CS103 Unit 4d – Multidimensional Arrays and Images

CSCI 103L Teaching Team

Motivation

- We've focused on 1D data structures: vectors (arrays) and linked lists.
- Let's look at how C++ represents multidimensional arrays.

MULTIDIMENSIONAL ARRAYS

Multidimensional Arrays

- Thus far, arrays can be thought of 1-dimensional (linear) sequences
  - only indexed with 1 value (coordinate)
- We often want to view our data as 2D, 3D or higher dimensional data
  - Matrix data or images (2D for B&W, 3D for color)
  - Index w/ 2 coordinates (row,col)

```
char dat[6] =
{1,2,3,4,5,6};
```

dat

Address	Mem.
7419	ab
7420	01
7421	02
7422	03
7423	04
7424	05
7425	06
7426	fe

Most images use 8-bit (1-byte) numbers with the range: 0-255 to represent shades of grade or color intensity.

In C++, the type that is 1-byte and fits the range 0-255 is the `unsigned char` type. Thus, you will see a lot of `unsigned char` 2D and 3D arrays in this unit.

So, remember, chars and unsigned chars don't necessarily mean ASCII, they simple mean a small integer

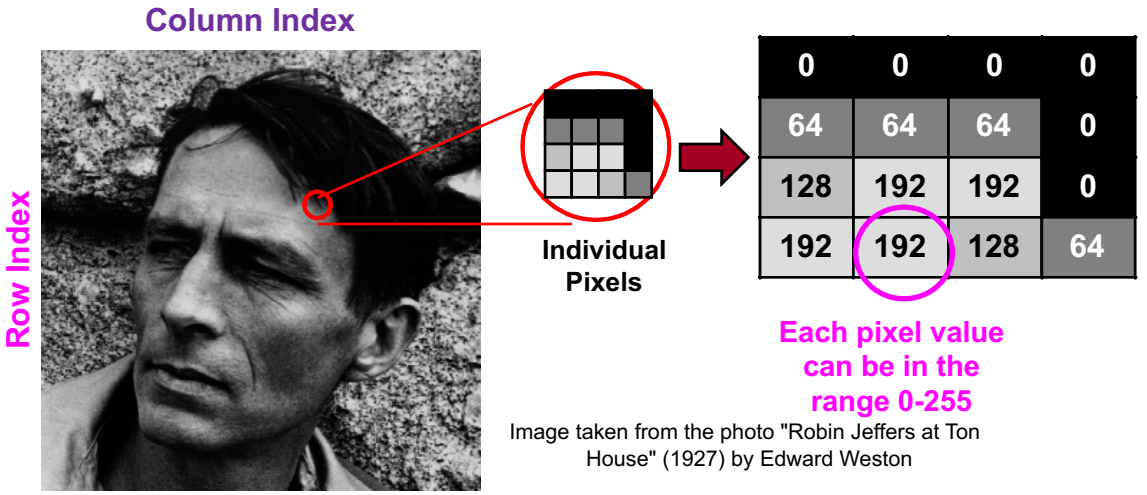


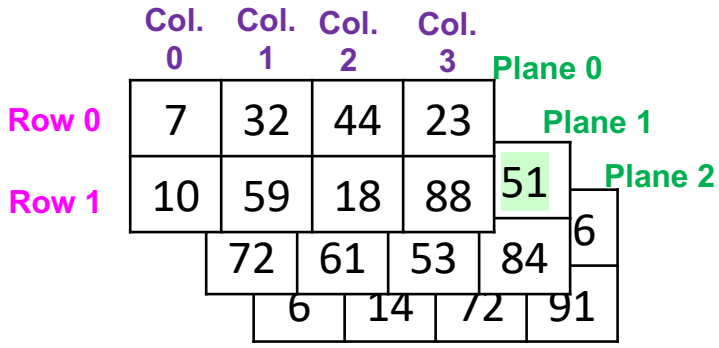
Image taken from the photo "Robin Jeffers at Ton House" (1927) by Edward Weston

Multidimension Array Declaration

- 2D: Declare by providing size along both dimensions (normally rows first then columns) and access with 2 indices
  - Declaration: `int matrix[2][3];`
  - Access elements with appropriate indices
    - `my_matrix[0][1]` evaluates to 3, `my_matrix [1][2]` evals to 2

	Col. 0	Col. 1	Col. 2
Row 0	5	3	1
Row 1	6	4	2

- 3D: Declare and access data with 3 indices
  - Declaration: `unsigned char image[2][4][3];`
  - Access elements with appropriate indices
    - `image[0][3][1]` evals to 51



Interpreting Dimensions

- Given the declaration: **`int matrix[2][3];`** is this
  - 2 rows by 3 columns? or
  - 2 columns by 3 rows?
- It could be either!! In C/C++ it is YOUR CHOICE for how to interpret the dimension ordering.
  - If you want to think of this as 2 rows by 3 columns, fine.
  - If you prefer 2 columns by 3 rows, fine as well!
- Wait, huh? How? Why?

	Col. 0	Col. 1	Col. 2
Row 0	5	3	1
Row 1	6	4	2

	Col. 0	Col. 1
Row 0	5	6
Row 1	3	4
Row 2	1	2

Why The Dimension Order Doesn't Matter

- You want to buy tickets to a concert for your friends.
  - You are part of **3 clubs**, each having **2 teams** and **4 members** per team (for a total of 24 tickets).
  - The friends from each club want to sit together
- I want to buy tickets to the same concert for my family.
  - I have my side and my wife's (**2 sides**), each with **3 siblings** and **4 members of each siblings' family** (for a total of 24 tickets).
    [I know…my family and my wife's are the same => 20 tickets…just go with me]
  - Each side wants to sit with its own members
- We each buy 24 tickets. Does the ticket sales office care how we divide those 24 tickets?
- **No!! How we organize, add hierarchy, and distribute our 24 tickets doesn't matter** to the ticket office, but it does matter with regard to who is next to each other.
- In the same way, when we allocate an array, the **computer just sets aside a total amount of memory** for us, but **how we list the dimensions determines what elements "sit" next to each other**

Interpreting Multidimensional Data

- Given the declaration: `unsigned char image[2][4][3];`
  - "Higher" Dimension
  - "Lower" Dimension
- The computer allocates 24 bytes (2*4*3*sizeof(char))
- We can visualize this as:
  - 2 Rows by 4 columns by 3 planes, or
  - 2 Planes by 4 rows by 3 columns, or
  - …

Address	Mem.			
7416	ab	ab	ab	ab
image 7420	35	03	01	06
7424	14	72	10	81
7428	63	40	75	18
7432	65	71	12	84
7436	47	49	23	52
7440	25	95	37	07
7444	cd	cd	cd	cd

	Col 0	Col 1	Col 2	Col 3	
Row 0	35	06	10	40	Plane 0
Row 1	65	84	23	95	
	03	14	81	75	Plane 1
	71	47	52	37	
	01	72	63	18	Plane 2
	12	49	25	07	

