Pset 5: Ciência Forense

Tenha certeza de que seu código é bem comentado
de forma que a funcionalidade seja aparente apenas pela leitura dos
comentários.

Objetivos.

- Familiarizá-lo com I/O de arquivos.
- Torná-lo mais confortável com estruturas de dados, hexadecimais, e ponteiros.
- Introduzir você a professores de Harvard.
- Ajudar o Coronel Mostarda.

Leitura recomendada.

- Seções 21 a 26, 31, 32, 35 e 40 de http://informatica.hsw.uol.com.br/programacao-em-c.htm
- Seções 1 a 3 de `http://en.wikipedia.org/wiki/BMP_file_format`
- Seções 1, 2 e 6 de `http://en.wikipedia.org/wiki/Hexadecimal`
- Seções 1 a 5 e 7 de `http://en.wikipedia.org/wiki/Jpg`

diff hacker5.pdf hacker5.pdf.

- A edição Hacker desafia você a reduzir (e ampliar) BMPs.

Honestidade Acadêmica.

Todo o trabalho feito no sentido do cumprimento das expectativas deste curso deve ser exclusivamente seu, a não ser que a colaboração seja expressamente permitida por escrito pelo instrutor do curso. A colaboração na realização de Psets não é permitida, salvo indicação contrária definida na especificação do Set.

Ver ou copiar o trabalho de outro indivíduo do curso ou retirar material de um livro, site ou outra fonte, mesmo em parte e apresentá-lo como seu próprio constitui desonestidade acadêmica, assim como mostrar ou dar a sua obra, mesmo em parte, a um outro estudante. Da mesma forma é desonestidade acadêmica apresentação dupla: você não poderá submeter o mesmo trabalho ou similar a este curso que você enviou ou vai enviar para outro. Nem poderá fornecer ou tornar as soluções disponíveis para os Psets para os indivíduos que fazem ou poderão fazer este curso no futuro.

Você está convidado a discutir o material do curso com os outros, a fim de melhor compreendê-lo. Você pode até discutir sobre os Psets com os colegas, mas você não pode compartilhar o código. Em outras palavras, você poderá se comunicar com os colegas em Português, mas você não pode comunicar-se em, digamos, C. Em caso de dúvida quanto à adequação de algumas discussões, entre em contato com o instrutor.

Você pode e deve recorrer à Web para obter referências na busca de soluções para os Psets, mas não por soluções definitivas para os problemas. No entanto, deve-se citar (como comentários) a origem de qualquer código ou técnica que você descubra fora do curso.

Todas as formas de desonestidade acadêmica são tratadas com rigor.

Licença.

Notas:

Seu trabalho neste Pset será avaliado em três quesitos principais:

Exatidão. Até que ponto o seu código é consistente com as nossas especificações e livre de bugs?
Design. Até que ponto o seu código é bem escrito*(*escrito claramente, funcionando de forma eficiente, elegante, e / ou lógica)?
Estilo. Até que ponto o seu código é legível (comentado e indentado, com nomes de variáveis apropriadas)?

Começando.

☑ Apenas mais alguns checkboxes!

☐ Abra o Terminal e crie um diretório dentro de cc50 chamado pset5. Extraia todos os arquivos de hacker5.zip dentro desse diretório. Agora liste os arquivos dentro de hacker5/, você deverá ver o seguinte:

```
bmp/  jpg/  questions.txt
```

Como você vê, a maioria do seu trabalho para esse Pset será organizado dentro de dois subdiretórios. Vamos começar.

☐ Se você já viu um wallpaper padrão do Windows XP (pense em colinas verdejantes e céu azul), então você já viu um BMP. Se você já olhou para uma página da web, você provavelmente já viu um GIF. Se você já olhou para uma foto digital, você provavelmente já viu um JPEG. Se você já tirou uma screenshot em um Mac, você provavelmente já viu um PNG. Leia um pouco sobre os formatos de arquivo BMP, GIF, JPEG, PNG.[1] Depois, em `pset5/questions.txt`, responda as questões abaixo.

0. Quantas cores diferentes cada formato suporta?
1. Qual destes formatos suporta animação?
2. Qual é a diferença entre a compressão com perdas e sem perdas?
3. Qual destes formatos é comprimido com perdas?

☐ Se você quiser, dê uma olhada no artigo do MIT abaixo (em inglês).

```
http://cdn.cs50.net/2010/fall/psets/5/garfinkel.pdf
```

Se você não entender nada, não se preocupe pois a linguagem é bastante técnica. Tente, então, responder as seguintes perguntas em `pset5/questions.txt`, utilizando os conceitos que vimos em sala de aula, o artigo do MIT ou outras fontes que você encontrar na internet.

