# Resize

## [**tl;dr**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#tldr)

Implement a program that resizes BMPs, per the below.

$ ./resize 4 small.bmp large.bmp

## [**Introduction**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#introduction)

https://youtu.be/DL27z7YRxj4

## [**Getting Started**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#getting-started)

Here’s how to download this problem’s "distribution code" (i.e., starter code) into your own CS50 IDE. Log into [CS50 IDE](https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

1. Execute cd to ensure that you’re in ~/ (aka your home folder).
2. Execute mkdir pset3 to make (i.e., create) a directory called pset3 in your home directory.
3. Execute cd pset3 to change into (i.e., open) that directory.
4. Execute wget <https://cdn.cs50.net/2018/fall/psets/3/resize/more/resize.zip> to download a (compressed) ZIP file with this problem’s distribution.
5. Execute unzip resize.zip to uncompress that file.
6. Execute rm resize.zip followed by yes or y to delete that ZIP file.
7. Execute ls. You should see a directory called resize, which was inside of that ZIP file.
8. Execute cd resize to change into that directory.
9. Execute ls. You should see a directory called more.
10. Execute cd more to change into that directory.
11. Execute ls. You should see this problem’s distribution, including bmp.h, copy.c, large.bmp, small.bmp, and smiley.bmp.

## [**Background**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#background)

First, be sure you’re familiar with the structure of 24-bit uncompressed BMPs, as introduced in [Whodunit](https://lab.cs50.io/cs50/labs/2019/x/whodunit).

To reiterate a bit from that lab, recall that a file is just a sequence of bits, arranged in some fashion. A 24-bit BMP file, then, is essentially just a sequence of bits, (almost) every 24 of which happen to represent some pixel’s color. But a BMP file also contains some "metadata," information like an image’s height and width. That metadata is stored at the beginning of the file in the form of two data structures generally referred to as "headers," not to be confused with C’s header files. (Incidentally, these headers have evolved over time. This problem only expects that you support the latest version of Microsoft’s BMP format, 4.0, which debuted with Windows 95.) The first of these headers, called BITMAPFILEHEADER, is 14 bytes long. (Recall that 1 byte equals 8 bits.) The second of these headers, called BITMAPINFOHEADER, is 40 bytes long. Immediately following these headers is the actual bitmap: an array of bytes, triples of which represent a pixel’s color. (In 1-, 4-, and 16-bit BMPs, but not 24- or 32-, there’s an additional header right after BITMAPINFOHEADER called RGBQUAD, an array that defines "intensity values" for each of the colors in a device’s palette.) However, BMP stores these triples backwards (i.e., as BGR), with 8 bits for blue, followed by 8 bits for green, followed by 8 bits for red. (Some BMPs also store the entire bitmap backwards, with an image’s top row at the end of the BMP file. But we’ve stored this problem set’s BMPs as described herein, with each bitmap’s top row first and bottom row last.) In other words, were we to convert the 1-bit smiley above to a 24-bit smiley, substituting red for black, a 24-bit BMP would store this bitmap as follows, where 0000ff signifies red and ffffff signifies white; we’ve highlighted in red all instances of 0000ff.

ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff

ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff

0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff

0000ff ffffff ffffff ffffff ffffff ffffff ffffff 0000ff

0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff

0000ff ffffff ffffff 0000ff 0000ff ffffff ffffff 0000ff

ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff

ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff

Because we’ve presented these bits from left to right, top to bottom, in 8 columns, you can actually see the red smiley if you take a step back.

To be clear, recall that a hexadecimal digit represents 4 bits. Accordingly, ffffff in hexadecimal actually signifies 111111111111111111111111 in binary.

Now look at the underlying bytes that compose smiley.bmp. Within CS50 IDE’s file browser, right- or control-click **smiley.bmp** and select **Open as hexadecimal** in order to view the file’s bytes in hexadecimal (i.e., base-16). In the tab that appears, change **Start with byte** to **54**, change **Bytes per row** to **24**, change **Bytes per column** to **3**. Then click **Set**. **If unable to change these values, try clicking View > Night Mode and try again**. You should see the below, byte 54 onward of smiley.bmp. (Recall that a 24-bit BMP’s first 14 + 40 = 54 bytes are filled with metadata, so we’re simply ignoring that for now.) As before, we’ve highlighted in red all instances of 0000ff.

ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff

ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff

0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff

0000ff ffffff ffffff ffffff ffffff ffffff ffffff 0000ff

0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff

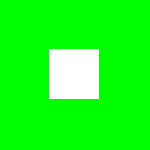
0000ff ffffff ffffff 0000ff 0000ff ffffff ffffff 0000ff

ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff

ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff

So, smiley.bmp is 8 pixels wide by 8 pixels tall, and it’s a 24-bit BMP (each of whose pixels is represented with 24 ÷ 8 = 3 bytes). Each row (aka "scanline") thus takes up (8 pixels) × (3 bytes per pixel) = 24 bytes, which happens to be a multiple of 4.

It turns out, though, that BMPs are stored a bit differently if the number of bytes in a scanline is not, in fact, a multiple of 4. In small.bmp, for instance, is another 24-bit BMP, a green box that’s 3 pixels wide by 3 pixels wide. If you view it (as by double-clicking it), you’ll see that it resembles the below, albeit much smaller. (Indeed, you might need to zoom in again to see it.)



