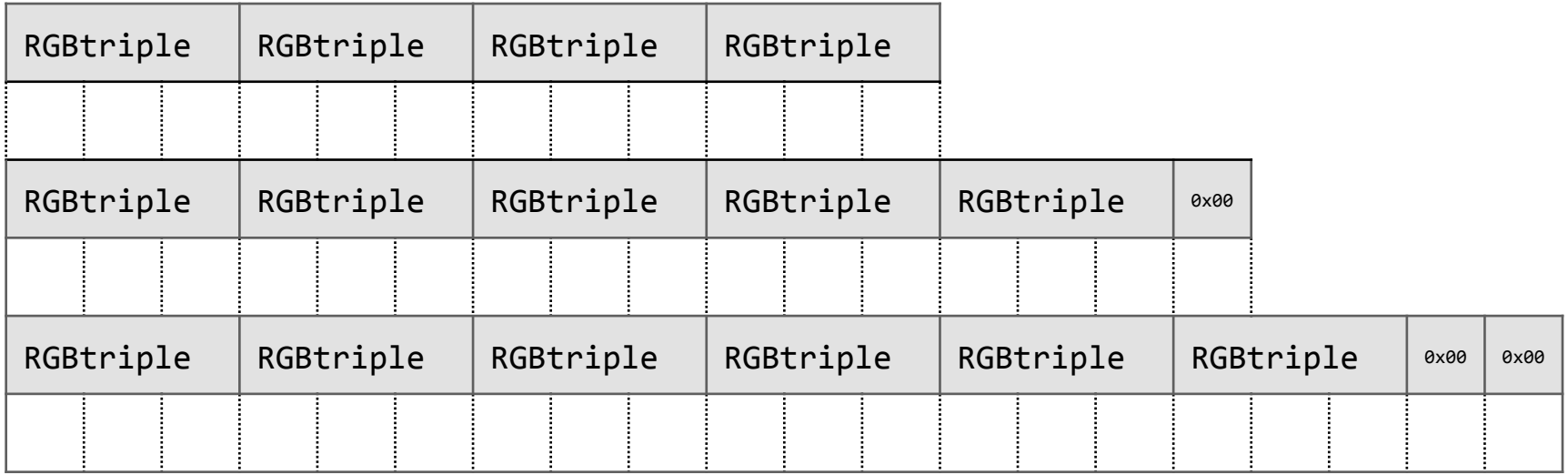
padding



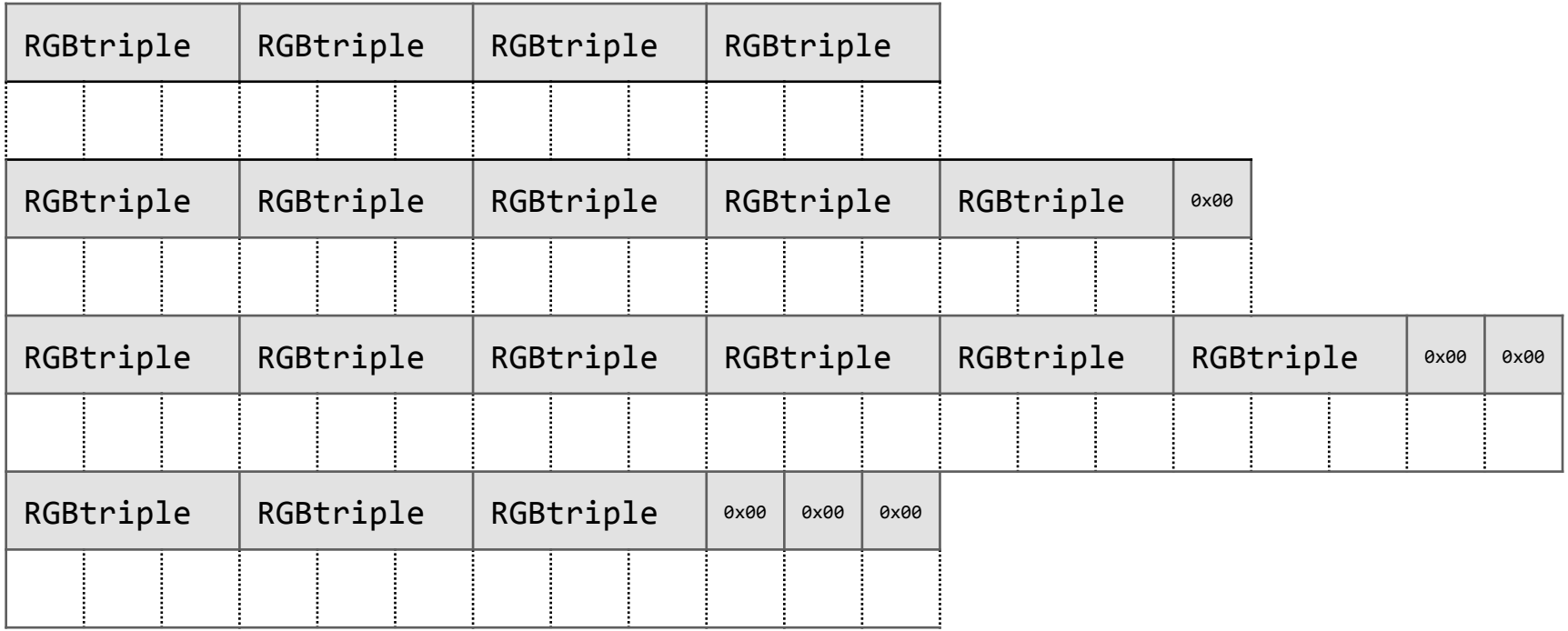
padding


RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
0x00

padding


RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
0x00
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
0x00
0x00

padding


RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
0x00
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
RGBtriple
0x00