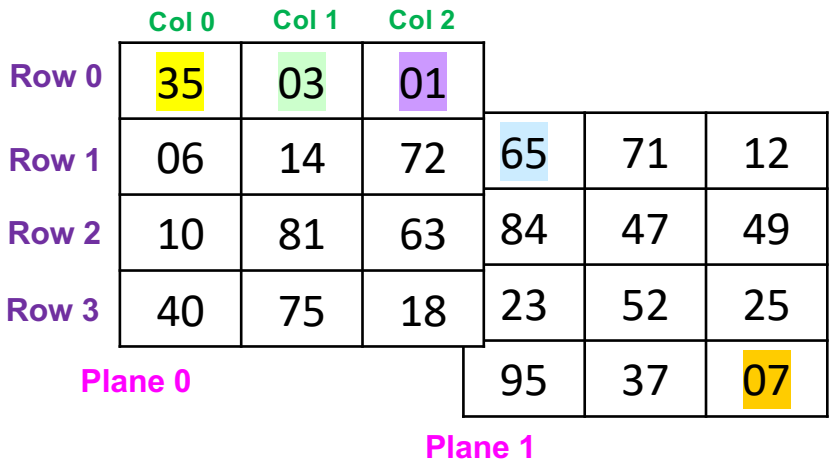
or

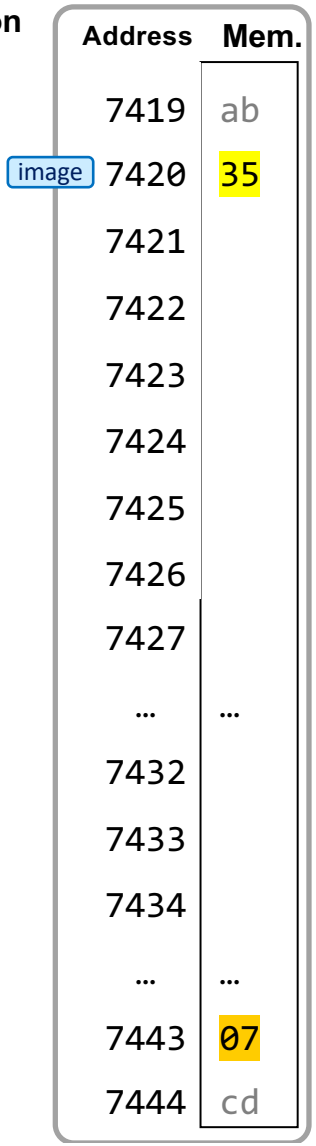
	Col 0	Col 1	Col 2	
Row 0	35	03	01	
Row 1	06	14	72	
Row 2	10	81	63	
Row 3	40	75	18	Plane 0
	65	71	12	
	84	47	49	
	23	52	25	
	95	37	07	Plane 1

Linear-izing Multidimensional Data

"Higher" Dimension "Lower" Dimension

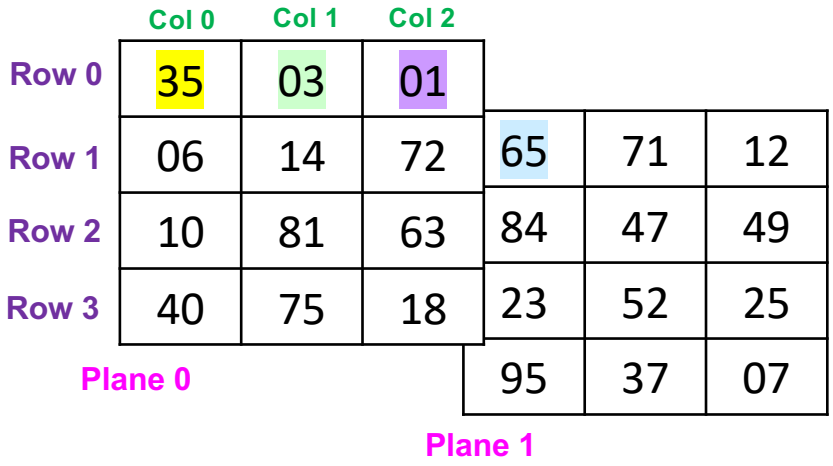
- Given the declaration: `unsigned char image[2][4][3];`
- If you had to slot this multidimensional data into a linear (1 dimensional) array, how would you do it?
  - You could iterate through the **lowest** dimension first and work your way up to the **highest** dimension
  - OR you could start at the **highest** dimension first and work your way down to the **lowest** dimension

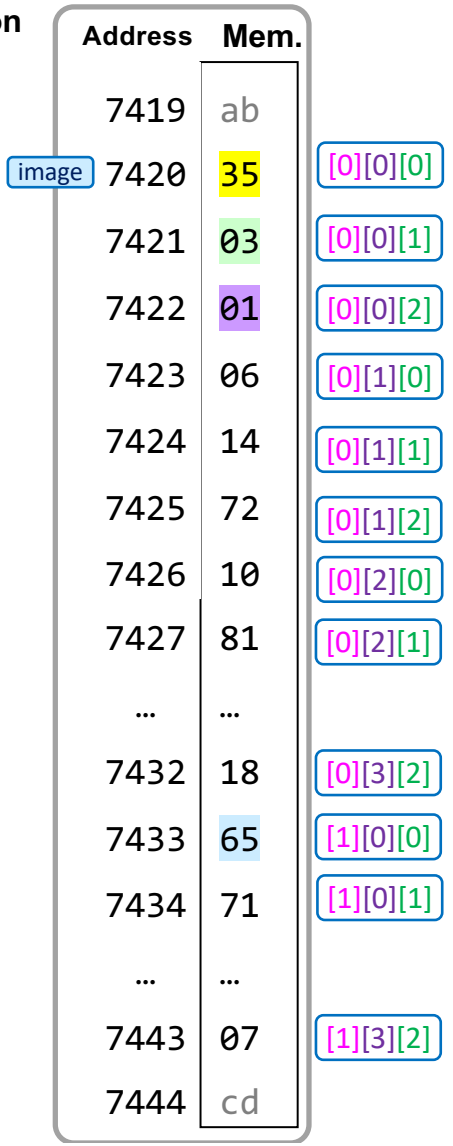




Linear-izing Multidimensional Data

- Given the declaration: `unsigned char image[2][4][3];`
  - "Higher" Dimension
  - "Lower" Dimension
- If you had to slot this multidimensional data into a linear (1 dimensional) array, how would you do it?
- C/C++ uses the policy that the **elements of the lowest dimension are "next" to each other** while the highest dimension is "farthest" apart in memory





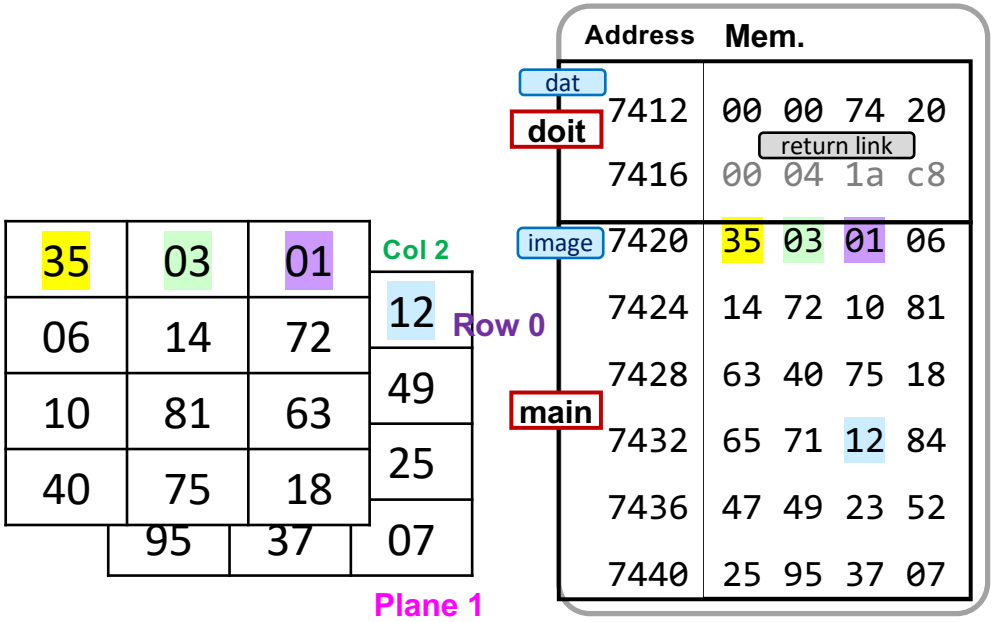
Passing Multi-Dimensional Arrays

- **Formal Parameter**: Must give dimensions of all but first dimension (and you may give that dimension if you please)
- **Actual Parameter**: Still just the array name (i.e. still only passes the starting address)
- But why **don't** we have to list the upper dimension size in the formal argument type? (And why do the lower dimension sizes NEED to be provided?)
  - Let's explore briefly…

```
void doit(char dat[][4][3]) // or char dat[2][4][3]
{
  dat[1][0][2] += 10;
}

int main()
{
  char image[2][4][3];

  doit(image);
  ...
  return 0;
}
```