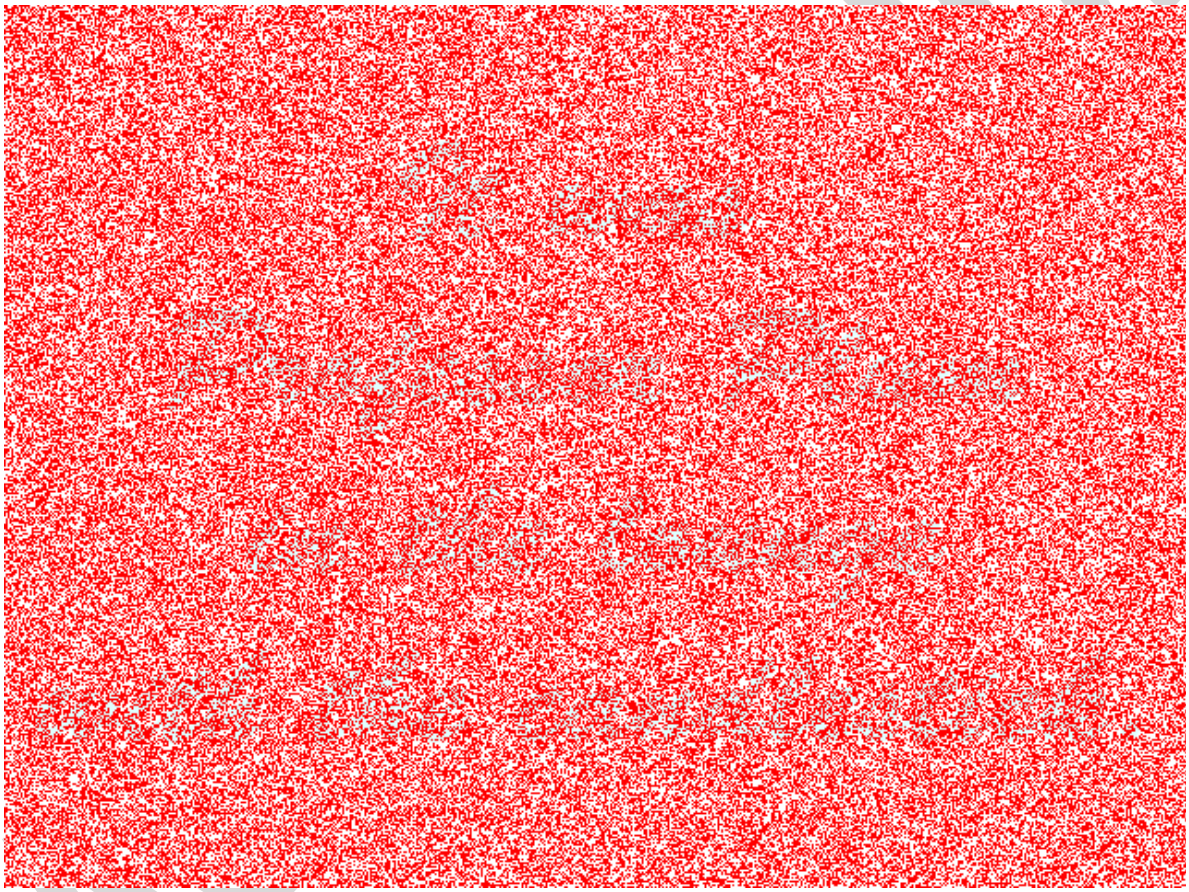
4. O que acontece, tecnicamente falando, quando um arquivo é removido de um sistema de arquivos FAT?
5. O que alguém como você pode fazer para garantir (com alta probabilidade) que os arquivos que você excluir não poderão ser recuperados?

[1] Para isso você é bem-vindo a consultar o Google, a Wikipedia, um amigo, ou qualquer outra pessoa, desde que suas palavras sejam, em última análise, suas próprias!

Whodunit.

☐ Bem-vindo à Mansão Tudor. O seu anfitrião, o Coronel Mostarda, teve um fim prematuro. Para ganhar esse jogo, você deve determinar a resposta a estas três perguntas: Quem cometeu o crime? Onde? E com que arma?

Infelizmente para você (embora, infelizmente, ainda mais para o Coronel Mostarda), a única pista que você tem é um arquivo BMP de 24-bits chamado `clue.bmp`, mostrado na foto abaixo, que o Coronel Mostarda conseguiu colocar no seu computador em seus momentos finais. Uma mensagem dele para você está escondida dentro do "ruído" vermelho desse arquivo.

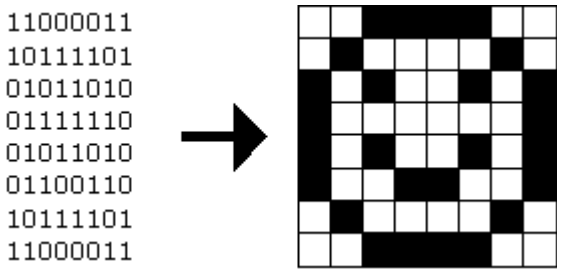


Há muito tempo você jogou fora aquele pedaço de plástico vermelho da sua infância que resolvia esse mistério, e por isso você deve agora atacar o problema como um cientista da computação.

Mas, primeiro, alguma ajuda.