0x00
RGBtriple
RGBtriple
RGBtriple
0x00
0x00
0x00

padding

```
padding = (4 – (bi.biWidth * sizeof(RGBTRIPLE)) % 4) % 4
```

- the outfile and infile have different widths
  - so the padding is different!
- padding isn't an `RGBTRIPLE`
  - we can't `fread` padding

writing padding

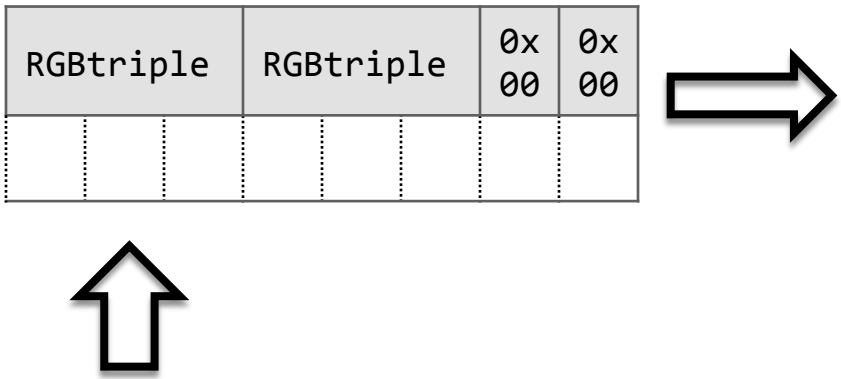
```
fputc(chr, outptr);
```

- chr: char to write
- outptr: `FILE *` to write to

```
fputc(0x00, outptr);
```