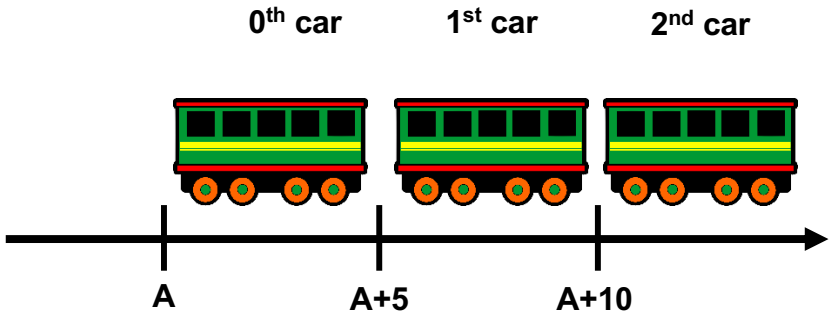
Recall

- Consider a train with many copies of the same car
  - The "0th" car starts at point A on the number line
  - Each car is 5 meters long
- Write an expression of where the i-th car is located (at what meter does it start?)
- Suppose a set of integers start at memory address A, write an expression for where the i-th integer starts?
- Suppose a set of doubles start at memory address A, write an expression for where the i-th double starts?
- **General Formula**: `start_addr + i*data_size`
- **Does such a formula exist for multidimensional arrays?**

A + 5*i

A + 4*i

A + 8*i



Linearization (Finding the address) in 2D Arrays

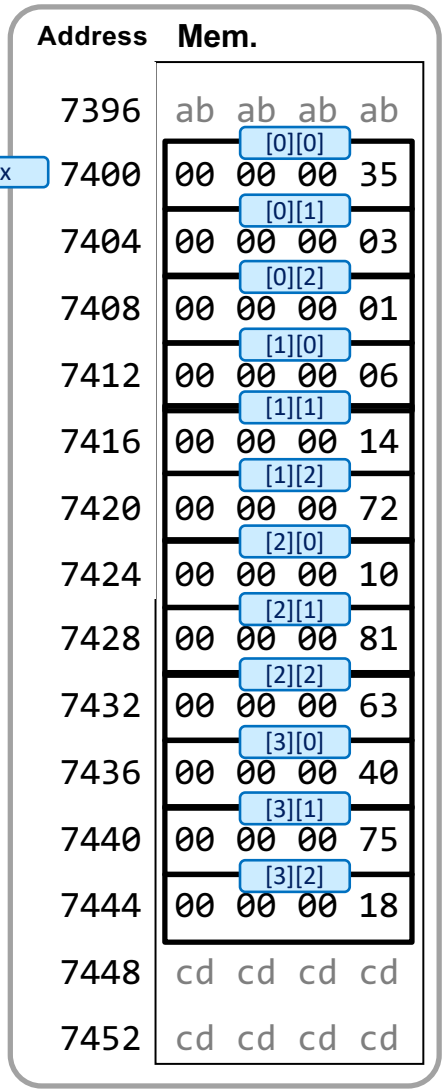
- Given an array with **NR** rows and **NC** columns
  - Using the memory displayed at the right, what address does each row start at (i.e. …, `x[1][0]`, `x[2][0]`, …)

Declaration: `int x[4][3]; // NR=4, NC = 3;`

Mem. Address:

	Col 0	Col 1	Col 2
7400 Row 0	35	03	01
7412 Row 1	06	14	72
7424 Row 2	10	81	63
7436 Row 3	40	75	18

Moving down a row requires skipping over **NC** elements. So moving to row **i** requires skipping **i*NC** elements.



- Derive a formula for the address of item at row `i`, column `j`

Access: `x[i][j]:`

```
7400 + (i*NC + j)*sizeof(int)
```

`x[1][2] = 7400 + (1*3 + 2)*4 = 7400 + 5*4 = 7420`

`x[3][1] = 7400 + (3*3 + 1)*4 = 7400 + 10*4 = 7440`

Linearization (Finding the address) in 3D Arrays

- Given an array with **NP** planes, **NR** rows, and **NC** columns
  - Using the memory displayed at the right, what address does each plane start at (i.e. `x[0][0][0]`, `x[1][0][0]`, …)

Declaration: `int x[2][4][3]; // NP=2, NR=4, NC = 3;`

Mem. Address:

Plane 0

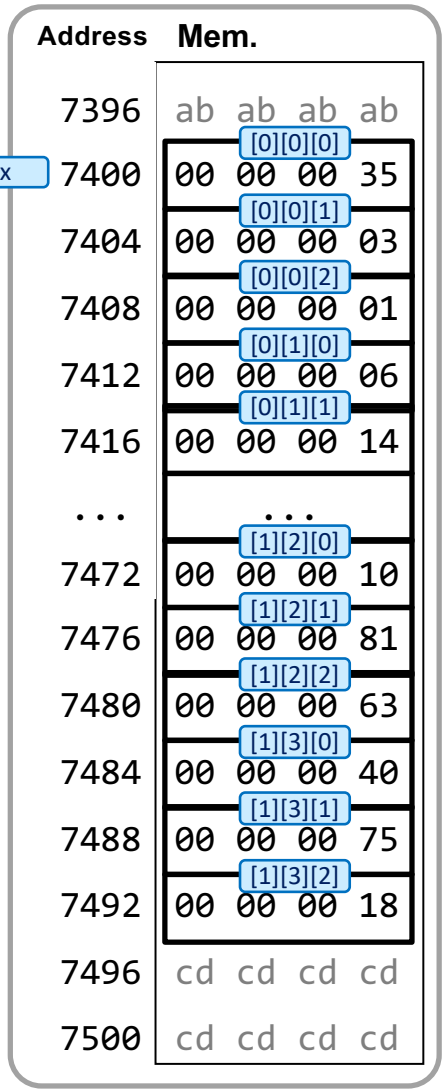
7400	35	03	01
7412	06	14	72
7424	10	81	63
7436	40	75	18

skip NR*NC elements.

skip NC elements

Plane 1

7448	65	71	12
7460	84	47	49
7472	23	52	25
7484	95	37	07



- Derive a formula for the address of item at row `i`, column `j`

Access: `x[p][i][j]:`

```
7400 + (p*NR*NC + i*NC + j)*sizeof(int)
```

`x[0][1][2] = 7400 + (0*4*3 + 1*3 + 2)*4 = 7420`

`x[1][3][0] = 7400 + (1*4*3 + 3*3 + 0)*4 = 7484`

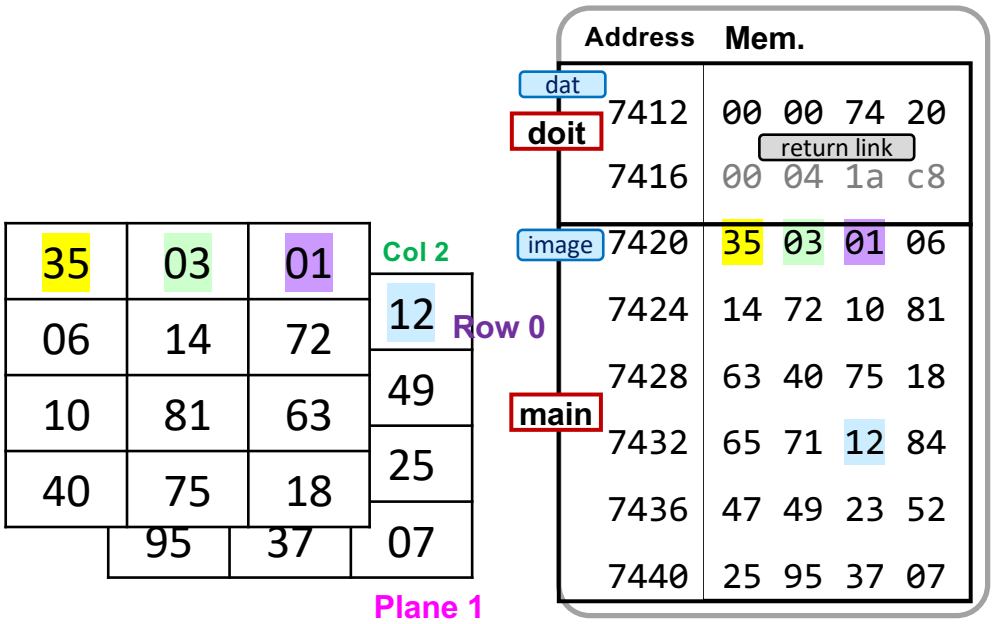
Passing Multi-Dimensional Arrays

- **Formal Parameter**: Must give dimensions of all but first dimension (and you may give that dimension if you please)
- **Actual Parameter**: Still just the array name (i.e. still only passes the starting address)

```
void doit(char dat[][4][3]) // or char dat[2][4][3]
{
  dat[1][0][2] += 10;
}

int main()
{
  char image[2][4][3];

  doit(image);
  ...
  return 0;
}
```