☐ Talvez a maneira mais simples de se representar uma imagem é com uma grade de pixels (pontos), cada um dos quais tendo uma cor diferente. Para imagens em preto-e-branco,

precisamos de 1 bit por pixel: 0 poderia representar preto e 1 poderia representar branco, como mostrado abaixo.[1]



Pensando assim uma imagem é somente um bitmap (um mapa de bits). Para ter imagens mais coloridas, você simplesmente precisa usar mais bits por pixel. Em um formato de arquivo que suporta cores de 8 bits (como o GIF), cada pixel tem exatamente 8 bits. Um formato de arquivo (como BMP, JPEG ou PNG) que suporta cores de 24 bits usa 24 bits por pixel.[2]

Um arquivo BMP de 24 bits como o do Coronel Mostarda usa 8 bits para indicar a quantidade de vermelho, 8 bits para indicar a quantidade de verde e 8 bits para indicar a quantidade de azul na cor de um pixel. Se você já ouviu falar do sistema RGB, é exatamente disso que estamos falando: Red (vermelho), Green (verde), Blue (azul).

Se os valores R, G e B de um certo pixel em um BMP são, digamos, 0xff, 0x00 e 0x00 (em hexadecimal), esse pixel é puramente vermelho, já que 0xff (também conhecido como 255 em decimal) significa "um monte de vermelho", e os dois 0x00 significam "nenhum verde" e "nenhum azul". A partir disso já podemos deduzir que o arquivo do Coronel Mostarda tem claramente um monte de pixels com os valores RGB citados acima. Mas também tem alguns com outros valores.

De forma interessante, XHTML e CSS (línguas utilizadas para escrever páginas da web) modelam cores da mesma maneira. Para mais "códigos" RGB, consulte a URL abaixo.

```
http://www.w3schools.com/html/html_colors.asp
```

Agora vamos ficar um pouco mais técnicos. Lembre-se que um arquivo é apenas uma sequência de bits, organizados de alguma forma. Um arquivo BMP de 24-bits é essencialmente apenas uma seqüência de bits, onde (quase) todos os blocos de 24 bits servem para representar a cor de algum pixel. Mas um arquivo BMP também contém alguns "metadados", informações sobre, por exemplo, a altura e a largura da imagem. Os metadados são armazenados no início do arquivo dentro de duas estruturas de dados geralmente chamadas de "headers" (não confunda com os headers de C).[3] O primeiro desses headers, chamado `BITMAPFILEHEADER`, possui 14 bytes de comprimento. (Lembre-se que 1 byte equivale a 8 bits.) O segundo desses headers, chamado `BITMAPINFOHEADER`, possui 40 bytes de comprimento. Imediatamente após esses headers você

[1] Imagem adaptada de http://www.brackeen.com/vga/bitmaps.html

[2] BMPs na realidade suportam cores de 1, 4 , 8, 16, 24 ou 32 bits.

[3] Esses headers de imagens vem evoluindo ao longo do tempo. Esse Pset só espera que você lide com a versão 4.0 (mais recente) do formato BMP da Microsoft, que estreou com o Windows 95. Ah, Windows 95.

encontrará o bitmap real: Um array de bytes, onde cada tripla representa a cor de um pixel.[1] No entanto, os arquivos BMP armazenam essas triplas de forma invertida *(*como BGR), com 8 bits para o azul, seguidos de 8 bits para o verde e 8 bits para o vermelho.[2] Se formos converter o smiley de 1-bit de cor acima para um smiley de 24-bits de cor, substituindo o preto para vermelho, armazenaríamos essa imagem da seguinte forma: (`0000ff` significa vermelho e `ffffff` significa branco, destacamos em vermelho todos os `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
```

Nós apresentamos esses bytes de uma forma organizada em 8 colunas, então você pode ver o smiley vermelho se se afastar um pouco da tela (e se esforçar um pouco).

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

Ok, pare! Não prossiga até que você esteja certo que você entendeu porque `0000ff` representa um pixel vermelho em um arquivo BMP de 24-bits.

☐ Ok, vamos passar da teoria à prática. Vá para `pset5/bmp`/. Você vai encontrar um arquivo chamado `smiley.bmp`. Se você visualizar esse arquivo, você vai perceber que ele se parece com o abaixo, embora muito menor (pois possui apenas 8 pixels por 8 pixels).

Abra este arquivo no `xxd`, um "editor hexadecimal", executando o comando abaixo.

