Introduction

```
1   P3
2   # CREATOR: GIMP PNM Filter Version 1.1
3   480 353
4   255
5   8
6   11
7   4
8   7
9   10
10  3
11  7
12  10
13  3
```

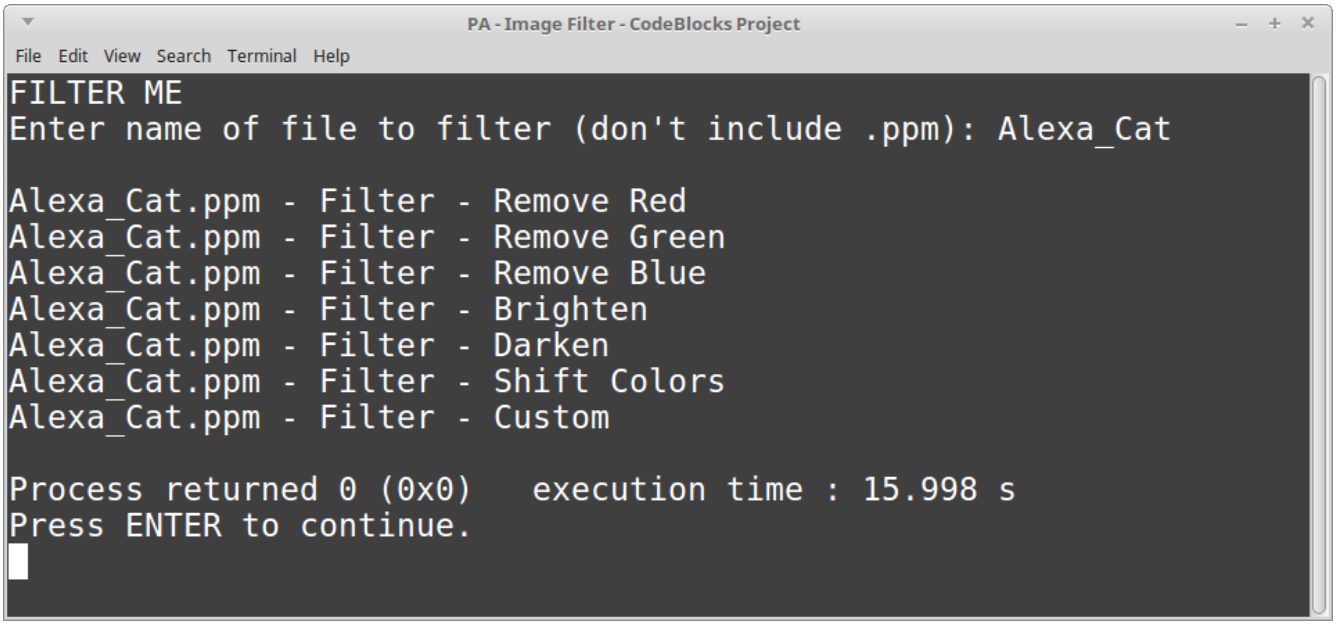
In this assignment, we will be working with loading in photos as text files, and modifying their pixels, before then exporting them again.

We will need to work with **.ppm** image file types, because as part of this file type, all the pixels are written out in plaintext, easily readable by a human, and by our programs.

Every **.ppm** file in the same format, so if we know *how* to read the numbers in the file, we can tell our program how to as well. After loading the pixels in, we will make modifications and write them back out in order to create "filters".

You will need to work with a class to create a structure that stores these pixels for an image to make it easier to work with these images in our program.

When you are finished, you will run the program and enter a filename:



It will go through the file and apply each filter to them and save them in the **output** folder.


Brighten


Darken


Shift Colors


Remove Blue


Remove Green


Remove Red

You will also implement a custom filter – it can be whatever you want to implement.

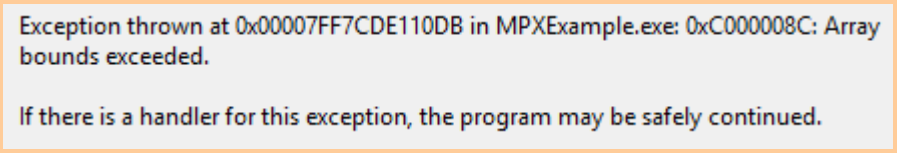
Programming Tips for Newbies

- **Build after every one or two lines of code that you write**
  – so that you can figure out if what you just wrote caused a **build error**! It is a lot easier to fix build errors when writing code step-by-step, instead after implementing everything!

- **Use the IDE's debugging tools**
  such as **breakpoints**! If your program isn't behaving as you were expecting, set a breakpoint in the function that isn't behaving right and step thru the program to watch its flow and its variable values!

