University of Cape Town
Department of Computer Science

CSC3022F/CSC3023F
VolImage: A Volumetric Image Manipulation Tool using raw pointers

February, 2020

Your task is to build a volumetric image tool, which can load a "stack" of 2D images and manipulate them as a group. Volumetric images are produced by many instruments, including radio astronomy telescopes and MRI machines used in medical imaging.

To simplify this problem, you will only have to load a sequence of 2D unsigned byte (values from 0..255) buffers, one for each image slice. These will be provided for you, as a sequence of binary (".raw") files with a header file that provides the image dimensions. So, for example, an image sequence called "scan" would have a file "scan.dat" (the header file) and a sequence of images numbered as follows: "scan0.raw", "scan1.raw", "scan2.raw" etc. You can determine the number of images to open and read by first opening the header file. Your application will be invoked as follows on the command line:

```
volimage <imageBase> [-d i j output_file_name] [-x i output_file_name]
```

where 'volimage' is the name you should give your executable, <imageBase>is the the prefix for the file sequence ("scan" in our example) (the angle brackets are a convention to show the argument must be provided). There are then two optional command line arguments (indicated by the [] brackets):

1. -d i j output_file_name: compute a difference map between images i and j, and write this out to file (note: we index files from 0, see below).
2. -x i output_file_name: extract and write the slice with number i and write this out to file.

You will have to parse the command line using the argc and argv values passed into main(). Remember, argc tells you the number of all items on the command line (the application name included - so "volimage scan -x 5 exoutput" would cause argc to be 5; the argv array would contain simple C-strings for each of these. argv[0] is always the application name, and argv[1] the first argument. Note: you can turn a string, such as "5", into an integer by reading into an int variable from an istringstream constructed with the string source.

If you provide no optional flags, the application should go ahead and build the internal representation and then exit after ensuring memory is correctly cleaned up, and print the following to the console:

Number of images: (int)
Number of bytes required: (int)

The number of bytes required will be the number of bytes required to store the image data ONLY (including pointers).

If you request an operation, the same information should be displayed, along with a message stating what the operation achieved (you can determine the output message format - this is not auto-marked).

Header format

The header is a file which contains 3 integers:

width height number_images

The with and height provides the pixel with and height of the images and number_images tells us how many images are in the sequence. Note that white space is used to separate these numbers.

Operations

1. extract: retrieve the slice with the index given and write this to an output sequence called "output". You will need to generate one header, "output.dat", with the width, height and number of slices set to 1. You will also need to write out the 2D unsigned byte array as "output.raw".
2. difference map: computes the difference map between "slices" i and j. Every output pixel at coordinate (r,c) is computed as follows:

```
(unsigned char)(abs((float)volume[i][r][c] - (float)volume[j][r][c])/2)
```

You can do a difference map or extraction, but not both during the same invocation of volimage.

[extra credit: define a new operation [-g i], which extracts an image along row i of the volume, across all slices, and write this out as above. As noted above, this would be an operation that cannot be done with others]

Grading: if you do everything correctly and the program passes all tests, you will score 95%. To get the extra 5% you need to do the "extra credit" requirement as noted above.

Implementation details

You *must* use a `vector<unsigned char **>` to store your image slices (see class outline below). When building the vector, you can create space for each slice as you read the .raw data and use push_back() to push it onto the back of the vector. You can also set the vector to be the right number of elements and used the usual array index [] operator to set each element as required.

You *must* use dynamic memory allocation (using the "new" operator) to allocate space for each slice, so that you can write a statement like "slices[sliceIndex][row][col]" to access