```
xxd-c 24 g 3-s 54 smiley.bmp
```


[1] Em BMPs de 1, 4 e 16-bits (não nos de 24 ou 32), há ainda um header adicional após BITMAPINFOHEADER chamado RGBQUAD, um array que define a intensidade de cada uma das cores do arquivo.

[2] Alguns BMPs também armazenam o bitmap inteiro ao contrário, com a primeira linha de uma imagem no final do arquivo BMP. Mas nós armazenamos todos os BMPs desse Pset conforme descrito acima, com a linha superior de cada bitmap no começo e a última no fim.

Você deverá ver algo semelhante ao abaixo; destacamos de novo em vermelho todos os `0000ff`.

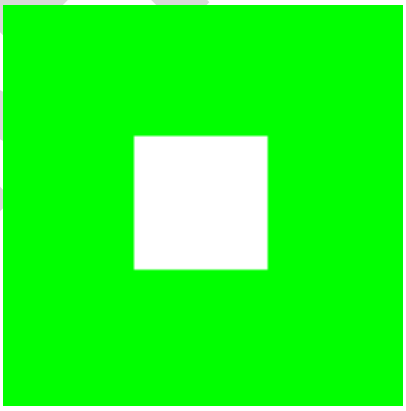
```
0000036: ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff  ........................
000004e: ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff  ........................
0000066: 0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff  ........................
000007e: 0000ff ffffff ffffff ffffff ffffff ffffff ffffff 0000ff  ........................
0000096: 0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff  ........................
00000ae: 0000ff ffffff ffffff 0000ff 0000ff ffffff ffffff 0000ff  ........................
00000c6: ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff  ........................
00000de: ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff  ........................
```

Na coluna mais à esquerda e acima estão endereços de dentro do arquivo ou, equivalentemente, a distância entre o começo dessa linha e o primeiro byte do arquivo, todos eles dados em hexadecimal. Note que `00000036` é `54` em decimal. Então você está olhando para o byte `54` de `smiley.gif`. Lembre-se que os primeiros 14 + 40 = 54 bytes de um BMP de 24-bits de cor são preenchidos com metadados. Se você realmente quer ver os metadados, além do bitmap, execute o comando abaixo.

```
xxd -c 24 -g 3 smiley.bmp
```

Se `smiley.bmp` tivesse caracteres ASCII, você iria vê-los na coluna mais à direita do `xxd` em vez de todos aqueles pontos.

☐ Bem, `smiley.bmp` possui 8 pixels de largura por 8 pixels de altura, e é um BMP de 24-bits de cor (cada um dos pixels é representado por 24 ÷ 8 = 3 bytes). Cada linha (ou "scanline"), portanto, ocupa (8 pixels) x (3 bytes por pixel) = 24 bytes, que acontece de ser um múltiplo de 4. Acontece que BMPs são armazenados de uma forma um pouco diferente se o número de bytes de uma scanline não é um múltiplo de 4. Por exemplo, `small.bmp` é outro BMP de 24-bits de cor, uma caixa verde de 3 pixels de altura por 3 pixels de largura. Se você visualizar o arquivo, verá algo parecido com o abaixo:



Cada scanline de `small.bmp` tem (3 pixels) X (3 bytes por pixel) = 9 bytes, o que não é um múltiplo de 4. Então a scanline é "compensada" com vários zeros, tantos quantos forem precisos para que o comprimento da scanline seja um múltiplo de 4, e esses bytes zerados são chamados de padding. Em outras palavras, de 0 a 3 bytes de padding são necessários para cada scanline em um BMP de 24-bits de cor (entendeu por quê?). No caso de `small.bmp`, 3 bytes cheios de zeros são necessários, uma vez que (3 pixels) X (3 bytes por pixel) + (3 bytes de padding) = 12 bytes, que é de fato um múltiplo de 4.

Para "ver" esse monte de zeros, vá em frente e execute o abaixo.

```
xxd -c 12 -g 3 -s 54 small.bmp
```

Note que estamos usando um valor para `-c` diferente do que usamos em `smiley.bmp` de modo que o `xxd` organize o output em apenas 4 colunas (3 para a caixa verde e 1 para o padding). Você deverá ver um output como o a seguir; destacamos em verde todas as aparições de `00ff00`.

```
0000036: 00ff00 00ff00 00ff00 000000  ............
0000042: 00ff00 ffffff 00ff00 000000  ............
000004e: 00ff00 00ff00 00ff00 000000  ............
```

Agora vamos visualizar `large.bmp` com o `xxd`. Esse arquivo parece idêntico ao `small.bmp` mas é, na verdade, de 12 pixels por 12 pixels, quatro vezes maior. Vá em frente e execute o seguinte, você pode precisar alargar a sua janela para evitar a quebra.

```
xxd -c 36 -g 3 -s 54 large.bmp
```

Você deverá ver o seguinte:

```
0000036: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00  ....................................
000005a: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00  ....................................
000007e: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00  ....................................
00000a2: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00  ....................................
00000c6: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 00ff00 00ff00 00ff00  ....................................
00000ea: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 00ff00 00ff00 00ff00  ....................................
000010e: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 00ff00 00ff00 00ff00  ....................................
0000132: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 00ff00 00ff00 00ff00  ....................................
0000156: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00  ....................................
000017a: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00  ....................................
000019e: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00  ....................................
00001c2: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00  ....................................
```

Note agora que esse BMP não possui padding nenhum! Afinal de contas, (12 pixels) X (3 bytes por pixel) = 36 bytes é realmente um múltiplo de 4.

Saber de tudo isso tem que ser útil!

☐ Ok, `xxd` só nos mostrou os bytes nesses BMPs. Como é que vamos chegar a eles, programando realmente? Bem, `copy.c` é um programa cujo único propósito na vida é criar uma cópia de um BMP, pedacinho por pedacinho. Claro, você poderia simplesmente usar `cp` para isso. Mas `cp` não vai conseguir ajudar o Coronel Mostarda. Esperemos que `copy.c` ajude!

Vá em frente e compile `copy.c` em um programa chamado `copy` (lembra-se como?). Em seguida, execute um comando como o abaixo.