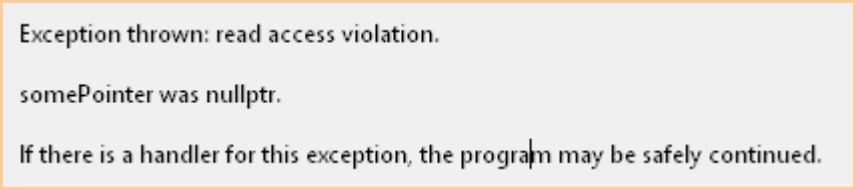
Common Errors

`"Exception thrown at (address): Array bounds exceeded"`



This error occurs when you go **outside the bounds of an array**. Click on the **Break** button to see the line of code that caused this error.

`"Exception thrown: read access violation. (pointer) was nullptr"`



This error happens when you try to **de-reference a pointer that is pointing to null.** Click on **Break** to see the line of code that caused this error.

Do not allow any memory leaks in your program! Memory leaks occur when you allocate memory with the **new** keyword but don't deallocate that memory with **delete** before the program ends!

Table of Contents

Table of Contents

Using GIMP

GIMP is a free image editing tool that you can download online at https://www.gimp.org/. You might have some other image editor on your machine that can open .ppm files, but if not, you can use GIMP.

Opening ppm files

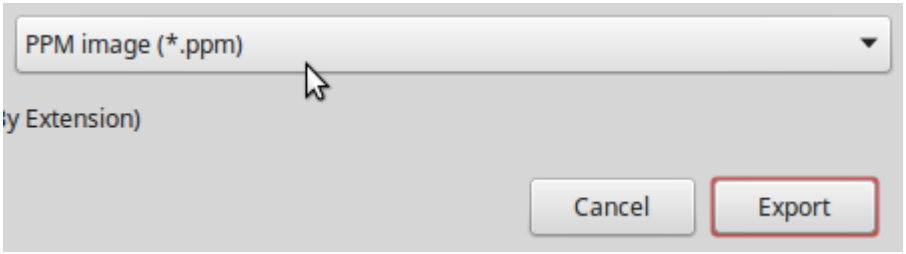
To open a file in GIMP, you can either click-and-drag the .ppm file into the open GIMP window, or you can go to **File > Open…** and locate the file to open.

How to prep an image to be filtered

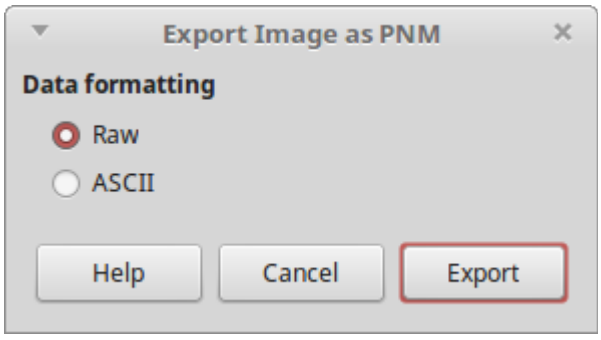
If you want to use an image besides the ones provided in the project folder, you can prep your own images.

First, open the image in GIMP. Then go to **File > Export As…**

In the file save screen, click on the file type dropdown and select **PPM image (*.ppm)** Make sure the filename also ends with .ppm.



Click on **Export**. It will ask whether you want to save it as “Raw” or “ASCII”. Select **ASCII** then click **Export** again. Then your PPM file will be saved.



Your file should be ready to import into your program now.

Files

This project has the following files:

program.cpp
Contains the main() function and some other functions that will run each filter.

You will not modify this file.

Image.hpp
Header files contain the class declaration and function declarations.

This has already been declared for you, but make sure to look at the file to be familiar with what the class contains.

Image.cpp
The source file is where class functions get defined. This is where you will implement your code.

The project folder also contains three stock images converted to .ppm files ready to work with. You can open these files with a text editor like notepad or notepad++ to see how they are structured. You can also convert your own .ppm images.

Alexa_Bubbles.ppm

Alexa_Cat.ppm

Pitsch_Dog.ppm

Files to be read in need to be stored **in the project location**. With Code::Blocks this will be where the CodeBlocks Project (.cbp) is. For Visual Studio, this is where your VS Project (.vcxproj) file is, NOT the solution file!

