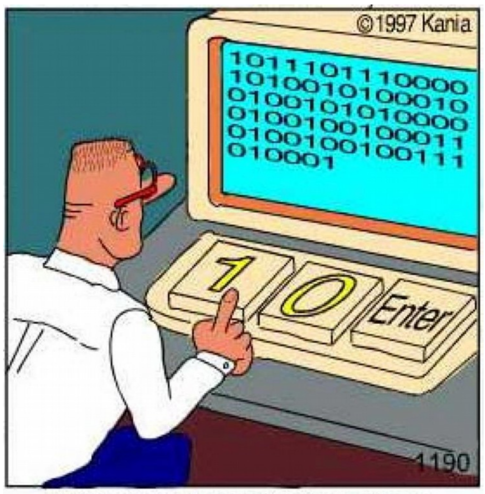
CS 218 – Assignment #11, Part A

Purpose: Become more familiar with operating system interaction, file input/output operations, and file I/O buffering.
Due: Monday (3/20)
Points: 250 (grading will include functionality, documentation, and coding style)

Assignment:

Write a program that will create a thumbnail[1] (small, 300x200 image). The program will read the source file and output file names from the command line. The program must perform basic error checking on the command line file names. For example, to create a thumbnail image named **newSmImg.bmp**, the command line might be as follows:

```
./makeThunb image.bmp thumb.bmp
```



Real programmers code in binary.

The required functions include the following:

- Function ***getImageFileNames()*** to read and verify the file names from the command line. The function should check the file names by attempting to open the files. If both files open, the function should return TRUE and the file descrptors. The function shoudl verify the ".bmp" extension. If there is an error, the function should display an appropriate error message and return FALSE.
- Function ***setImageInfo()*** to read, verify, and write the header information. The function must check the signature, color depth, and bitmap size consistency. Refer to the BMP File Format section. If the information is correct, the entire header should be updated to reflect the thumbnail dimensions (width and height) and written to the output file, the width and height should be returned to the main, and the function should return TRUE. If there is an error, the function should display an appropriate error message (pre-defined) and return FALSE.
- Function ***readRow()*** should return one row of pixels. If a row can be returned, the function should return the row and return TRUE. If a row can not be returned (because there is no more data) the function should return FALSE. If there is a read error, the function should dispaly an appropriate error message (pre-defined) and return FALSE. To ensure overall efficiency, the function ***must*** perform buffered input with a buffer size of BUFF_SIZE (originally set to 500,000). *Note*, this will be changed for part B.
- Function ***writeRow()*** should write one row of pixels to the output file. If successful, the function should return TRUE. If there is a write error, the function should dispaly an appropriate error message (pre-defined) and return FALSE. This function is not required to buffer the output. As such, each row may be written directly to the output file.

Provided Main

The provided main program calls the various functions. ***The provided main program must not be changed in any way.*** All your functions must be in a separate, independently assembled source file. *Note,* to help with the initial program testing, it might be best to temporarily skip (comment out) the calls to the image manipulation statements (in the main).

1 For more information, refer to: https://en.wikipedia.org/wiki/Thumbnail

Submission:

When complete, submit:

- A copy of the functions **source file** via the class web page (assignment submission link) by class time. ***Assignments received after the due date/time will not be accepted.***

Assembly:

An assembly and link script file (asm11) is provided. *Note*, the script file will require execute privilege (i.e., **`chmod +x asm11`**). To assemble/link script can be executed as follows:

```
ed-vm% ./asm11 makeThumb a11tmp
```

Assuming the main file is named **`makeThumb.cpp`** and the functions file is named **`a11tmp.asm`**.

Debugging -> Command Line Arguments

When debugging a program that uses command line arguments, the command line arguments must be entered after the debugger has been started. The debugger is started normally (ddd <program>) and once the debugger comes up, the initial breakpoint can be set. Then, when you are ready to run the program, enter the command line arguments. This can be done either from the menu (Propgram -> Run) or on the GDB Console Window (at bottom) by typing **`run <commandLineArguments>`** at the (gdb) prompt (bottom window).