```
./copy smiley.bmp copy.bmp
```

Se você executar `ls`, em seguida, (com o switch apropriado), você verá que `smiley.bmp` e `copy.bmp` são de fato do mesmo tamanho. Vamos verificar novamente para checar se eles são realmente a mesma coisa! Execute o seguinte.

```
diff smiley.bmp copy.bmp
```

Se esse comando não lhe diz nada, os arquivos são realmente idênticos.[1] Sinta-se livre para abrir os dois arquivos e compará-los. Mas `diff` faz uma comparação byte por byte, assim é provável que ele seja um pouco mais minucioso que o seu olho!

Mas, como é que essa cópia aconteceu? Acontece que `copy.c` depende de `bmp.h`. Vamos dar uma olhada. Abra `bmp.h` (com o Nano, por exemplo), e você vai ver definições daqueles headers que mencionamos, adaptados das implementações da própria Microsoft. Além disso, esse arquivo define `BYTE`, `DWORD`, `LONG` e `WORD`, tipos de dados normalmente encontrados no mundo da programação Win32 (Windows). Observe como esses são apenas apelidos para as estruturas de dados primitivas com as quais você (espero) já está familiarizado. Parece que `BITMAPFILEHEADER` e `BITMAPINFOHEADER` fazem uso dessas estruturas. Esse arquivo também define uma `struct` chamada `RGBTRIPLE` que, simplesmente, "encapsula" três bytes: um azul, um verde e um vermelho (a ordem real (BGR) na qual esperamos encontrar essas triplas RGB no disco).

Para que precisamos dessas `structs`? Bem, lembre-se que um arquivo é apenas uma seqüência de bytes (ou, pensando mais primitivamente ainda, bits) no disco. Mas esses bytes estão geralmente organizados de tal forma que os primeiros representam alguma coisa, os próximos representam outra coisa, e assim por diante. Os formatos de arquivos existem porque o mundo vem padronizando quais bytes significam o que. Agora, nós poderíamos simplesmente ler um arquivo do disco para a RAM como um conjunto grande de bytes. Mas nós poderíamos apenas lembrar que o byte na posição `[i]` representa uma coisa, enquanto o byte na posição `[j]` representa outra. Mas por que não dar nomes a conjuntos de bytes que tem algo em comum para que possamos recuperá-los da memória mais facilmente? Isso é precisamente o que as `structs` em `bmp.h` nos permitem fazer. Ao invés de pensar em algum arquivo como uma longa seqüência de bytes, pensamos nele como uma seqüência de `structs`.

[1] Note que alguns programas (como o Photoshop) incluem zeros ao fim de alguns BMPs. Nossa versão de `copy` joga esses zeros fora, por isso não fique muito preocupado se você tentar copiar um BMP (que você baixou ou fez) e descobrir que a cópia é alguns bytes menor do que o original.

Lembre-se que `smiley.bmp` é de 8 pixels por 8 pixels, e por isso deve ocupar o espaço de 14 + 40 + 8 · 8 · 3 = 246 bytes no disco. (Confirme isso se você quiser usando `ls`). Esses bytes estão organizados nos seguintes grupos e levam os seguintes apelidos, de acordo com a Microsoft:

Posição	Tipo	Nome	
0	WORD	bfType	BITMAPFILEHEADER
2	DWORD	bfSize	
6	WORD	bfReserved1	
8	WORD	bfReserved2	
10	DWORD	bfOffBits	
14	DWORD	biSize	BITMAPINFOHEADER
18	LONG	biWidth	
22	LONG	biHeight	
26	WORD	biPlanes	
28	WORD	biBitCount	
30	DWORD	biCompression	
34	DWORD	biSizeImage	
38	LONG	biXPelsPerMeter	
42	LONG	biYPelsPerMeter	
46	DWORD	biClrUsed	
50	DWORD	biClrImportant	
54	BYTE	rgbtBlue	RGBTRIPLE
55	BYTE	rgbtGreen	
56	BYTE	rgbtRed	
57	BYTE	rgbtBlue	RGBTRIPLE
58	BYTE	rgbtGreen	
59	BYTE	rgbtRed	
...			
243	BYTE	rgbtBlue	RGBTRIPLE
244	BYTE	rgbtGreen	
245	BYTE	rgbtRed	

Como essa figura sugere, a ordem importa sim, quando se trata de `structs`. O byte 57 é um `rgbtBlue` (e não, digamos, `rgbtRed`), porque `rgbtBlue` é definido primeiro em `RGBTRIPLE`.[1]

Agora vá em frente e dê uma olhada nas URLs que explicam `BITMAPFILEHEADER` e `BITMAPINFOHEADER`, que podem ser encontradas nos comentários de `bmp.h`. Você está prestes a começar a usar a MSDN (Microsoft Developer Network)!

Dessa vez, ao invés de segurar a sua mão e passear por todo o `copy.c`, vamos fazer-lhe algumas perguntas e deixá-lo se ensinar como o código funciona. Como sempre, o `man` é seu amigo, e agora a MSDN também é. Se você não tem certeza sobre como responder a alguma pergunta, faça uma pesquisa rápida e descubra! Você pode querer usar o recurso abaixo também.

`http://www.cs50.net/resources/cppreference.com/stdio/`

[1] Nosso uso, aliás, do `__attribute__` chamado `__packed__` garante que o `gcc` não tente alinhar as structs de forma que o endereço do primeiro byte de cada struct seja um múltiplo de 4, para não acabarmos com "lacunas" em nossas estruturas que não existem de verdade no disco.

Permita-nos sugerir que você também rode `copy` no `gdb` a medida que responde a essas perguntas. Defina um ponto de interrupção em `main` e caminhe através do programa. Lembre-se que você pode dizer ao `gdb` para começar a executar o programa da seguinte forma (já dentro do prompt do gdb):