Review of Passing Multidimensional Arrays

- Why do we have to provide all but the first dimension?
  - So that the computer can determine where element: `dat[i][j][k]` is actually located in memory
- Recall our formulas for the location of 2 and 3D arrays below.
- Notice the highest dimension (NR for 2D and NP for 3D) does not appear in the fomulas?
  - Because C/C++ does no bounds checking the highest dimension is not needed

```
void doit(char dat[][4][3]) // or char dat[2][4][3]
{
  dat[1][0][2] += 10;
}

int main()
{
  char image[2][4][3];

  doit(image);
  ...
  return 0;
}
```

2D array element address formula (NR rows and NC cols)

```
start-address + (i*NC + j)*sizeof(type)
```

3D array element address formula (NP planes, NR rows, and NC cols)

```
start-address + (p*NR*NC + i*NC + j)*sizeof(type)
```

35	03	01	Col 2
06	14	72	12 Row 0
10	81	63	49
40	75	18	25
95	37		07

Plane 1

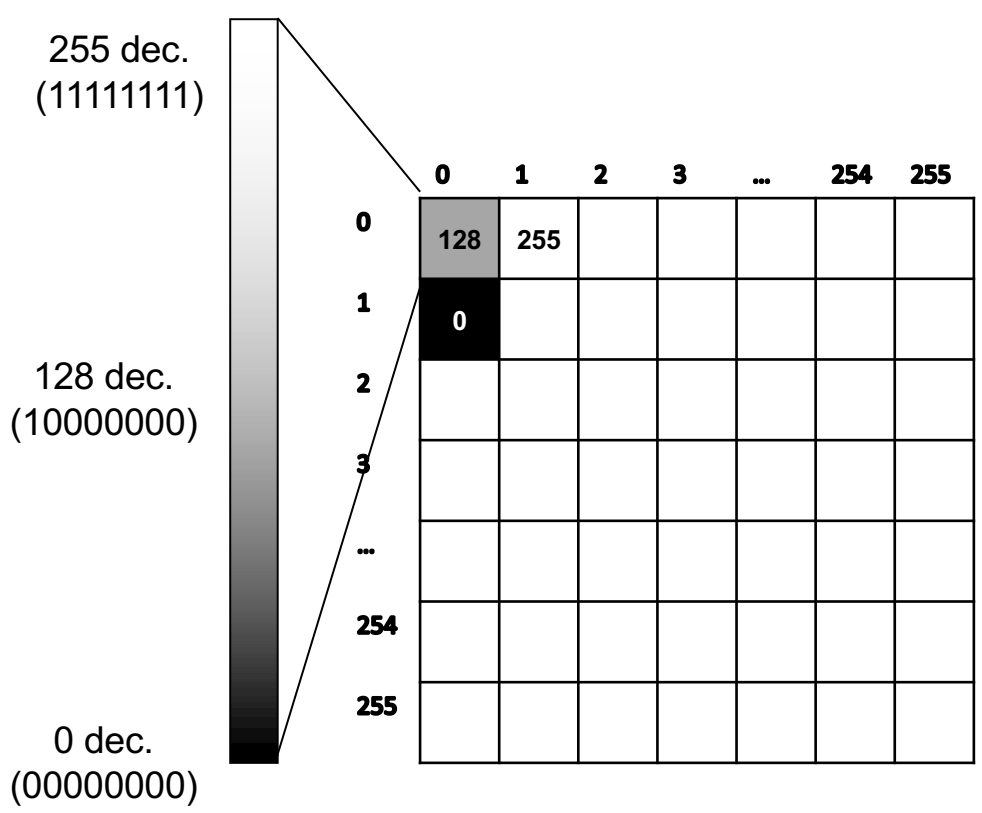
Address	Mem.	
dat		
doit	7412	00 00 74 20
		return link
	7416	00 04 1a c8
image	7420	35 03 01 06
	7424	14 72 10 81
	7428	63 40 75 18
main	7432	65 71 12 84
	7436	47 49 23 52
	7440	25 95 37 07

Using 2- and 3-D arrays to create and process images

IMAGE PROCESSING

Grayscale Images

- A grayscale image is simply a 2D array of pixels where each value corresponds to a shade of gray (see graphic to the right)
- Most grayscale images use 8-bit numbers for each pixel
  - 0 = black, 255 = white
  - This range corresponds to an **`unsigned char`**
  - Thus, our images are 2D arrays of type: **`unsigned char[256][256]`**
- Images and patterns can be "drawn" by setting the values at specific indices



Motivation

- We will spend some time looking at image processing because
  - It is EXCELLENT practice at finding and generalizing patterns which is a key skill for computer science and algorithm development
  - It helps with using arrays
  - It's fun and produces results that are intuitive to evaluate and check

Practice: Drawing

- See Codio instructions
  - Code to read (open) and write (save) .BMP files is provided in bmplib.h and bmplib.cpp
  - Look at bmplib.h for the prototype of the functions you can use in your main() program in gradient.cpp

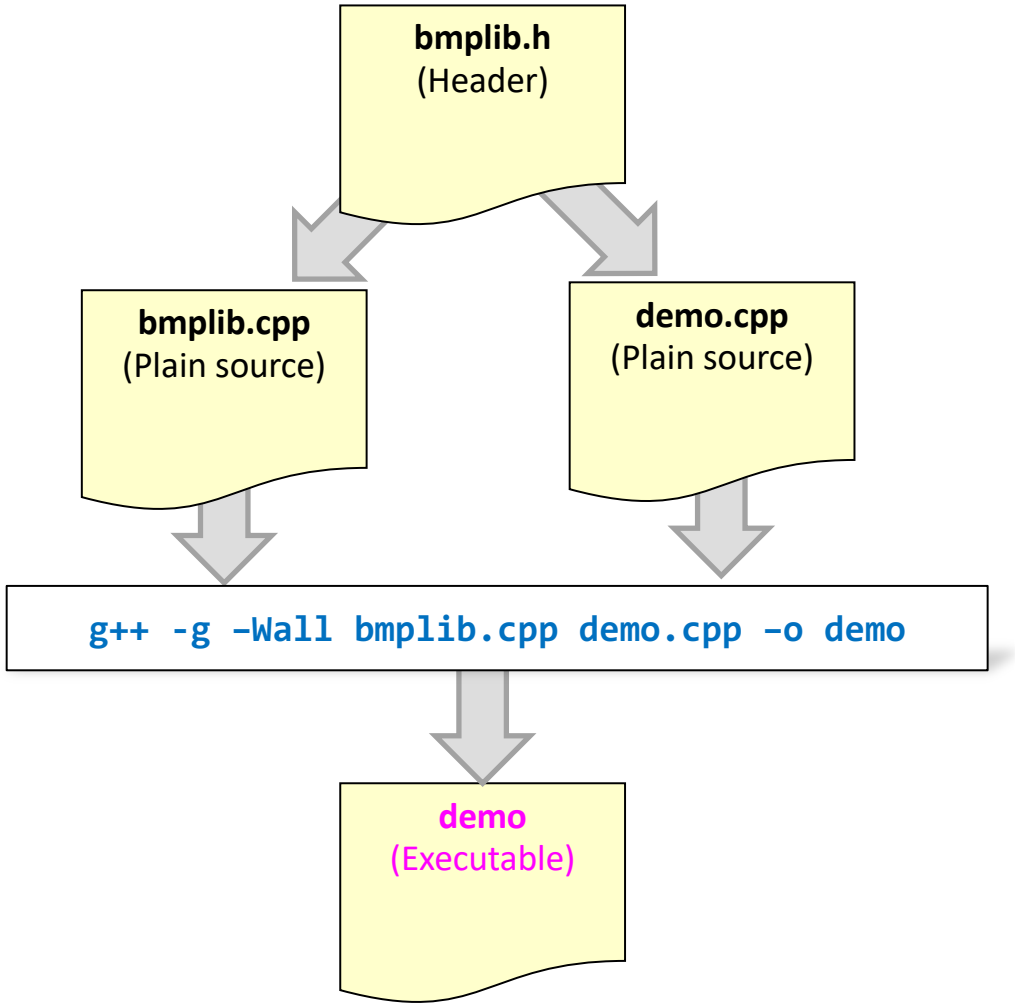
Multi-File Programs

- The BMP code is split into a separate .cpp (and .h) file so that we can:
  - Use it in many application
  - Update/replace it separately
- bmplib.h has prototypes for functions to read, write, and show .BMP files as well as constant declarations
- bmplib.cpp has the implementation of each function
- demo.cpp has the main application code
  - It #include's the .h file so as to have prototypes and constants available

