Este é o CS50

Introdução à Ciência da Computação (CS50)

OpenCourseWare

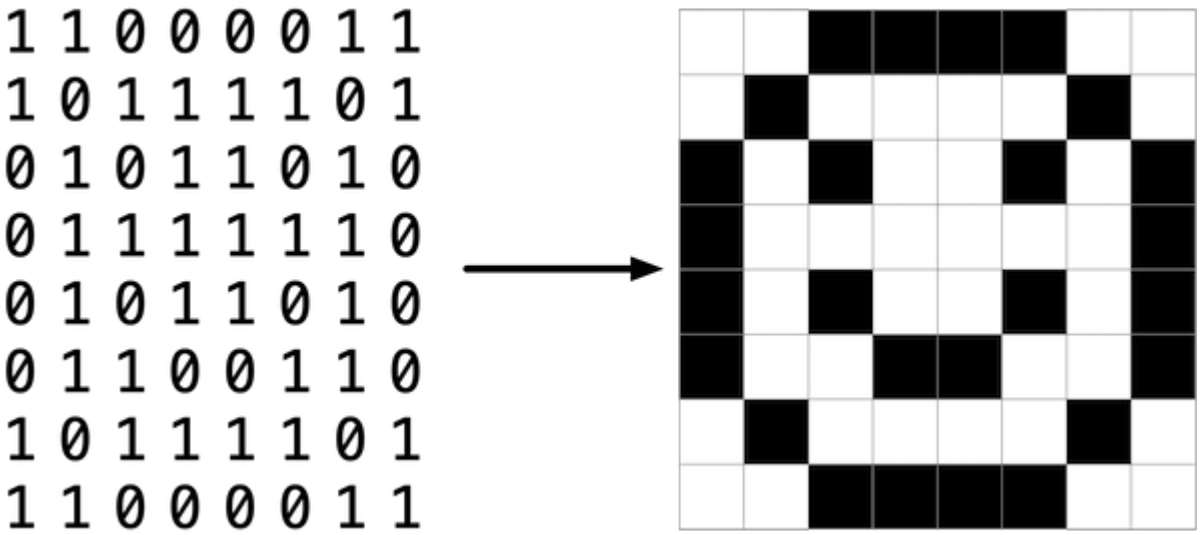
Doar (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Filtro

Problema a resolver

Talvez a maneira mais simples de representar uma imagem seja com uma grade de pixels (ou seja, pontos), cada um dos quais pode ter uma cor diferente. Para imagens em preto e branco, precisamos, portanto, de 1 bit por pixel, já que 0 pode representar preto e 1 pode representar branco, como no exemplo abaixo.



Nesse sentido, então, uma imagem é apenas um bitmap (ou seja, um mapa de bits). Para imagens mais coloridas, você simplesmente precisa de mais bits por pixel. Um formato de arquivo (como BMP (https://en.wikipedia.org/wiki/BMP_file_format) , JPEG (https://en.wikipedia.org/wiki/JPEG) ou PNG (https://en.wikipedia.org/wiki/Portable_Network_Graphics)) que suporta "cor de 24 bits" usa 24 bits por pixel. (O BMP, na verdade, suporta cores de 1, 4, 8, 16, 24 e 32 bits.)

Um BMP de 24 bits usa 8 bits para representar a quantidade de vermelho na cor de um pixel, 8 bits para representar a quantidade de verde na cor de um pixel e 8 bits para representar a quantidade de azul na cor de um pixel. Se você já ouviu falar de cores RGB, bem, aí está: vermelho, verde e azul.

Se os valores R, G e B de um pixel em um arquivo BMP forem, digamos, `0xff`, `0x00`, e `0x00` em hexadecimal, esse pixel é puramente vermelho, pois `0xff` (também conhecido como `255` em decimal) implica "muito vermelho", enquanto `0x00` e `0x00` implicam "nenhum verde" e "nenhum azul", respectivamente. Neste problema, você manipulará esses valores R, G e B de pixels individuais, criando, em última análise, seus próprios filtros de imagem.

Em um arquivo chamado `<nome_do_arquivo>` localizado `helpers.c` em uma pasta chamada `<nome_da_pasta>` `filter-more`, escreva um programa para aplicar filtros a imagens BMP.