Image.hpp

In Image.hpp, a class and a struct is declared:

The **Pixel** struct is a simple structure where our RGB (red/green/blue) values will be stored for each pixel in the image.

```
struct Pixel
{
    int r, g, b;
};
```

The **Image** class contains the functions needed to read and write image files, as well as the filters. The Image also contains a **dynamic array of Pixels**, and Image is also responsible for allocating and deallocating this memory.

```
class Image
{
    public:
    Image();
    ~Image();

    void ReadFile( const string& filename );
    void WriteFile( const string& filename );
    void PrintStats();

    void AllocateArray( int size );
    void DeallocateArray();

    void Filter_RemoveRed();
    void Filter_RemoveGreen();
    void Filter_RemoveBlue();
    void Filter_Brighten();
    void Filter_Darken();
    void Filter_ShiftColors();
    void Filter_Custom();

    private:
    string inputFilename;
    string magicNumber;
    string headerNote;
    int width;
    int height;
    int colorDepth;

    Pixel* pixelArray;
    int pixelCount;
};
```

Image.cpp

Phase 1: Allocating and deallocating memory

First we will implement what we need to make sure we're allocating memory and deallocating that memory. Because we don't know how big each image is until we've started reading it, we use a **dynamic array** to store all the pixels of the image.

`Image::Image()`

This function is the class' **constructor**. The constructor is a special function that is executed automatically when a new Image variable is declared. This is a good place to initialize class data.

- Within this function, all you need to do is set the **pixelArray** pointer to nullptr.

`Image::~Image()`

This function is the class' **destructor**. The destructor is another special function, and it is executed automatically when an Image variable is destroyed, such as when it goes out of scope. This is a good place to clean up the class.

- Within this function, call the **DeallocateArray()** function.

`void Image::AllocateArray( int size )`

Input parameters: int size

For this function, the desired *size* is passed in – this is how big we wish for the array to be.

- Using the member pointer variable **pixelArray**, allocate a new Pixel array of size **size**.
- Also set the member variable **pixelCount** to the value of **size**.

`void Image::DeallocateArray()`

For this function, we need to first check if the pixelArray pointer is pointing to an address, or to *nullptr.* If it is pointing to nullptr, we don't need to do anything – it should already be free. Otherwise, we need to destroy the memory it is pointing to.

- If the member pointer variable **pixelArray** is not assigned to *nullptr* then
  - Delete the **pixelArray** dynamic array.
  - Set the **pixelArray** pointer to *nullptr.*
  - Set **pixelCount** to 0.

Phase 2: Reading and writing the image files

Next we will write the Read and Write functions so that our program can handle the image files. Each file is a standardized format, which includes some data in the header, and then a string of RED, GREEN, BLUE values.

```
1   P3                                      "Magic Number"
2   # CREATOR: GIMP PNM Filter Version 1.1  Note
3   480 320                                 Width   Height
4   255                                     Color depth
5   31                                      Pixel 0 - Red
6   34                                      Pixel 0 - Green
7   41                                      Pixel 0 - Blue
8   31                                      Pixel 1 - Red
9   34                                      Pixel 1 - Green
10  41                                      Pixel 1 - Blue
```

First you will have to load in the header data and allocate space for the amount of pixels, then you can loop through the rest of the file reading in each pixel.

Likewise, when writing out the file, you begin with the header data, and then loop through all your pixels and output RED, GREEN, and BLUE values.

`void Image::ReadFile( const string& filename )`

Input parameters: const string& filename This is a const string reference

- Create an input file stream named **input**. Open the **filename** that was passed in.
- Using the >> operator, read in the first item from the **input** ifstream into the **magicNumber** member variable. This is the “P3” value.
- Use **input.ignore();** immediately after. This is a workaround, if we switch between >> and getline, it will skip a line.
- Use the **getline** function on **input** to store the note “# CREATOR: GIMP PNM Filter Version 1.1” into the **headerNote** variable.
- Use the >> operator with **input** again to read the last three header items: Width, Height, and Color Depth. Store these in the member variables **width**, **height**, and **colorDepth**.
- After this data is read in, we can allocate space for our array. Call the **AllocateArray** function, passing in ***width * height*** as the size – this is the total amount of pixels in the image.
- Create a for loop. Use an integer counter variable ***pixel*** and initialize it to 0. Loop while **pixel** is less than **pixelCount**. Increment **pixel** by 1 each time. Within the loop:
  - Working with the **pixelArray[pixel]**, you will load in its **r**, **g**, and **b** values from the **input** file. You will need to use the member-of operator “.” after **pixelArray[pixel]** to access **r**, **g**, or **b**, like: **pixelArray[pixel].r**
- After the for loop is over, make sure to **close** the **input** file.