Key Idea: The ***.h*** file tells you ***what*** library functions are available; The ***.cpp*** file tells you ***how*** it does it

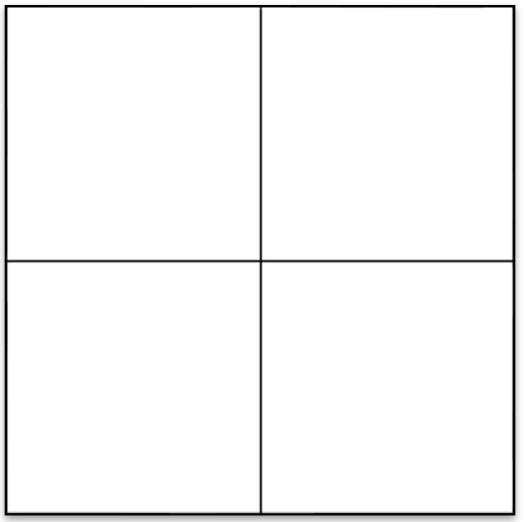
Reminder: Compilation

- You must list ALL source (`.cpp`) files in the g++ command
- Do **NOT** list header (`.h`) files in the g++ command, as the code from `.h` files is compiled as part of the `.cpp` files they are included in
- Use 'make' with the provide 'Makefile' script
  - Just type 'make' at the prompt



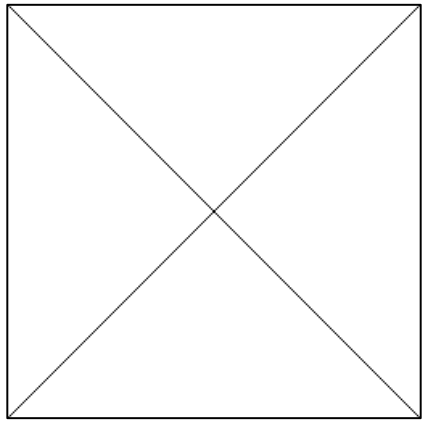
Demo Code

- The demo code draws a cross
- Cross
  - Use a loop to iterate down the center row and column setting the pixels to black

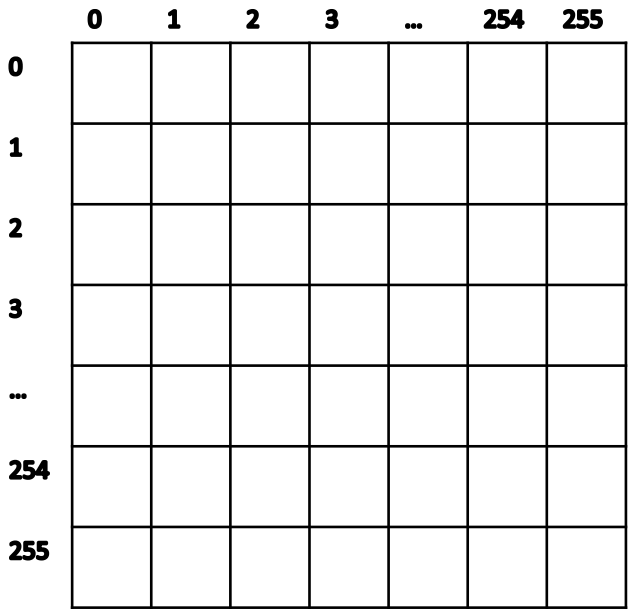


Practice: Drawing an X

- Draw an X on the image
  - Try to do it with only a single loop, not two in sequence
  - Identify patterns in indices of what you want to draw

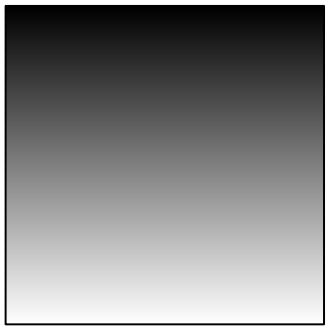


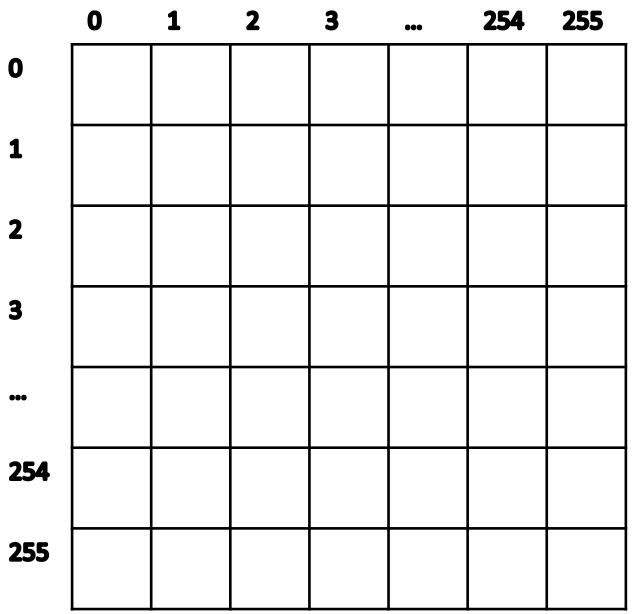
Tip: Write out the coordinates that need to be set and what value you want to set them to, then generalize to a pattern you can write with a loop in terms of the inductive variable (e.g. i).



Practice: Simple Gradient

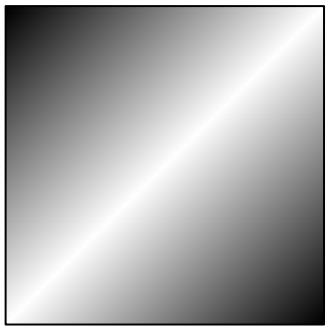
- Modify demo.cpp to draw a gradient down the rows (top row = black through last row = white with shades of gray in between





Practice: Not So Simple Gradient

- Modify demo.cpp to draw a diagonal gradient with black in the upper left through white down the diagonal and then back to black in the lower right



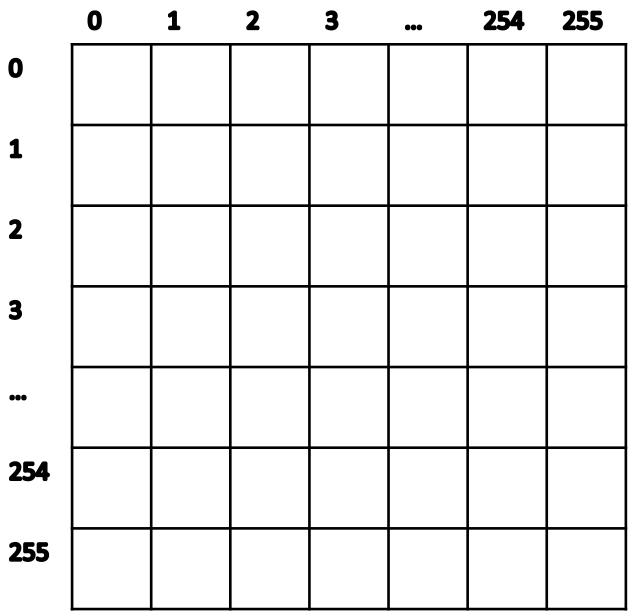
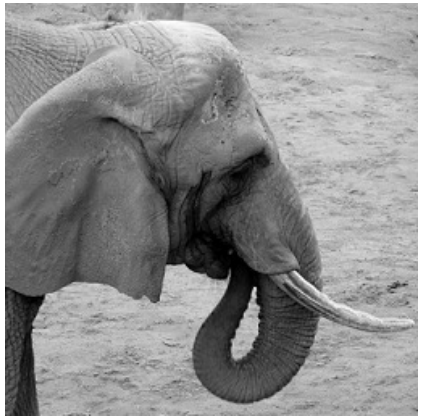


Image Processing

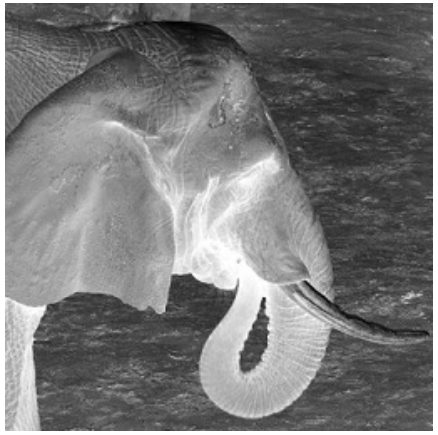
- Here is a first exercise…produce the "photo-negative"
  - Ask yourself: Are we transforming pixel locations, pixel values, or both?