```
run smiley.bmp copy.bmp
```

Se você disser ao `gdb` para imprimir os valores de `bf` e `bi` (uma vez que eles forem lidos do disco), você verá uma saída como a seguir. Ousamos dizer que você vai achar isso bastante útil.

```
{bfType = 19778, bfSize = 246, bfReserved1 = 0, bfReserved2 = 0,
  bfOffBits = 54}

{biSize = 40, biWidth = 8, biHeight = -8, biPlanes = 1, biBitCount = 24,
  biCompression = 0, biSizeImage = 192, biXPelsPerMeter = 2834,
  biYPelsPerMeter = 2834, biClrUsed = 0, biClrImportant = 0}
```

Em `~/pset5/questions.txt`, responda cada uma das seguintes perguntas em uma frase ou mais.

6. O que é `stdint.h`?
7. Porque se usa `uint8_t`, `uint32_t`, `int32_t` e `uint16_t` em um programa?
8. Quantos bytes tem um `BYTE`, uma `DWORD`, um LONG e uma WORD, respectivamente?[1]
9. O que (em ASCII, decimal ou hexadecimal) devem ser os dois primeiros bytes de qualquer arquivo BMP?[2]
10. Qual é a diferença entre `bfSize` e `biSize`?
11. O que significa se `biHeight` for negativo?
12. Que campo em `BITMAPINFOHEADER` especifica a cor do BMP (bits por pixel)?
13. Por que `fopen` pode retornar `NULL` em `copy.c:32`?
14. Porque é que o terceiro argumento de `fread` é sempre 1 em nosso código?
15. Qual o valor que `copy.c:69` atribui a `padding` se `bi.biWidth` é `3`?
16. O que é que `fseek` faz?
17. O que é `SEEK_CUR`?

Ok, de volta ao Coronel Mostarda.

☐ Escreva um programa chamado `whodunit` em um arquivo chamado `whodunit.c` que revela as últimas palavras do Coronel Mostarda.

OMG, o quê? Como?

Bem, pense novamente na sua infância quando você segurou aquele pedaço de plástico vermelho sobre mensagens ocultadas da mesma forma.[3] Essencialmente, o plástico fez tudo ficar vermelho, mas de alguma forma revelou as mensagens. Implemente essa mesma idéia em `whodunit`. Como `copy`, seu programa deve aceitar exatamente dois argumentos de linha de comando. E se

[1] Assuma uma arquitetura de 32-bit.

[2] Os primeiros bytes usados para identificar formatos de arquivo são geralmente chamados de "magic numbers".

você executar um comando como o abaixo, uma BMP na qual a mensagem que o Coronel Mostarda escreveu está legível deve ser armazenada em `verdict.bmp`.

```
whodunit clue.bmp verdict.bmp
```

Permita-nos sugerir que você comece a resolução desse mistério, executando o comando abaixo.

```
cp copy.c whodunit.c
```

Hmm... Você pode se surpreender, pois você só precisa escrever muito poucas linhas de código para ajudar o Coronel Mostarda.

Não há nada escondido em `smiley.bmp`, mas sinta-se livre para testar o seu programa em seus pixels de qualquer forma. Apenas porque esse BMP é pequeno e você pode, assim, compará-lo a saída do seu próprio programa com `xxd` durante o desenvolvimento.[1]

Fique tranquilo pois várias soluções são possíveis. Desde que o output do seu programa seja legível (pelo instruto), não importa(m) a(s) sua(s) cor(es), o Coronel Mostarda vai descansar em paz.

☐ Em `~/pset5/questions.txt`, responder à pergunta abaixo.

18. Quem fez isso? E onde? E com que arma?

☐ Ufa, isso foi divertido. Um pouco tarde demais para o Coronel Mostarda, no entanto.

Vamos fazer você escrever mais do que, o que, duas linhas de código? Implemente agora em `resize.c` um programa chamado `resize` que redimensiona BMPs de 24 bits por um fator `n`. Seu programa deve aceitar exatamente três argumentos de linha de comando, onde o primeiro `(f)` deve ser um valor de ponto flutuante entre (0.0, 100.0), o segundo deve ser o nome do arquivo a ser redimensionada e o terceiro o nome do arquivo onde a versão redimensionada vai ficar.