Testing

A script file to execute the program on a series of pre-defined inputs will be provided. *Note*, please follow the I/O examples (and use the provided error strings). The test utility should be downloaded and extracted into an empty directory and the program executable placed in that directory. The test script can be executed as follows:

```
ed-vm$ ./a11tst makeThumb
```

Assuming the executable file is named **`makeThumb`**, the test script compares the program output to pre-defined expected output (based on the example I/O). *Note*, the script file will require execute privilege (i.e., **`chmod +x a11tst`**). The test utility is only useful when the program is very close to working correctly. Using the test script on an incomplete or partially working program will not provide meaningful information.

Buffering

Since many image files can be very large, it would be difficult to pre-allocate enough memory to hold a complete large image. Further, that memory would be mostly unused for processing smaller images. To address this, the program will perform *buffered input*. Specifically, the program should read a full buffer of BUFF_SIZE bytes (originally set to 500,000). *Note*, this will be changed for part B. From that large buffer, the ***readRow()*** function would return one row (width *3 bytes). The next call to the ***readRow()*** function would return the next row. As such, the ***readRow()*** function must keep track of where it is in the buffer. When the buffer is depleted, the readRow() function must re-fill the buffer by reading BUFF_SIZE bytes from the file. Only after the last row has been returned, should the ***readRow()*** function return a NOSUCCESS status.

BMP File Format[2]

The BMP format supports a wide range of different types, including possible compression and 16, 24, and 32 bit color depths. For our purposes, we will only support uncompressed, 24-bit color BMP files. Under these restrictions, the header (or initial information) in the file is a 138 byte block containing a series of data items as follows:

Size	Description
2 bytes	Signature. Must be "BM". *Note*, must ensure is "BM" for this assignment.
4 bytes	File size (in bytes).
4 bytes	Reserved.
4 bytes	Size of header. Varies based on compression and color depth. For an uncompressed, 24-bit color depth, the header is 54 bytes.
4 bytes	Offset to start of image data in bytes. May vary based on compression and color depth.
4 bytes	Image width (in pixels).
4 bytes	Image height (in pixels).
2 bytes	Number of planes in image (typically 1).
2 bytes	Number of bits per pixel. Typically 16, 24, or 32. *Note*, must ensure is 24 for this assignment.
4 bytes	Compression Type (0=none). *Note*, must ensure is 0 for this assignment.
4 bytes	Size of image in bytes (may vary based on compression types).
100 bytes	Miscellanious (not used for uncompressed, 24-bit color depth).

Since the remaining parts of the program will only work for uncompressed, 24-bit color depth, these must be verified before the program can perform image manipulations. Specifically, the following items should be verified:

- File signature is valid (must be "BM" for signature).
- Color depth is 24-bits.
- Image data is not compressed (i.e., compression type must be 0).
- File bitmap block size consistency (file size = size of image in bytes + header size).

Appropriate error message strings are provided in the functions template. Once these are verified, the header information should be written to the output file.

24-bit Color Depth

For 24-bit color depth, the pixel color is stored as three bytes, a red value (0-255), a green value (0-255), and a blue value (0-255). The color values are stored in that order. Thus, each pixel is 3 bytes (or 24-bits). So, for a 800 pixel row, there are 2,400 bytes (800 * 3).

The main sets a row value maximum as 5,000 pixels (or 15,000 bytes). The provided main will perform this verification and display an appropriate error message as required.

2 For more information, refer to: http://en.wikipedia.org/wiki/BMP_file_format

Example Executions:

The following execution, which includes some errors, will read file **`image0.bmp`**, create the thumbnail image, and place the output image in a file named **`tmp.bmp`**.

```
ed-vm% ./makeThumb
Usage: ./makeThumb <inputFile.bmp> <outputFile.bmp>
ed-vm%
ed-vm% ./makeThumb img0.bp thm.bmp
Error, invalid source file name.  Must be '.bmp' file.
ed-vm%
ed-vm% ./ makeThumb none.bmp tmp. thm
Error, opening input file.
ed-vm%
ed-vm%  ./ makeThumb img0.bmp thm.bmpp
Error, invalid output file name.  Must be '.bmp' file.
ed-vm%
ed-vm% ./ makeThumb img0.bmp thm.bmp
ed-vm%
```

Note, a series of image files, including **`img0.bmp`**, are provided. However, the program will work on any uncompressed, 24-bit BMP format file.

Example Ouptut Images

The following table provides some examples of the expected final output.

Example Output		
Original Image `img1.bmp` Original Image Size: 5.8 MB		Image thumbnail `tmpImg4.bmp` Original Image Size: 180.1 KB