Demonstração

```
$ make filter
$ ./filter -g ./images/courtyard.bmp ./images/courtyard
$ ./filter -r ./images/stadium.bmp ./images/stadium-ref
$ ./filter -b ./images/tower.bmp ./images/tower-blurred
$ ./filter -e ./images/yard.bmp ./images/yard-edges.bmp
$ ./filter
Usage: ./filter [flag] infile outfile
$
```

Recorded with **asciinema**

Código de Distribuição

Para este problema, você deverá estender a funcionalidade do código fornecido pela equipe do CS50.

▼ **Baixe o código de distribuição.**

Faça login em cs50.dev (https://cs50.dev/) , clique na janela do terminal e execute `cd` o comando. Você verá que o prompt do terminal será semelhante ao seguinte:

```
$
```

Em seguida, execute

```
wget https://cdn.cs50.net/2024/fall/psets/4/filter-more.zip
```

para baixar um arquivo ZIP chamado `filter-more.zip` para o seu espaço de código.

Em seguida, execute

```
unzip filter-more.zip
```

para criar uma pasta chamada `filter-more`. Você não precisa mais do arquivo ZIP, então pode executar

```
rm filter-more.zip
```

e responda com “y” seguido de Enter quando solicitado para remover o arquivo ZIP que você baixou.

Agora digite

```
cd filter-more
```

Em seguida, pressione Enter para entrar (ou seja, abrir) esse diretório. Seu prompt agora deve ser semelhante ao abaixo.

```
filter-more/ $
```

Execute o programa `ls` sozinho e você deverá ver alguns arquivos: `bmp.h`, `filter.c`, `helpers.h`, `helpers.c`, e `Makefile`. Você também deverá ver uma pasta chamada `images` com quatro arquivos BMP. Se encontrar algum problema, repita esses mesmos passos e veja se consegue identificar onde errou!

Fundo

Um pouco mais técnico

Lembre-se de que um arquivo é apenas uma sequência de bits, organizados de alguma forma. Um arquivo BMP de 24 bits, portanto, é essencialmente apenas uma sequência de bits, (quase) cada 24 dos quais representam a cor de algum pixel. Mas um arquivo BMP também contém alguns "metadados", informações como a altura e a largura de uma imagem. Esses metadados são armazenados no início do arquivo na forma de duas estruturas de dados geralmente chamadas de "cabeçalhos", que não devem ser confundidas com os arquivos de cabeçalho do C. (Aliás, esses cabeçalhos evoluíram ao longo do tempo. Este problema usa a versão mais recente do formato BMP da Microsoft, a 4.0, que estreou com o Windows 95.)

O primeiro desses cabeçalhos, chamado `header` `BITMAPFILEHEADER`, tem 14 bytes de comprimento. (Lembre-se de que 1 byte equivale a 8 bits.) O segundo desses cabeçalhos, chamado `header` `BITMAPINFOHEADER`, tem 40 bytes de comprimento. Imediatamente após esses cabeçalhos está o bitmap propriamente dito: uma matriz de bytes, cujos triplos representam a cor de um pixel. No entanto, o BMP armazena esses triplos de trás para frente (ou seja, como BGR), com 8 bits para azul, seguidos por 8 bits para verde e, por fim, 8 bits para vermelho. (Alguns BMPs também armazenam o bitmap inteiro de trás para frente, com a linha superior da imagem no final do arquivo BMP. Mas armazenamos os BMPs deste conjunto de problemas conforme descrito aqui, com a linha superior de cada bitmap primeiro e a linha inferior por último.) Em outras palavras, se convertêssemos o smiley de 1 bit acima para um smiley de 24 bits, substituindo o preto pelo vermelho, um BMP de 24 bits armazenaria este