Original

```
#include "bmplib.h"
int main() {
   unsigned char image[SIZE][SIZE];
   readGSBMP("elephant.bmp", image);
   for (int i=0; i<SIZE; i++) {
      for (int j=0; j<SIZE; j++) {

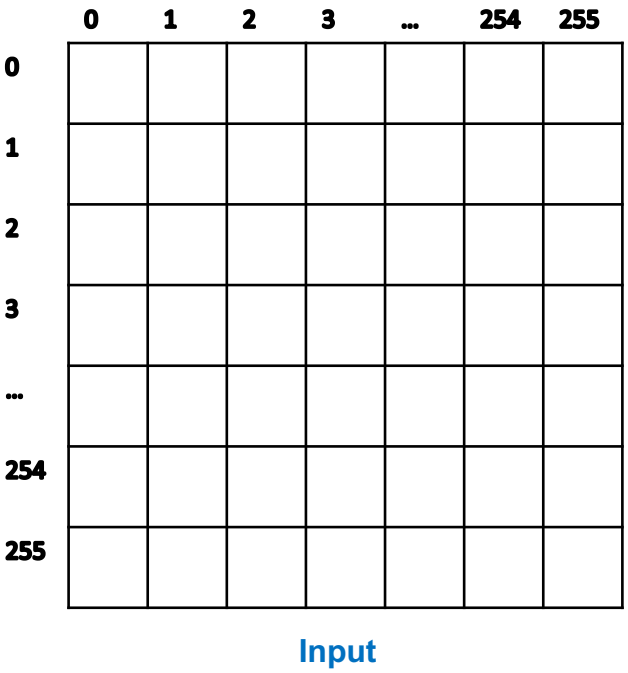
         image[i][j] = ____________________
         // invert color
      }
   }
   showGSBMP(image);
}
```

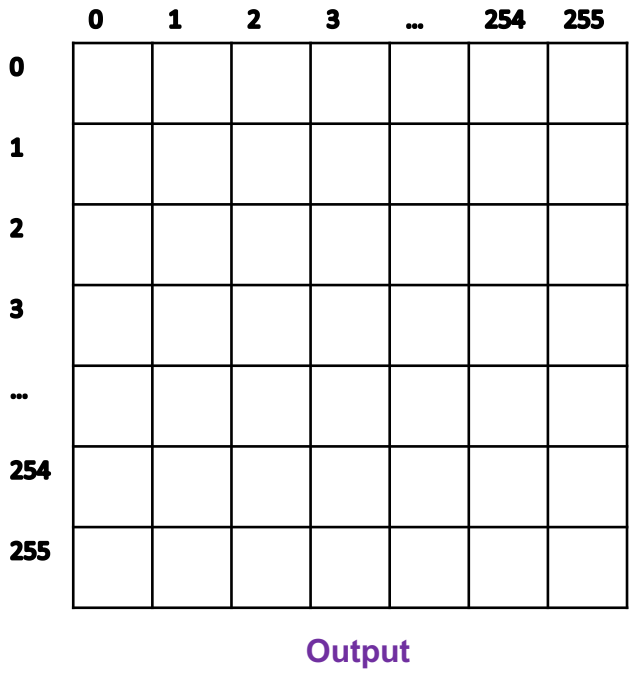


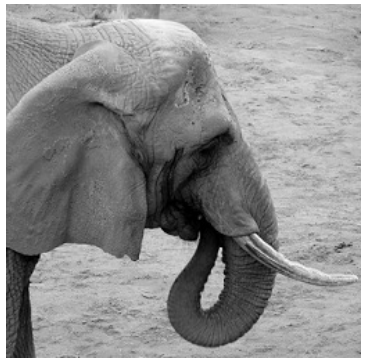
Inverted

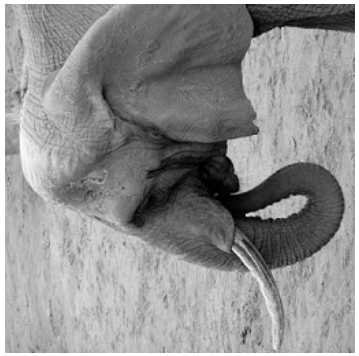
Practice: Image Processing

- Perform a diagonal flip
  - Consider the coordinates to visit from the input and output images
  - Ask yourself: Are we transforming pixel locations, pixel values, or both?



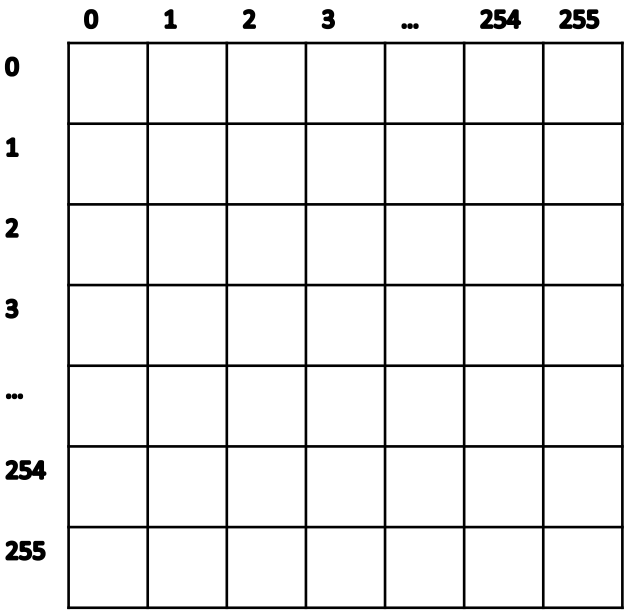




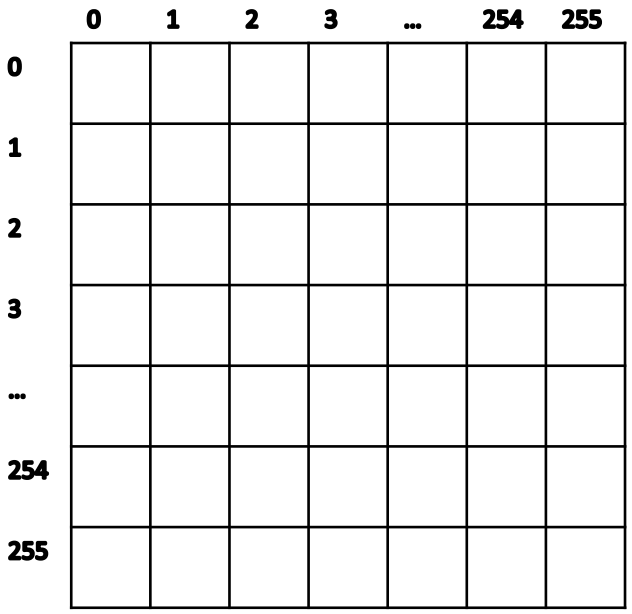


Practice: 2x2 Tile

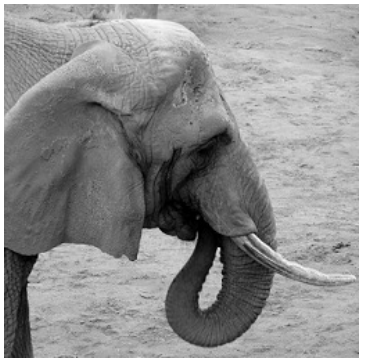
- Perform a 2x2 "tile" operation
  - Consider the coordinates to visit from the input and output images
  - Ask yourself: Are we transforming pixel locations, pixel values, or both?