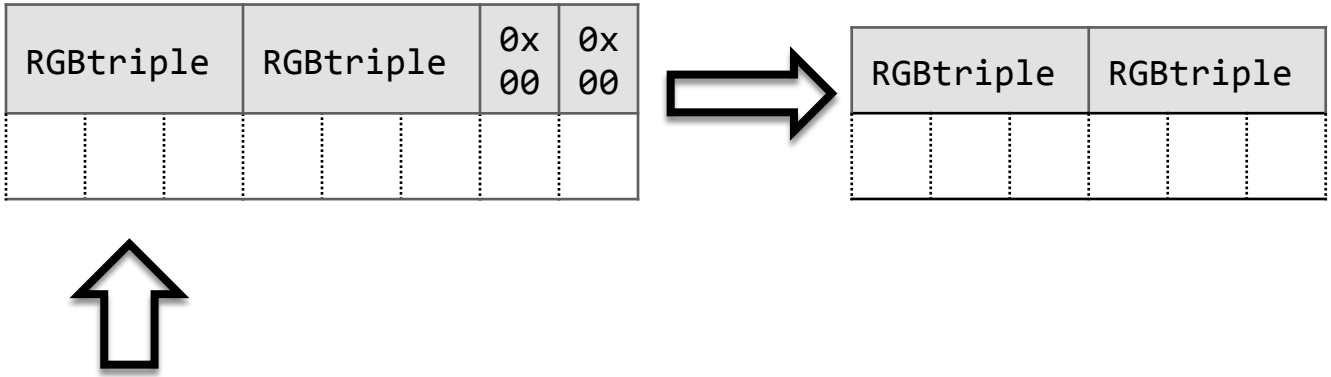
padding

n = 2



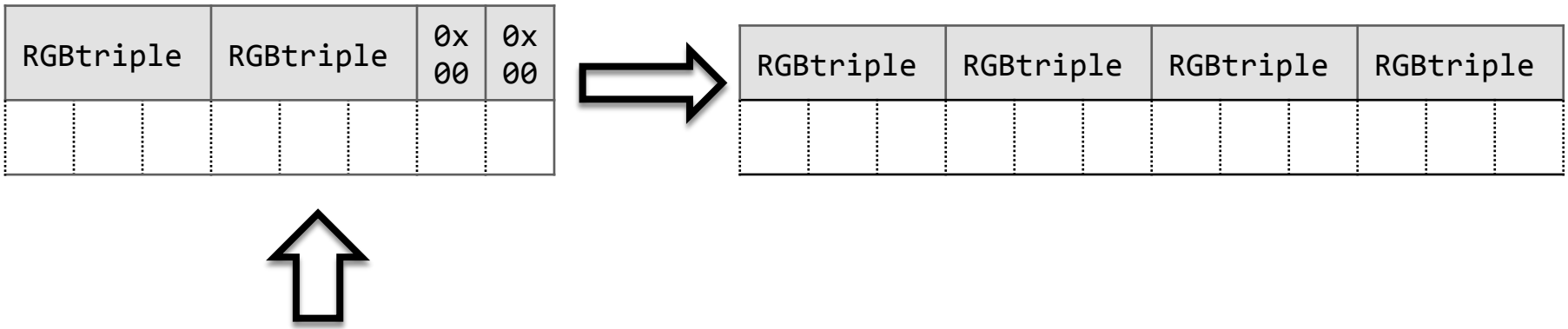
padding

n = 2



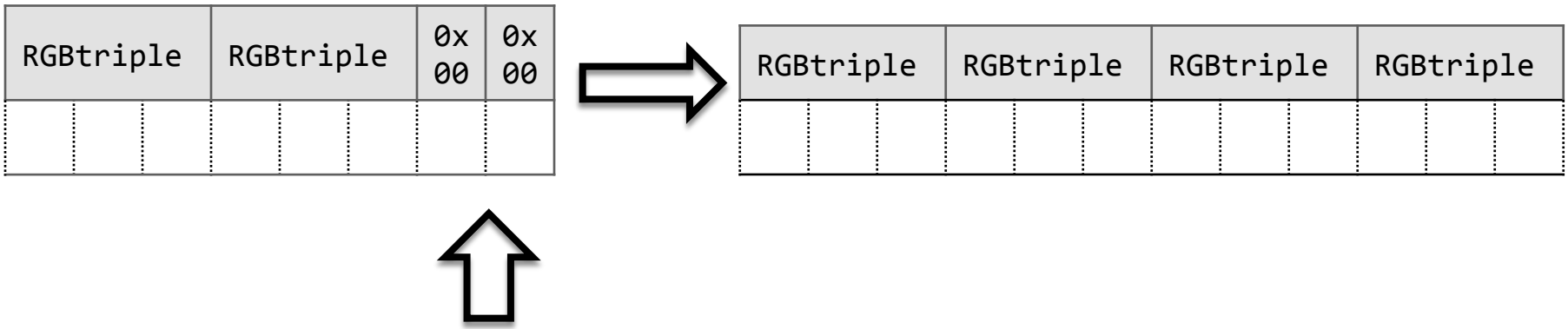
padding

n = 2


RGBtriple
RGBtriple
0x 00
0x 00
RGBtriple
RGBtriple
RGBtriple
RGBtriple

padding

n = 2


RGBtriple
RGBtriple
0x 00
0x 00
RGBtriple
RGBtriple
RGBtriple
RGBtriple

```
padding = (4 – (bi.biWidth * sizeof(RGBTRIPLE)) % 4) % 4
```

what’s changing?