bitmap da seguinte forma, onde ` `0000ff`0` significa vermelho e `ffffff`0` significa branco; Destacamos em vermelho todas as ocorrências de `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
```

Como apresentamos esses elementos da esquerda para a direita, de cima para baixo, em 8 colunas, você consegue ver o smiley vermelho se der um passo para trás.

Para ficar claro, lembre-se de que um dígito hexadecimal representa 4 bits. Portanto, `ffffff` em hexadecimal, na verdade significa `111111111111111111111111` em binário.

Note que você pode representar um bitmap como uma matriz bidimensional de pixels: onde a imagem é uma matriz de linhas, e cada linha é uma matriz de pixels. De fato, foi assim que optamos por representar imagens bitmap neste problema.

Filtragem de imagens

O que significa filtrar uma imagem? Podemos pensar na filtragem como a criação de pixels em uma imagem original, modificando cada pixel de forma a produzir um efeito específico na imagem resultante.

Escala de cinza

Um filtro comum é o filtro "escala de cinza", no qual pegamos uma imagem e queremos convertê-la para preto e branco. Como isso funciona?

Lembre-se de que, se os valores de vermelho, verde e azul forem todos definidos como `0x00` (hexadecimal para `0`), o pixel será preto. E se todos os valores forem definidos como `0xff` (hexadecimal para `255`), o pixel será branco. Contanto que os valores de vermelho, verde e azul sejam iguais, o resultado será uma variação de tons de cinza ao longo do espectro preto-branco, com valores mais altos representando tons mais claros (mais próximos do branco) e valores mais baixos representando tons mais escuros (mais próximos do preto).

Para converter um pixel em escala de cinza, basta garantir que os valores de vermelho, verde e azul sejam iguais. Mas como saber quais valores usar? Bem, é razoável supor que, se os valores originais de vermelho, verde e azul eram altos, o novo valor também deverá ser alto. E se os valores originais eram baixos, o novo valor também deverá ser baixo.

Na verdade, para garantir que cada pixel da nova imagem ainda tenha o mesmo brilho ou escuridão geral da imagem antiga, podemos calcular a média dos valores de vermelho, verde e azul para determinar qual tom de cinza usar para o novo pixel.

Se você aplicar isso a cada pixel da imagem, o resultado será uma imagem convertida para escala de cinza.

Reflexão

Alguns filtros também podem reorganizar os pixels. O filtro de reflexão, por exemplo, produz a imagem resultante como se a imagem original fosse colocada em frente a um espelho. Assim, quaisquer pixels no lado esquerdo da imagem devem aparecer no lado direito, e vice-versa.

Note que todos os pixels originais da imagem original ainda estarão presentes na imagem refletida; apenas esses pixels podem ter se reorganizado para ocupar um lugar diferente na imagem.

Borrão

Existem diversas maneiras de criar o efeito de desfoque ou suavização em uma imagem. Para este problema, usaremos o "desfoque por caixa", que funciona pegando cada pixel e, para cada valor de cor, atribuindo-lhe um novo valor através da média dos valores de cor dos pixels vizinhos.

Considere a seguinte grade de pixels, onde numeramos cada pixel.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

O novo valor de cada pixel seria a média dos valores de todos os pixels que estão a uma linha e coluna de distância do pixel original (formando uma caixa 3x3). Por exemplo, cada um dos valores de cor para o pixel 6 seria obtido pela média dos valores de cor originais dos pixels 1, 2, 3, 5, 6, 7, 9, 10 e 11 (observe que o próprio pixel 6 está incluído na média). Da mesma forma, os valores de cor para o pixel 11 seriam obtidos pela média dos valores de cor dos pixels 6, 7, 8, 10, 11, 12, 14, 15 e 16.

Para um pixel na borda ou no canto, como o pixel 15, ainda procuraríamos todos os pixels dentro de uma linha e coluna: neste caso, os pixels 10, 11, 12, 14, 15 e 16.

Bordas

Em algoritmos de inteligência artificial para processamento de imagens, é frequentemente útil detectar bordas em uma imagem: linhas na imagem que criam um limite entre um objeto e outro. Uma maneira de obter esse efeito é aplicando o operador de Sobel (https://en.wikipedia.org/wiki/Sobel_operator) à imagem.