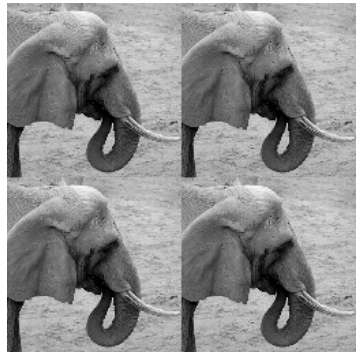
Input



Output



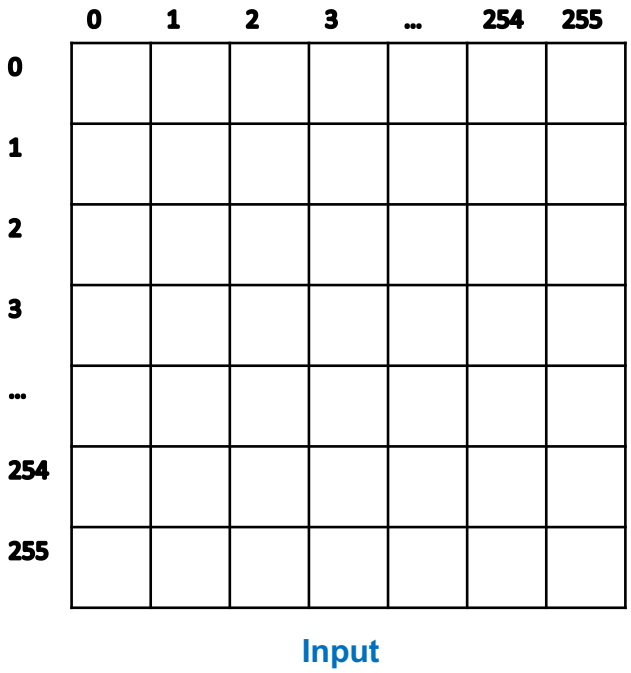
Input



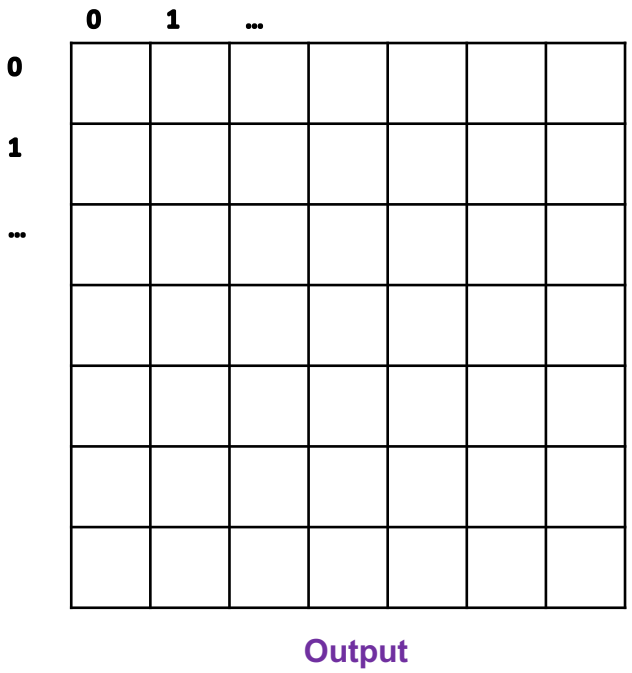
Output

Practice: Zoom

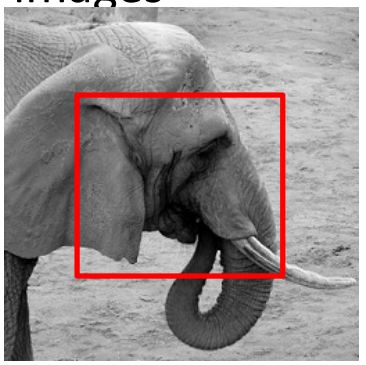
- Perform a "zoom" operation from (SIZE/4, SIZE/4) to (3*SIZE/4, 3*SIZE/4)
  - Consider the coordinates to visit from the input and output images
  - Ask yourself: Are we transforming pixel locations, pixel values, or both?



Input



Output



Input



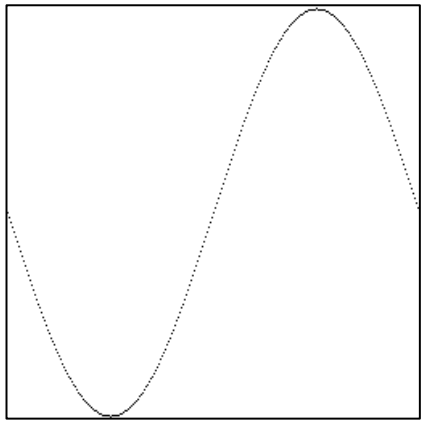
Output

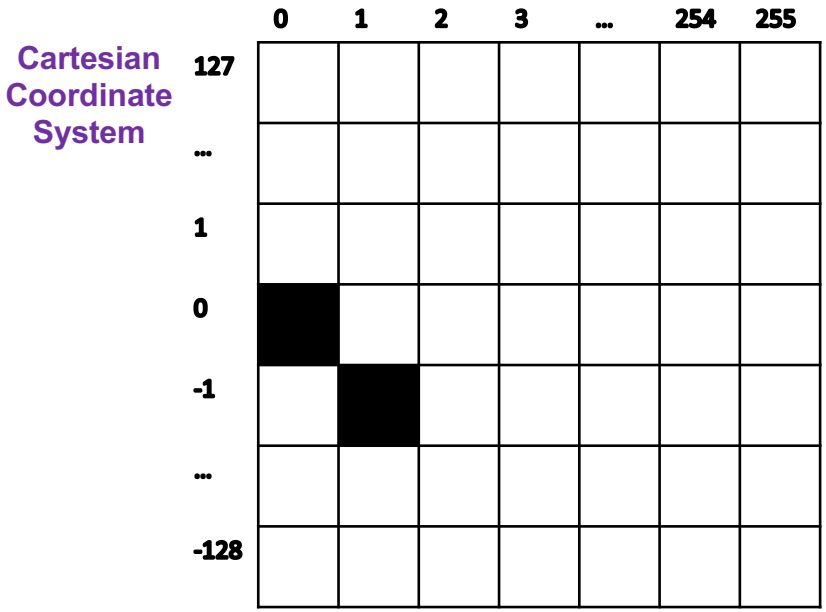
Selected Grayscale Solutions

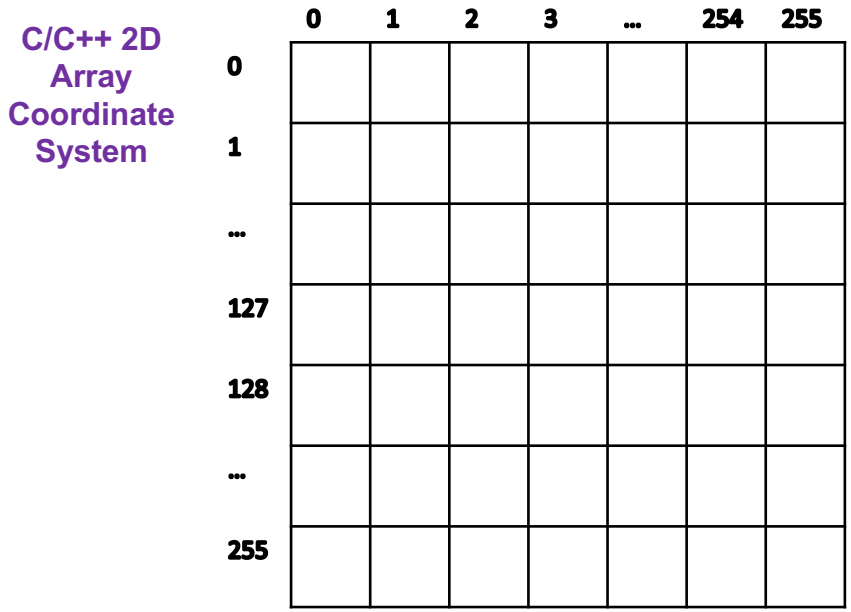
- **X**
  - http://bytes.usc.edu/files/cs103/graphics/x.cpp
- **Sin**
  - http://bytes.usc.edu/files/cs103/graphics/sin.cpp
- **Diagonal Gradient**
  - http://bytes.usc.edu/files/cs103/graphics/gradient_diag.cpp
- **Elephant-flip**
  - http://bytes.usc.edu/files/cs103/graphics/eg3-4.cpp
- **Elephant-tile**
  - http://bytes.usc.edu/files/cs103/graphics/eg3-5.cpp
- **Elephant-zoom**
  - http://bytes.usc.edu/files/cs103/graphics/zoom.cpp

Challenge: Drawing a sine-wave

- Draw a **single** period of a sine wave where y = -sin(x)
  - Tip: Carefully consider the difference and convert between `(x,y)` and `[i][j]` coordinates