old	new
bi.biWidth	bi.biWidth *= n
bi.biHeight	bi.biHeight *= n
bi.biSizeImage	...?
bf.bfSize	...?

what's changing?

old	new
bi.biWidth	bi.biWidth *= n
bi.biHeight	bi.biHeight *= n
bi.biSizeImage	...?
bf.bfSize	...?
padding	...?

pseudocode: resizing horizontally

```
for each row
    for each pixel in row
        write to outfile n times
    write outfile's padding
    skip over infile's padding
```

TODO:

- ☑ open file
- ☑ update outfile’s header info
- ☑ read infile’s scanline, pixel by pixel
- ☑ resize horizontally
- ☑ remember padding!
- ☐ resize vertically

resize

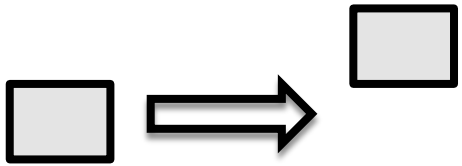
- every pixel repeated n times
- every row repeated n times

n = 3

resize

- every pixel repeated n times
- every row repeated n times

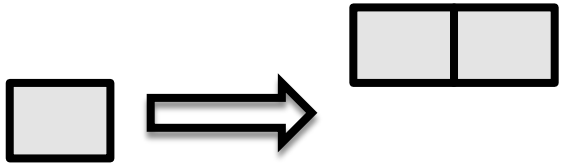
n = 3



resize

- every pixel repeated n times
- every row repeated n times

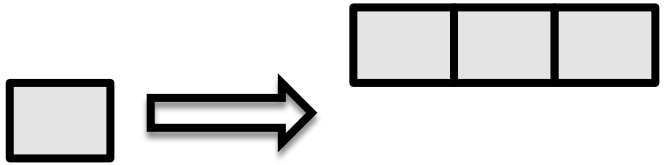
n = 3



resize

- every pixel repeated n times
- every row repeated n times

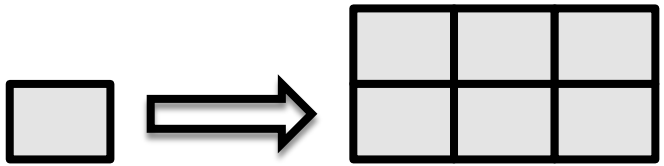
n = 3



resize

- every pixel repeated n times
- every row repeated n times