Assim como o desfoque de imagem, a detecção de bordas também funciona pegando cada pixel e modificando-o com base na grade de pixels 3x3 que o circunda. Mas, em vez de simplesmente calcular a média dos nove pixels, o operador de Sobel computa o novo valor de cada pixel fazendo uma soma ponderada dos valores dos pixels vizinhos. E como as bordas entre objetos podem ocorrer tanto na direção vertical quanto na horizontal, você calculará duas somas ponderadas: uma para detectar bordas na direção x e outra para detectar bordas na direção y. Em particular, você usará os dois "kernels" a seguir:

Gx

-1	0	1
-2	0	2
-1	0	1

Gy

-1	-2	-1
0	0	0
1	2	1

Como interpretar esses kernels? Resumidamente, para cada um dos três valores de cor de cada pixel, calcularemos dois valores, x `Gx` e y `Gy`. Para calcular `Gx` o valor do canal vermelho de um pixel, por exemplo, pegaremos os valores vermelhos originais dos nove pixels que formam uma caixa 3x3 ao redor do pixel, multiplicaremos cada um deles pelo valor correspondente no `Gx` kernel e somaremos os valores resultantes.

Por que esses valores específicos para o kernel? Na `Gx` direção, por exemplo, multiplicamos os pixels à direita do pixel alvo por um número positivo e os pixels à esquerda do pixel alvo por um número negativo. Ao somarmos os valores, se os pixels à direita tiverem uma cor semelhante à dos pixels à esquerda, o resultado será próximo de 0 (os números se cancelam). Mas se os pixels à direita forem muito diferentes dos pixels à esquerda, o valor resultante será muito positivo ou muito negativo, indicando uma mudança de cor que provavelmente resulta de uma fronteira entre objetos. Um argumento semelhante se aplica ao cálculo de bordas na `y` direção.

Usando esses kernels, podemos gerar um valor `Gx` de x e um `Gy` valor de y para cada um dos canais vermelho, verde e azul de um pixel. Mas cada canal só pode assumir um valor, não dois: portanto, precisamos de alguma forma de combinar `Gx` x e y `Gy` em um único valor. O algoritmo do filtro de Sobel combina x `Gx` e y `Gy` em um valor final calculando a raiz quadrada de y `Gx^2 + Gy^2`. E como os valores dos canais só podem assumir valores inteiros de 0 a 255, certifique-se de que o valor resultante seja arredondado para o inteiro mais próximo e limitado a 255!

E quanto ao tratamento de pixels na borda ou no canto da imagem? Existem muitas maneiras de lidar com pixels na borda, mas para os propósitos deste problema, pediremos que você trate a imagem como se houvesse uma borda preta sólida de 1 pixel ao redor da borda da imagem: portanto, tentar acessar um pixel além da borda da imagem deve ser tratado como um pixel preto sólido (valores de 0 para cada uma das cores vermelha, verde e azul). Isso efetivamente ignorará esses pixels de nossos cálculos de `Gx` e `Gy`.

Especificação

Implemente as funções de `helpers.c` forma que o usuário possa aplicar filtros de escala de cinza, reflexo, desfoque ou detecção de bordas às suas imagens.

- A função `grayscale` deve pegar uma imagem e transformá-la em uma versão em preto e branco da mesma imagem.
- A `reflect` função deve receber uma imagem e espelhá-la horizontalmente.
- A `blur` função deve pegar uma imagem e transformá-la em uma versão desfocada da mesma imagem.
- A `edges` função deve receber uma imagem e destacar as bordas entre os objetos, de acordo com o operador de Sobel.

Você não deve modificar nenhuma das assinaturas de função, nem deve modificar nenhum outro arquivo além de `helpers.c`.

Entendimento

Vamos agora analisar alguns dos arquivos fornecidos como código de distribuição para entender o que eles contêm.

`bmp.h`

Abra o arquivo `bmp.h` (clicando duas vezes nele no explorador de arquivos, por exemplo) e dê uma olhada.

Você verá as definições dos cabeçalhos que mencionamos (`BITMAPINFOHEADER` e `BITMAPFILEHEADER`). Além disso, esse arquivo define `BYTE` , `DWORD` , `LONG` e `WORD` , tipos de dados normalmente encontrados no mundo da programação para Windows. Observe como eles são apenas aliases para tipos primitivos com os quais você (esperamos) já esteja familiarizado. Parece que `BITMAPFILEHEADER` e `BITMAPINFOHEADER` fazem uso desses tipos.