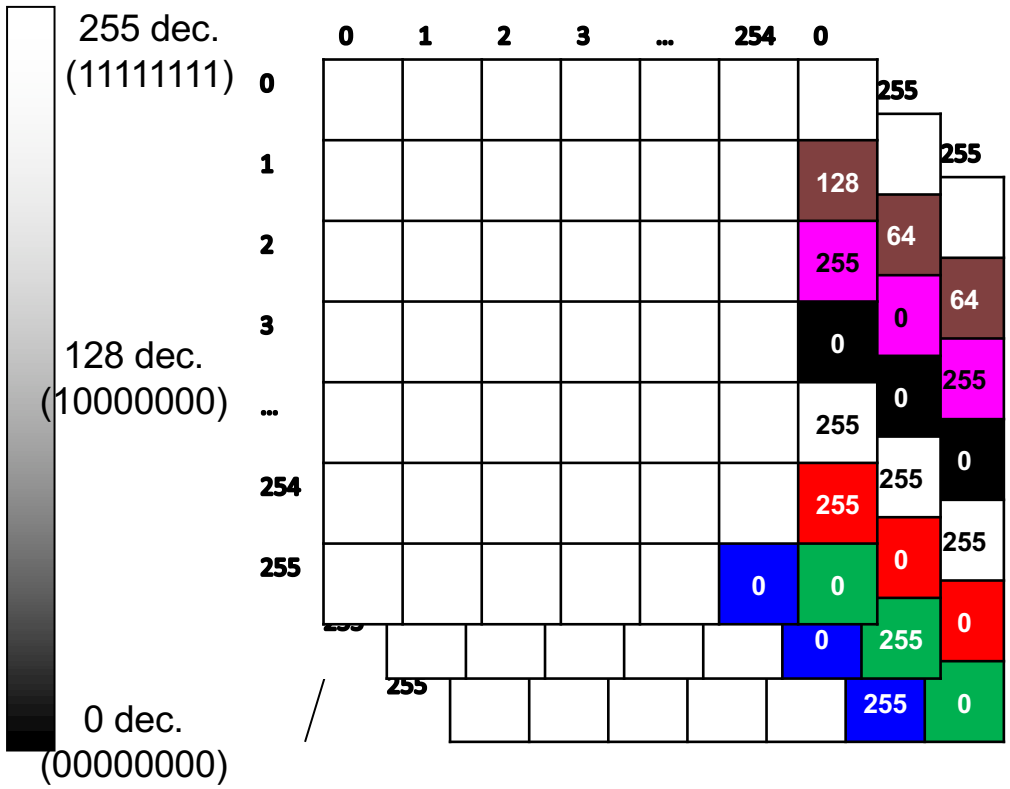






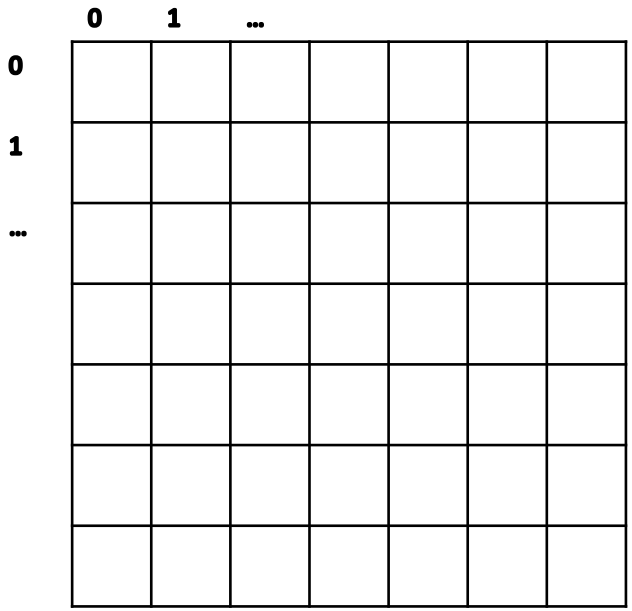
Color Images

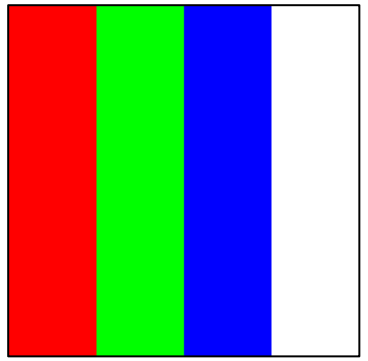
- Color images are represented as 3D arrays (3 planes corresponding to red, green, blue)
  - 256 rows x 256 columns x 3 planes
  - Thus, our images are 3D arrays of type:
    `unsigned char[SIZE][SIZE][3]`
- We will assume:
  - Plane 0 corresponds to red
  - Plane 1 corresponds to green
  - Plane 2 corresponds to blue
- By combining / overlaying the corresponding R, G, B values we can produce arbitrary colors



Practice: Color Bars

- Generate vertical color bars: Red, Green, Blue, White, every 1/4 of the screen





Color Image Processing Examples

- Base Image
- Each color plane inverted
- Grayscaled
  - Using NTSC formula:
    .299R + .587G + .114B





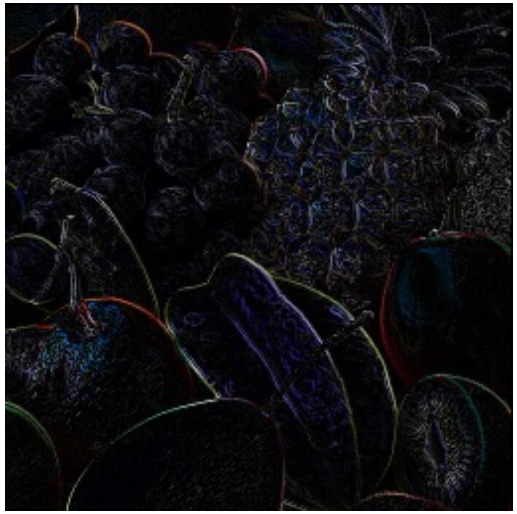


More Effects/Filters (1)

- Glass filter
  - Each destination pixel is from a random nearby source pixel
    - http://bytes.usc.edu/files/cs103/graphics/glass.cpp



- Edge detection
  - Each destination pixel is the difference of a source pixel with its south-west neighbor



More Effects/Filters (2)

- Smooth
  - Each destination pixel is average of 8 neighbors
    - http://bytes.usc.edu/files/cs103/graphics/smooth.cpp



Original



Smoothed

Selected Color Solutions

- **Color fruit – Inverted**
  - http://bytes.usc.edu/files/cs103/graphics/eg4-1.cpp
- **Color fruit – Grayscale**
  - http://bytes.usc.edu/files/cs103/graphics/eg4-3.cpp
- **Color fruit – Glass Effect**
  - http://bytes.usc.edu/files/cs103/graphics/glass.cpp
- **Color fruit – Edge Detection**
  - http://bytes.usc.edu/files/cs103/graphics/eg5-4.cpp
- **Color fruit – Smooth**
  - http://bytes.usc.edu/files/cs103/graphics/smooth.cpp

Aliases for a type name

TYPEDEF'S

typedefs

- **typedefs** allow us to create an "alias" for a data-type.
- Format:

```
typedef <existing_type> <alias_name>
```

- Examples:

```
typedef int score_t;
score_t x; // x is an int
```

OR

```
typedef double decimal_t;
decimal_t y;  // y is a double
```

- Makes code more readable and easy to maintain as well as saving keystrokes
  - See example below

```
int getScore(int scores[], int index)
{
   return scores[index];
}
void print(int scores[], int len){
   for(int i=0; i < len; i++){
     cout << scores[i] << endl;
   }
}
```

What if we decide we want to change scores to be doubles rather than integers. Could we just Find..Replace all 'int' with 'double'?

```
typedef int SCORE_T;
// simply change to typedef double SCORE_T;
SCORE_T getScore(SCORE_T scores[], int index)
{
   return scores[index];
}
void print(SCORE_T scores[], int len){
   for(int i=0; i < len; i++){
     cout << scores[i] << endl;
   }
}
```

Becomes easy to change types. Simply update the typedef.

More typedefs

- In CS104 you'll see VERY long types and variable declarations like:
  - `std::map<std::string, std::vector<int>> my_map;`
- Rather than having to repeat that in many places, we could make a typedef:
  - `typedef std::map<std::string, std::vector<int>> Records;`
- Then anywhere we'd use that type in a function prototype or in declaring a variable, we could just write:
  - `Records my_map;`