```
Usage: resize f infile outfile
```

Com um programa como esse, poderíamos criar `large.bmp` a partir de `small.bmp` redimensionando o último por um fator 4.0 *(*ou seja multiplicando a sua largura e altura por 4.0), com o seguinte comando.[2]

```
./resize 4.0 small.bmp large.bmp
```

Você pode começar copiando (mais uma vez) `copy.c` e chamando a cópia de `resize.c`. Mas passe algum tempo pensando sobre o que significa redimensionar um BMP, particularmente se `f`

[3] Se você não se lembra de nenhum pedaço de plástico vermelho da sua infância, melhor perguntar para um amigo ou parente sobre a infância deles.

[1] Ou talvez haja uma mensagem escondida em `smiley.bmp` também. Não, não há. Embora talvez haja. Não. Talvez...

está entre (0.0, 1.0)[1,2]. Você decide inteiramente como lidar com a imprecisão e o arredondamento dos valores de ponto flutuante, e como você lida com a perda inevitável de detalhe. Decida qual dos campos em `BITMAPFILEHEADER` e `BITMAPINFOHEADER` você talvez precise modificar. Considere se você terá ou não que adicionar ou subtrair padding nas scanlines.

CSI.[3]

☐ Tudo bem, agora vamos colocar todas as suas novas habilidades a prova.

Antes desse Pset, passei alguns meses tirando fotos de pessoas que eu conheço, que foram salvas por minha câmera digital no formato JPEG em um cartão de 1GB CompactFlash (CF).[4,5] Infelizmente eu não sou muito bom com computadores, e de alguma forma eu excluí todas elas![6] Felizmente, no mundo da informática, "excluir" não significa tanto "excluir" mas na verdade "esquecer". Meu computador insiste que o cartão CF está em branco, mas eu tenho certeza que ele está mentindo para mim.

Escreva um programa em `~/pset5/jpg/` chamado `recover` que recupera as fotos.

Ahm, quê? Como?

Bem, o negócio é o seguinte. Mesmo que JPEGs sejam mais complicadas do que BMPs, JPEGs tem "assinaturas", padrões de bytes que os distinguem de outros formatos de arquivos. Na verdade, a maioria dos JPEGs começa com uma dentre duas sequencias de bytes. Especificamente, os primeiros quatro bytes da maioria dos JPEGs ou são

```
0xff 0xd8 0xff 0xe0
```

ou

```
0xff 0xd8 0xff 0xe1
```

a partir do primeiro byte até o quarto byte, da esquerda para a direita. É bem provável que, se você encontrar um desses padrões de bytes em um disco que foi usado para armazenar fotos (por exemplo o meu cartão CF), eles demarquem o início de um JPEG.[7]

Felizmente, as câmeras digitais tendem a armazenar fotografias em cartões CF de forma contínua, onde cada foto é armazenada imediatamente após a foto tirada anteriormente. Assim, o início de um JPEG normalmente demarca o fim de outro. No entanto, as câmeras digitais geralmente inicializam cartões CF com um sistema de arquivos FAT, cujo "tamanho de bloco" é 512 bytes (B).

---

[2] E nós ainda usamos Photoshop

[1] Você pode assumir que `f` vezes o tamanho da `infile` não poderá exceder 2^{32}-1.

[2] No caso `f`=1.0, o resultado deve ser de fato uma `outfile` com dimensões idênticas às da `infile`.

[3] Computer Science Investigation

[4] É possível que só parte dessa frase seja verdadeira.

[5] Crédito real para as fotos vai para ACM, Dan Armendariz, Eliza Grinnell, Harvard Crimson, Harvard Gazette, NVIDIA, SEAS, Titus Zhang, *etc*.

[6] Essa frase é bastante verdadeira.

A implicação é que essas câmeras apenas escrevem nesses cartões em unidades de 512 B. Uma foto que tem 1 MB (1.048.576 B) ocupa, portanto, 1.048.576 ÷ 512 = 2048 "blocos" em um cartão CF. Mas o mesmo espaço é ocupado por uma foto que tem, digamos, um byte a menos de dimensão *(*1.048.575 B)! O espaço desperdiçado em disco é chamado de "slack space" ou espaço de folga. Investigadores forenses olham, muitas vezes para esse “slack space” para procurar restos de dados suspeitos.

A implicação de todos esses detalhes é que você, o investigador, pode provavelmente escrever um programa que itera sobre uma cópia do meu cartão CF, à procura de assinaturas JPEGs. Cada vez que você encontrar uma assinatura, você pode abrir um novo arquivo para escrita e começar a preencher esse arquivo com bytes copiados do meu cartão CF, fechando o arquivo apenas uma vez que você se deparar com outra assinatura. Além disso, ao invés de ler um byte do meu cartão CF de cada vez, você pode colocar 512 deles de cada vez em um buffer, para serem lidos, em nome da eficiência. Graças ao FAT, você pode saber que as assinaturas JPEGs serão “alinhadas nos blocos”. Ou seja, você só precisa procurar por essas assinaturas nos quatro primeiros bytes de um bloco.

Perceba, é claro, que JPEGs podem ocupar vários blocos contínuos. Caso contrário, nenhum JPEG poderia ser maior do que 512 B. Mas o último byte de um arquivo JPEG pode não estar no byte final de um bloco. Lembre-se da existência do slack space. Mas não se preocupe. Porque esse cartão CF era novíssimo quando eu comecei a tirar fotos, então é bem provável que ele tenha sido "zerado" (preenchido com 0s) pelo fabricante, por isso qualquer espaço de folga será preenchido com 0s. Está tudo bem se os 0s acabarem nos JPEGs que você recuperar, as imagens ainda deverão ser visíveis.

Bem, eu só tenho um único cartão CF, mas tem vários de vocês! E assim eu fui em frente e criei uma "imagem forense" do cartão, armazenando o seu conteúdo, byte após byte, em um arquivo chamado `card.raw` que veio junto com esse documento. De modo que você não perca tempo iterando ao longo de milhões de 0s desnecessariamente, eu só criei uma imagem a partir dos primeiros 6,1 MB do cartão CF. Para ler os dados desse arquivo você provavelmente utilizará um código como o seguinte:

```
FILE *fp = fopen("/home/cc50/pset5/jpg/card.raw", "r");
```

No fim você deve encontrar 50 JPEGs, cada um dos quais tem entre 8 KB e 200 KB de tamanho, mais ou menos.

Para economizar espaço, você não precisa mandar as imagens que encontrar nem o arquivo card.raw na hora de enviar esse pset pois eu já tenho a minha cópia do arquivo e, se o seu programa funcionar, conseguirei recuperar as fotos sem que você as mande!

Repare ainda que `~/pset5/jpg/` está vazio a não ser por card.raw. Você é quem deve criar, pelo menos, um arquivo `recover.c` para esse programa (deixamos para você decidir como compilá-lo). Para simplificar, você pode embutir o caminho para `card.raw` em seu programa, que não

[7] Você poderia encontrar esses padrões em algum disco puramente por acaso, portanto, a recuperação de dados não é uma ciência exata que funciona com 100% de chance.

precisa aceitar nenhum argumento de linha de comando. Quando executado, porém, seu programa deve recuperar cada um dos JPEGs de `card.raw`, armazenando cada um como um arquivo no diretório de trabalho atual. Seu programa deve numerar os arquivos de saída nomeando cada um da forma `###.jpg`, onde `###` é um número decimal de três dígitos de `000` para cima (fique amigo de `sprintf`). Você não precisa tentar recuperar os nomes originais dos JPEGs. Para verificar se o seu programa está cuspindo os JPEGs corretamente, simplesmente dê dois cliques no arquivo e dê uma olhada.[1] Se todas as fotos aparecerem intactas, a operação foi provavelmente um sucesso!

Mas não fique tão animado! Bem provavelmente, os JPEGs cospidos pelo seu programa na(s) primeira(s) veze(s) não serão corretos (se você abri-los e não ver nada, eles provavelmente não estão corretos!). E também é provável que você queira executar o comando abaixo para apagar todos os JPEGs em seu diretório de trabalho atual.

```
rm *.jpg
```

Se você prefere não ser solicitado a confirmar cada exclusão, execute o comando abaixo:

```
rm -f *.jpg
```

Basta ter cuidado com esse switch `-f`, que "força" todas as exclusões sem misericórdia.

Verificação.

Antes de considerar esse Set de Problemas feito, melhor perguntar a si mesmo estas perguntas e depois voltar e melhorar o seu código conforme o necessário! Não considere isso uma lista de todas as nossas expectativas, mas apenas uma lista de alguns lembretes úteis. Esse documento inteiro representa a lista completa! Para ser claro, considere as perguntas abaixo como retóricas. Não há necessidade de respondê-las por escrito para nós, pois todas as suas respostas devem ser "sim!"

- ☐ Você preencheu `questions.txt` com respostas a todas as perguntas?
- ☐ O BMP que sai de `whodunit` é legível?
- ☐ O `resize` aceita três e apenas três argumentos de linha de comando?
- ☐ O `resize` funciona para qualquer f em [0.0, 100.0]?
- ☐ O `resize` atualiza `bfSize`, `biHeight`, `biSizeImage` e `biWidth` do arquivo de saída corretamente?
- ☐ O `resize` adiciona ou remove padding conforme o necessário?
- ☐ O `recover` cospe 50 JPEGs quando executado? Todos os 50 podem ser visualizados?
- ☐ Ao rodar o `recover`, o nome dos JPEGs é `###.jpg`, onde `###` é um número de três dígitos de `000` a `049`?
- ☐ Todos os seus arquivos estão onde eles deveriam estar em `/hacker5`/?

Como sempre, se você não pode respondeu "sim" a uma ou mais das perguntas acima é porque você está tendo alguns problemas então acesse cc50.com.br/forum!

[1] Fora do terminal ;D