Each scanline in small.bmp thus takes up (3 pixels) × (3 bytes per pixel) = 9 bytes, which is **not** a multiple of 4. And so the scanline is "padded" with as many zeroes as it takes to extend the scanline’s length to a multiple of 4. In other words, between 0 and 3 bytes of padding are needed for each scanline in a 24-bit BMP. (Understand why?) In the case of small.bmp, 3 bytes' worth of zeroes are needed, since (3 pixels) × (3 bytes per pixel) + (3 bytes of padding) = 12 bytes, which is indeed a multiple of 4.

To "see" this padding, right- or control-click **small.bmp** in CS50 IDE’s file browser and select **Open as hexadecimal**. In the tab that appears, change **Start with byte** to **54**, change **Bytes per row** to **12**, and change **Bytes per column** to **3**. Then click **Set**. You should see output like the below; we’ve highlighted in green all instances of 00ff00.

00ff00 00ff00 00ff00 000000

00ff00 ffffff 00ff00 000000

00ff00 00ff00 00ff00 000000

For contrast, let’s next look at large.bmp, which looks identical to small.bmp but, at 12 pixels by 12 pixels, is four times as large. Right- or control-click it in CS50 IDE’s file browser, then select **Open as hexadecimal**. You should see output like the below; we’ve again highlighted in green all instances of 00ff00

00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00

00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00

Worthy of note is that this BMP lacks padding! After all, (12 pixels) × (3 bytes per pixel) = 36 bytes is indeed a multiple of 4.

Knowing all this has got to be useful!

## [**Specification**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#specification)

Implement a program called resize that resizes (i.e., enlarges or shrinks) 24-bit uncompressed BMPs by a factor of f.

* Implement your program in a file called resize.c inside of ~/pset3/resize/more/.
* Your program should accept exactly three command-line arguments, whereby
  + the first (f) must be a floating-point value in (0.0, 100.0],
  + the second must be the name of a BMP to be resized, and
  + the third must be the name of the resized version to be written.

+ If your program is not executed with such, it should remind the user of correct usage, as with printf, and main should return 1.

* Your program, if it uses malloc, must not leak any memory.

## [**Walkthrough**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#walkthrough)

https://youtu.be/iv9cw-PTSnA

## [**Usage**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#usage)

Your program should behave per the examples below. Assumed that the underlined text is what some user has typed.

$ ./resize

Usage: ./resize f infile outfile

$ echo $?

1

$ ./resize .5 large.bmp smaller.bmp

$ echo $?

0

$ ./resize 2 small.bmp larger.bmp

$ echo $?

0

## [**Hints**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#hints)

With a program like this, we could have created large.bmp out of small.bmp by resizing the latter by a factor of 4 (i.e., by multiplying both its width and its height by 4), per the below.

./resize 4 small.bmp large.bmp

You’re welcome to get started by copying (yet again) copy.c and naming the copy resize.c. But spend some time thinking about what it means to resize a BMP, particularly if f is in (0.0, 1.0). (You may assume that n times the size of infile will not exceed 232 - 1. As for a value of 1.0 for f, the result should indeed be an outfile with dimensions identical to infile's.) How you handle floating-point imprecision and rounding is entirely up to you, as is how you handle inevitable loss of detail. Decide which of the fields in BITMAPFILEHEADER and BITMAPINFOHEADER you might need to modify. Consider whether or not you’ll need to add or subtract padding to scanlines. And do be sure to support a value of 1 for n, the result of which should be an outfile with dimensions identical to infile's.

If you happen to use malloc, be sure to use free so as not to leak memory. Try using valgrind to check for any leaks!

## [**Testing**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#testing)

If you’d like to peek at, e.g., large.bmp's headers (in a more user-friendly way than xxd allows), you may execute the below.

~cs50/2019/x/hacker3/peek large.bmp

Better yet, if you’d like to compare your outfile’s headers against those from the [staff’s solution](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#staffs-solution), you might want to execute commands like the below. (Think about what each is doing.)

./resize 4 small.bmp student.bmp

~cs50/2019/x/hacker3/resize 4 small.bmp staff.bmp

~cs50/2019/x/hacker3/peek student.bmp staff.bmp

### [**check50**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#check50)

Here’s how to check the correctness of your code using check50. But be sure to compile and test it yourself as well!

check50 cs50/problems/2019/x/resize/more

### [**style50**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#style50)

Here’s how to evaluate the style of your code using style50.

style50 resize.c

## [**Staff’s Solution**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#staffs-solution)

To try out the staff’s own implementation of resize, execute

./resize

within [this sandbox](<https://sandbox.cs50.io/cf791495-d296-437d-bc60-cf27e43c7622>).

## [**How to Submit**](https://docs.cs50.net/2019/x/psets/3/resize/more/resize.html#how-to-submit)

Execute the below, logging in with your GitHub username and password when prompted. For security, you’ll see asterisks (\*) instead of the actual characters in your password.

submit50 cs50/problems/2019/x/resize/more

You can then go to <https://cs50.me/cs50x> to view your current scores!