`void Image::WriteFile( const string& filename )`

Input parameters: const string& filename This is a const string reference

- Create an output file stream named **output**. Open the **filename** that was passed in.
- Output the **magic number** on its own line.
- Output the **headerNote** on its own line.
- Output the **width**, then a space, then the **height** on its own line.
- Output the **colorDepth** on its own line.
- Create a for loop to iterate over every pixel. With an integer counter variable named **pixel**, initialize it to 0, loop while **pixel < pixelCount**, and increment by 1 each time. In the loop:
  - Output the **r**, **g**, and **b** values (separated by spaces) of the **pixelArray[pixel]** item.
- After the for loop is over, **close** the **output** file.

`Test the program!`

Make sure to run and test the program before continuing! At the moment, it should just read in the file and output a copy of the same file to the **output** folder.

Phase 3: Implementing the filters

Now we will implement filters. Each filter will iterate through every pixel of the image and change the pixel R, G, B values in some way. To iterate through every pixel, you will use a for loop like this:

```
for ( int i = 0; i < pixelCount; i++ )
{
    // do something with pixelArray[i].r,
   // pixelArray[i].g, and/or pixelArray[i].b.
}
```

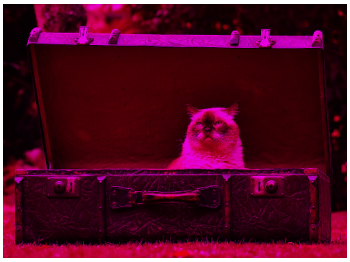
Test after implementing each filter!

`void Image::Filter_RemoveRed()`

As you iterate through all the pixels, set **only the red values** to 0.

`void Image::Filter_RemoveGreen()`

As you iterate through all the pixels, set **only the green values** to 0.



`void Image::Filter_RemoveBlue()`

As you iterate through all the pixels, set **only the blue values** to 0.

`void Image::Filter_Brighten()`

As you iterate through each pixel, ***double*** each of the red, green, and blue values.

Also make sure they don't go above 255. If red, green, or blue goes above 255, set that individual color value back to 255.

`void Image::Filter_Darken()`

As you iterate through each pixel, ***halve*** each of the red, green, and blue values.

No error checking needed here; the division won't result in a negative value.

`void Image::Filter_ShiftColors()`

As you iterate through each pixel, shift the RGB values:

- Set Red to the Green value
- Set Green to the Blue value
- Set Blue to the Green value

You may need to create some temporary variables to store the original RGB values.

void Image::Filter_Custom()

Create your own custom filter as you want to do. Some ideas…:

- Reverse the R and B values
- Add scanlines to the image
- Only filter *half* the image
- Borrow red, green, and blue values from neighboring pixels

Turning in your project

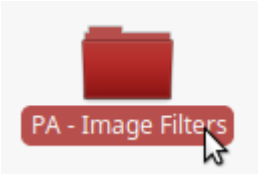
Option 1: Just the source files

All I really need are your source files. You *can* just upload these and it's enough:

- Image.hpp
- Image.cpp
- program.cpp

Option 2: The entire project

Navigate outside the project folder and zip up the entire folder.



Note: If you converted your own personal images for use with the filter, you might want to remove those first.

Grading breakdown

You will gain points (0-5) for each feature. There are also penalties, which will cut down on your grade, based on how bad the error is (0-4).

Breakdown			
Score			
Item	**Score (0-5)**	**Task weight**	**Weighted score**
`Image constructor`	5	5.00%	**5.00%**
`Image destructor`	5	5.00%	**5.00%**
`ReadFile`	5	15.00%	**15.00%**
`WriteFile`	5	15.00%	**15.00%**
`AllocateArray`	5	5.00%	**5.00%**
`DeallocateArray`	5	5.00%	**5.00%**
`Filter_RemoveRed`	5	5.00%	**5.00%**
`Filter_RemoveBlue`	5	5.00%	**5.00%**
`Filter_RemoveGreen`	5	5.00%	**5.00%**
Filter_Brighten	5	10.00%	**10.00%**
Filter_Darken	5	10.00%	**10.00%**
Filter_ShiftColors	5	10.00%	**10.00%**
Filter_Custom	5	5.00%	**5.00%**
Score totals		100.00%	**100.00%**

Penalties			
Item	**Score (0-4)**	**Max penalty**	**Weighted penalty**
Syntax errors (doesn't build)	0	-50.00%	0.00%
Logic errors	0	-10.00%	0.00%
Run-time errors	0	-10.00%	0.00%
Memory errors (leaks, bad memory access)	0	-10.00%	0.00%
Ugly code (bad indentation, no whitespacing)	0	-5.00%	0.00%
Ugly UI (no whitespacing, no prompts, hard to use)	0	-5.00%	0.00%
Not citing code from other sources	0	-100.00%	0.00%
Penalty totals			**0.00%**