Talvez o mais importante para você seja que este arquivo também define um `struct` parâmetro chamado `RGBTRIPLE` `encapsulate`, que, de forma bastante simples, "encapsula" três bytes: um azul, um verde e um vermelho (a ordem, lembre-se, em que esperamos encontrar triplas RGB no disco).

Por que esses structs são `struct` úteis? Bem, lembre-se de que um arquivo é apenas uma sequência de bytes (ou, em última análise, bits) no disco. Mas esses bytes geralmente são ordenados de tal forma que os primeiros representam algo, os próximos representam outra coisa e assim por diante. Os "formatos de arquivo" existem porque o mundo padronizou o significado de cada byte. Agora, poderíamos simplesmente ler um arquivo do disco para a RAM como um grande array de bytes. E poderíamos simplesmente lembrar que o byte em

`array[i]` representa uma coisa, enquanto o byte em `array[j]` representa outra. Mas por que não dar nomes a alguns desses bytes para que possamos recuperá-los da memória mais facilmente? É exatamente isso que os structs em `std::string` `bmp.h` nos permitem fazer. Em vez de pensar em um arquivo como uma longa sequência de bytes, podemos pensar nele como uma sequência de `struct` structs.

`filter.c`

Agora, vamos abrir o `filter.c` arquivo. Ele já foi escrito para você, mas há alguns pontos importantes que vale a pena observar aqui.

Primeiro, observe a definição de ` `filters` on` na linha 10. Essa string informa ao programa quais são os argumentos de linha de comando permitidos: `b` `--display`, `e` `--display`, `g` `--display`, `--display` e `r` `--display`. Cada um deles especifica um filtro diferente que podemos aplicar às nossas imagens: desfoque, detecção de bordas, escala de cinza e reflexão.

As próximas linhas abrem um arquivo de imagem, verificam se é realmente um arquivo BMP e leem todas as informações de pixel em uma matriz 2D chamada `image`.

Desça até a `switch` declaração que começa na linha 101. Observe que, dependendo da `filter` nossa escolha, uma função diferente é chamada: se o usuário escolher `filter` `b`, o programa chama a `blur` função; se `filter` `e`, então ` `edges` filter` é chamada; se `filter` `g`, então `grayscale` `filter` é chamada; e se `filter` `r`, então ` `reflect` filter` é chamada. Observe também que cada uma dessas funções recebe como argumentos a altura da imagem, a largura da imagem e a matriz 2D de pixels.

Estas são as funções que você implementará (em breve!). Como você pode imaginar, o objetivo de cada uma dessas funções é editar a matriz 2D de pixels de forma a aplicar o filtro desejado à imagem.

As linhas restantes do programa pegam o resultado `image` e o gravam em um novo arquivo de imagem.

`helpers.h`

Em seguida, dê uma olhada em `helpers.h`. Este arquivo é bem curto e fornece apenas os protótipos das funções que você viu anteriormente.

Aqui, observe que cada função recebe `image` como argumento uma matriz bidimensional chamada `x`, onde `x` `image` é uma matriz com `height` várias linhas, e cada linha é, por sua vez, outra matriz com `width` vários `RGBTRIPLE` pixels. Portanto, se `x` `image` representa a imagem inteira, então `x` `image[0]` representa a primeira linha e `x` `image[0][0]` representa o pixel no canto superior esquerdo da imagem.

`helpers.c`

Agora, abra o arquivo `helpers.c`. É aqui que a implementação das funções declaradas em `functions` `helpers.h` deve ser feita. Mas observe que, neste momento, as implementações estão faltando! Esta parte depende de você.

`Makefile`

Finalmente, vamos analisar o arquivo ` `Makefile`.bashrc`. Este arquivo especifica o que deve acontecer quando executamos um comando de terminal como ` `make filter`.bashrc`. Enquanto os programas que você pode ter escrito antes se limitavam a um único arquivo, o `.bashrc` `filter` parece usar vários arquivos: `.bashrc`, `.bash_profile` `filter.c` e ` `helpers.c`.bash_profile`. Portanto, precisamos informar `make` como compilar este arquivo.

Experimente compilar `filter` você mesmo acessando seu terminal e executando o seguinte comando:

```
$ make filter
```

Em seguida, você pode executar o programa digitando o seguinte comando:

```
$ ./filter -g images/yard.bmp out.bmp
```

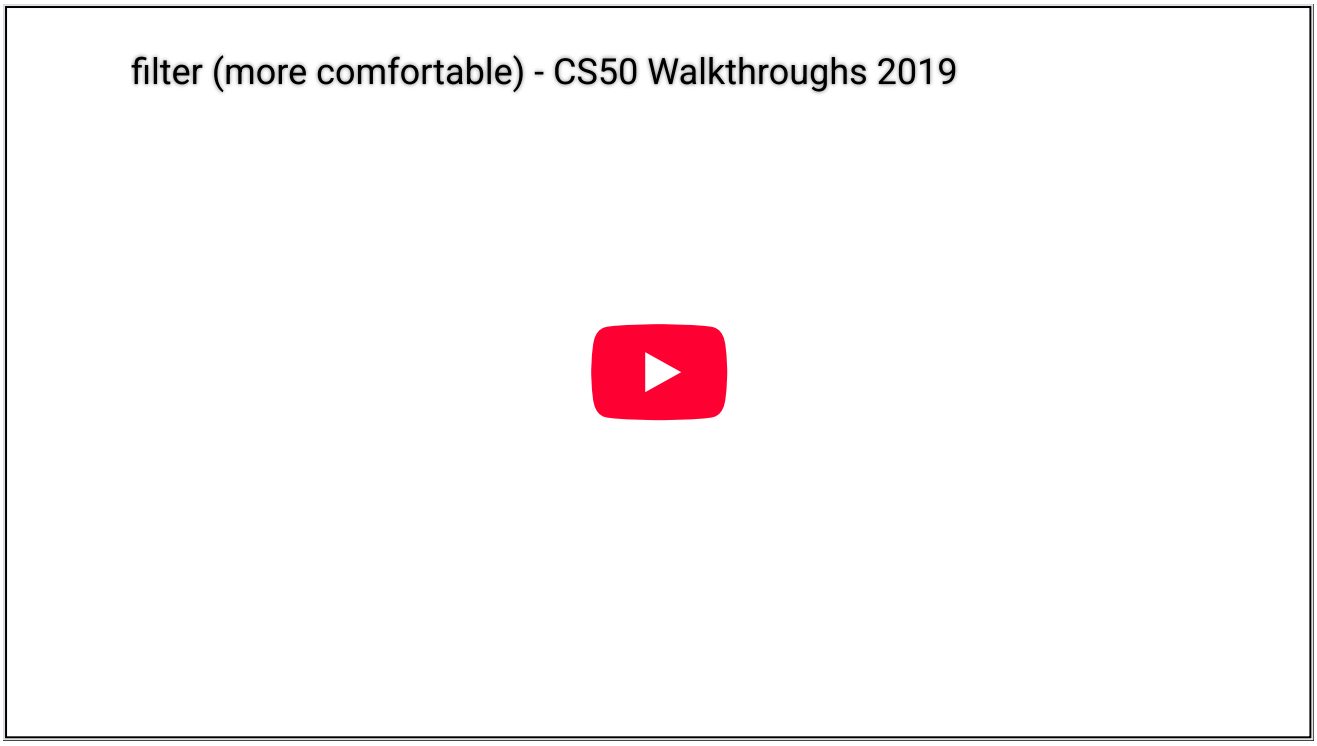
que pega a imagem em `images/yard.bmp`, e gera uma nova imagem chamada `out.bmp` após processar os pixels pela `grayscale` função. `grayscale` No entanto, ainda não faz nada, então a imagem de saída deve ser igual à imagem original do quintal.

Dicas

- `rgbtRed` Os valores dos componentes x , `rgbtGreen` y e z de um pixel `rgbtBlue` são todos inteiros, portanto, certifique-se de arredondar quaisquer números de ponto flutuante para o inteiro mais próximo ao atribuí-los a um valor de pixel!

Passo a passo

Observe que esta lista de reprodução contém 5 vídeos.



Como testar

Não se esqueça de testar todos os seus filtros nos arquivos bitmap de exemplo fornecidos!

Correção

```
check50 cs50/problems/2025/x/filter/more
```

Estilo

```
style50 helpers.c
```

Como enviar

```
submit50 cs50/problems/2025/x/filter/more
```