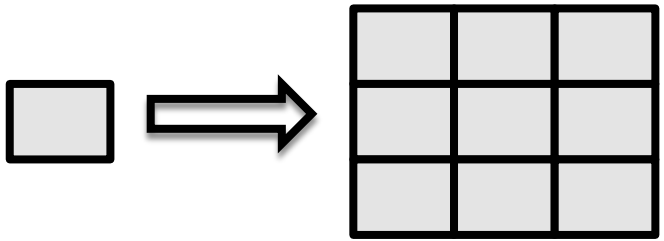
n = 3



resize

- every pixel repeated n times
- every row repeated n times

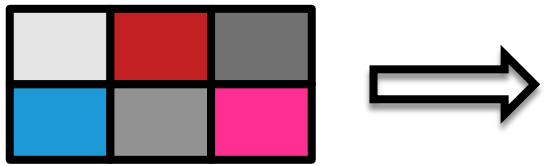
n = 3



resize

- every pixel repeated n times
- every row repeated n times

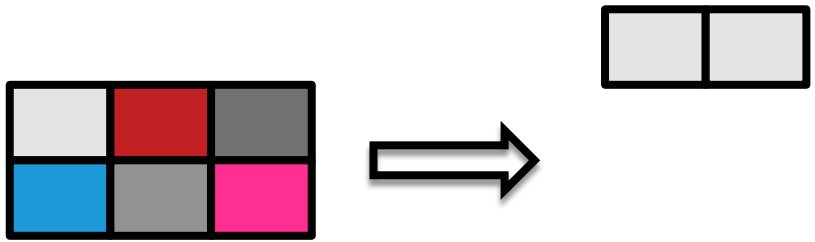
n = 2



resize

- every pixel repeated n times
- every row repeated n times

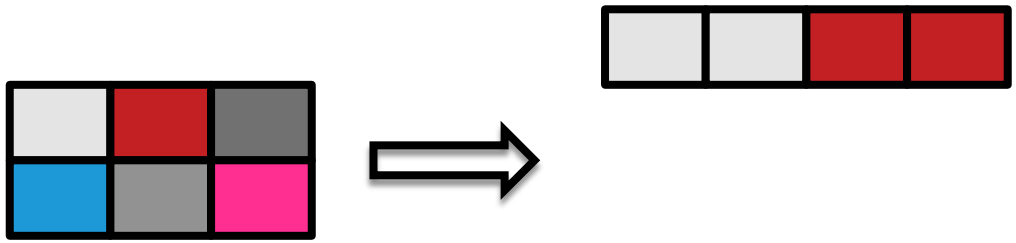
n = 2



resize

- every pixel repeated n times
- every row repeated n times

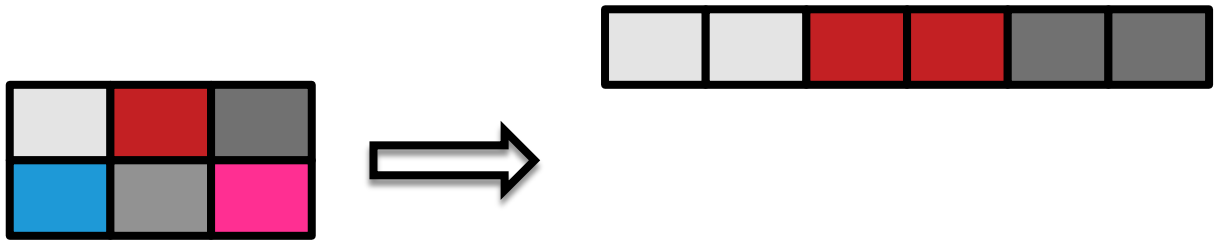
n = 2



resize

- every pixel repeated n times
- every row repeated n times

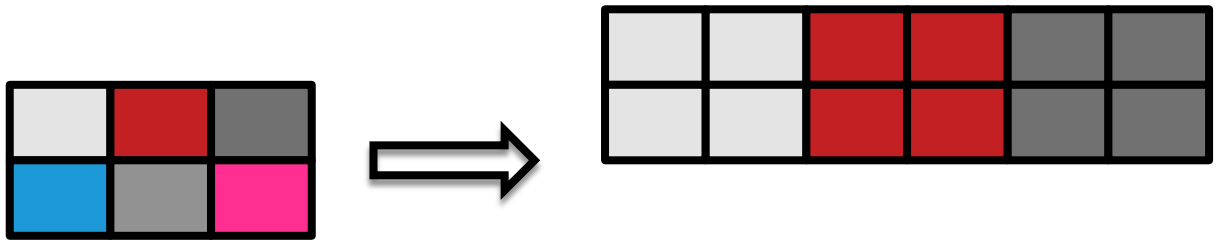
n = 2



resize

- every pixel repeated n times
- every row repeated n times

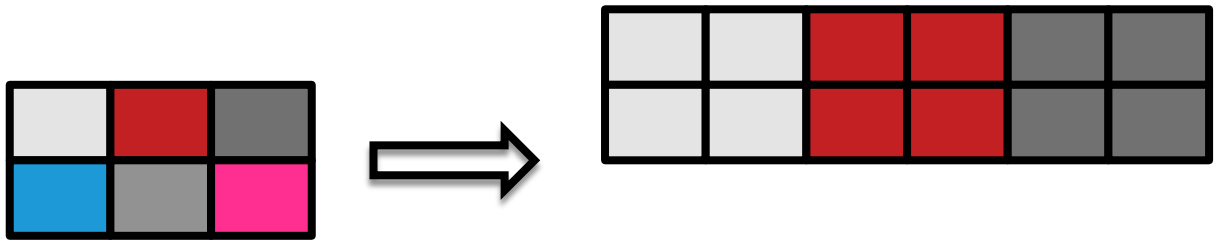
n = 2



resize

- every pixel repeated n times
- every row repeated n times

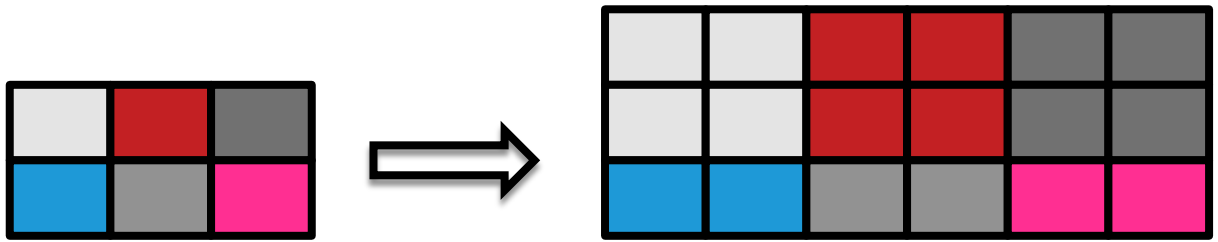
n = 2



resize

- every pixel repeated n times
- every row repeated n times

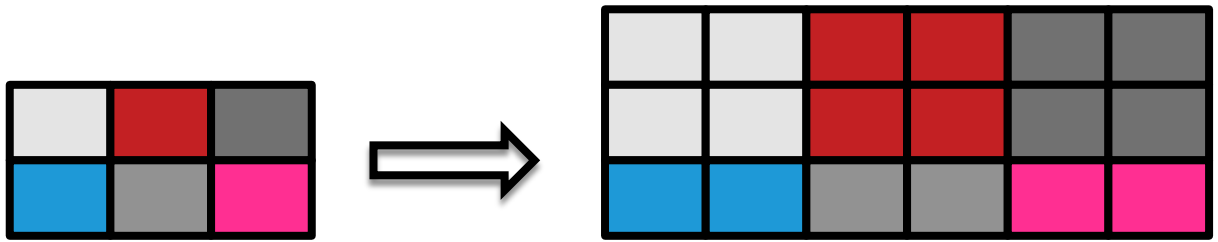
n = 2



resize

- every pixel repeated n times
- every row repeated n times

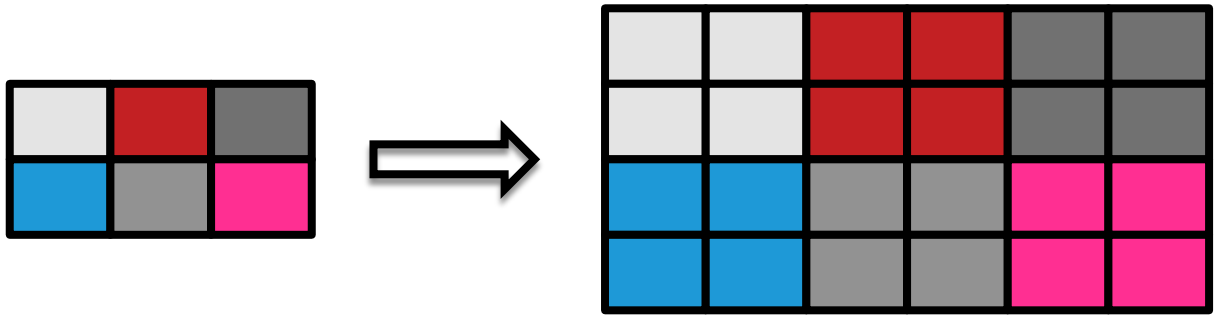
n = 2



resize

- every pixel repeated n times
- every row repeated n times

n = 2



resize vertically

- multiple ways to do this!

1. “rewrite” methods
   - remember pixels in an array
   - write array as many times as needed
2. “re-copy” methods
   - go back to the start of the original scanline
   - re-scale scanline

file position indicator

```
fseek(inptr, offset, from);
```

- inptr: `FILE *` to seek in
- offset: number of bytes to move cursor
- from:
  - `SEEK_CUR` (current position in file)
  - `SEEK_SET` (beginning of file)
  - `SEEK_END` (end of file)

pseudocode: “rewrite” method

```
for each row
    for each pixel
        write to array n times
    for n times
        write array to outfile
        write outfile padding
    skip over infile padding
```

pseudocode: “recopy” method

```
for each row
    for n-1 times
        write pixels, padding to outfile
        send infile cursor back
    write pixels, padding to outfile
    skip over infile padding
```

TODO

- ☑ open file
- ☑ update outfile’s header info
- ☑ read infile’s scanline, pixel by pixel
- ☑ resize horizontally
- ☑ remember padding!
- ☑ resize vertically

this was resize