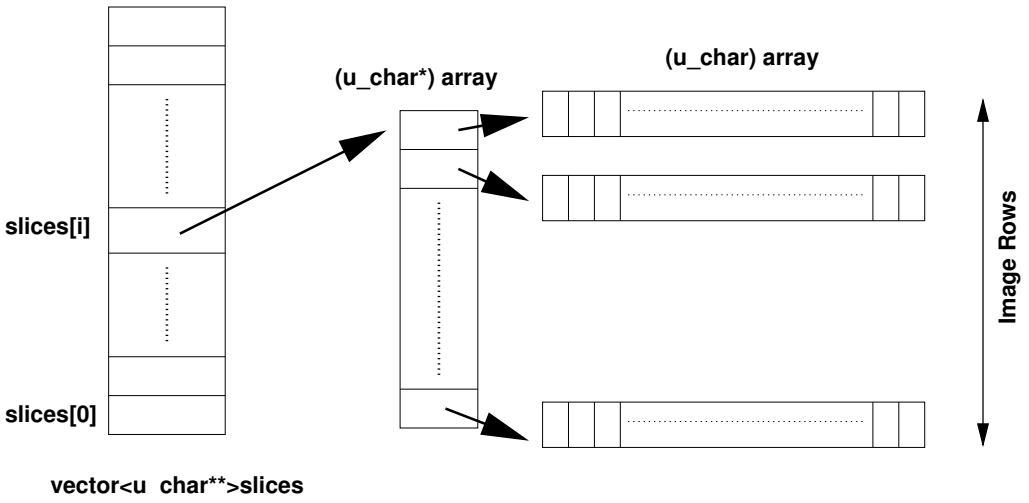

Figure 1: Memory allocation layout for the volimage data structure

the appropriate 3D intensity value in your later code. You must also ensure you can correctly delete all the memory you allocated (deletion occurs in your destructor). See figure 1 for a visualization on how this allocation should be implemented.

The following class structure is mandated (you must also user header and implementation files as expected):

```
class VolImage
{
private: // private members

   int width, height; // width and height of image stack
   std::vector<unsigned char**> slices; // data for each slice, in order

public: // public members

  VolImage(); // default constructor - define in .cpp
  ~VolImage(); // destructor - define in .cpp file

   // populate the object with images in stack and
   //set member variables define in .cpp
   bool readImages(std::string baseName);

   // compute difference map and write out;  define in .cpp
   void diffmap(int sliceI, int sliceJ, std::string output_prefix);

   // extract slice sliceId and write to output - define in .cpp
   void extract(int sliceId, std::string output_prefix);

   // number of bytes uses to store image data bytes
   //and pointers (ignore vector<> container, dims etc)
   int volImageSize(void); // define in .cpp
};
```

You should not create any non-default constructors – that will be addressed later in the

course, the constructor must simply zero width and height - the vector will be created with 0 size initially.

The destructor is where you place code to delete all the memory you have dynamically allocated (with new). If you do not do this correctly, your application will "leak" memory. Which is bad.

For your convenience we've attached a command line RAW file viewer which should be invoked as follows:

```
python viewer.py <raw filename> <width> <height>
```

You can also use tools such as "Gimp" to view the images - this assume a row-major 2D array of bytes as you can tell it the width and height for an "indexed image".

1. You can use stringstreams for many parsing tasks: if you need to extract or convert data to/from strings. Have a look at the examples in the notes and slides.
2. command line parsing can be done using third party parsers - **do not do this, write your own code**. The order of arguments is clearly specified, you can use a for loop with argc to visit each parameter and decide if it's the base name or a switch (like -x) or an argument for a switch. Remember that argv[i] is not a string object, it's a C-string. You can turn it into a string as follows: string s = string(argv[i]); you can then write code like: if (s == "-x") do something ...
3. You can use ifstream and ofstream to read in and write out binary data in C++. Using C-style fread/fwrite is not allowed. You can read more on this at `http://www.cplusplus.com/doc/tutorial/files/`

Handin Date: 2 March 2020 at 10:00AM

Please Note:

1. A working makefile must be submitted. If the tutor cannot compile your program in the lab by typing make, you will receive **50%** of your final mark.
2. You must use version control from the get-go. This means that there must be a .git folder alongside the code in your project folder. A **10%** penalty apply should you fail to include a local repository.
3. You must provide a README file explaining what each file submitted does and how it fits into the program as a whole. The README file should **not** explain any theory that you have used. These will be used by the tutors if they encounter any problems.
4. Do **not** hand in any binary files. Do **not** add binaries (.o files and your executable) to your local repository.
5. Please ensure that your tarball works and is not corrupt (you can check this by trying to extract the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - **no exceptions!**

6. A 10% penalty per day will be incurred for all late submissions. No hand-ins will be accepted if later than 5 days.

7. **DO NOT COPY. All code submitted must be your own.** ***Copying is punishable by 0 and can cause a blotch on your academic record.*** **Scripts will be used to check that code submitted is unique.**