Software Design

Multi-Dimensional Arrays

Introduction

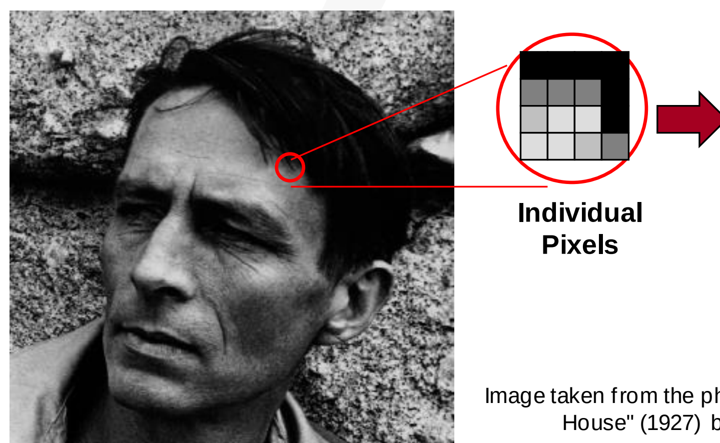
- Thus far arrays can be thought of 1-dimensional (linear) sets
  - only indexed with 1 value (coordinate)
  - char x[6] = {1,2,3,4,5,6};
- We often want to view our data as 2-D, 3-D or higher dimensional data
  - Matrix data
  - Images (2-D)
  - Index w/ 2 coordinates (row,col)

0	1	2	3	4	5	
01	02	03	04	05	06	…

Memory

Column Index

Row Index



Individual Pixels

0	0	0	0
64	64	64	0
128	192	192	0
192	192	128	64

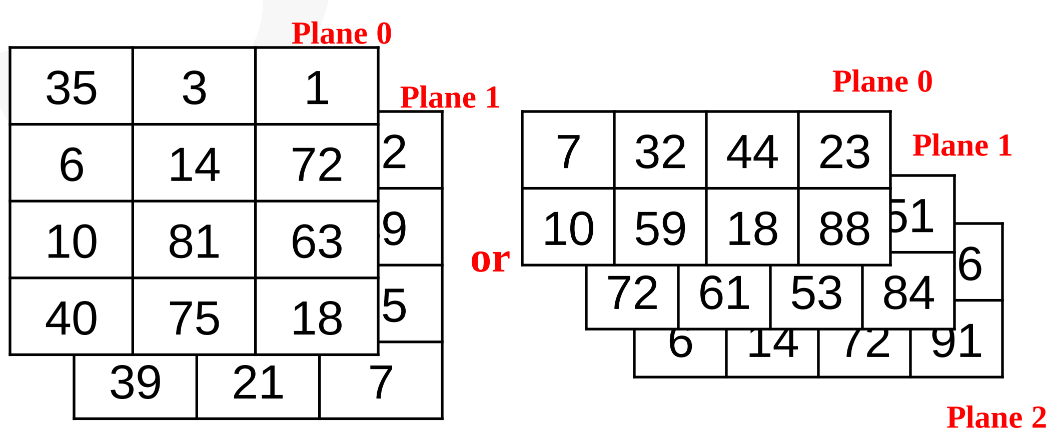
Image taken from the photo "Robin Jeffers at Ton House" (1927) by Edward Weston

MD Array Declaration

- 2D: Provide size along both dimensions (normally rows first then columns)
  - Access w/ 2 indices
  - Declaration: `int my_matrix[2][3];`
  - Access elements with appropriate indices
    - my_matrix[0][1] evals 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: Access data w/ 3 indices
  - Declaration: `char image[2][4][3];`
  - Up to human to interpret meaning of dimensions
    - Planes x Rows x Cols
    - Rows x Cols x Planes



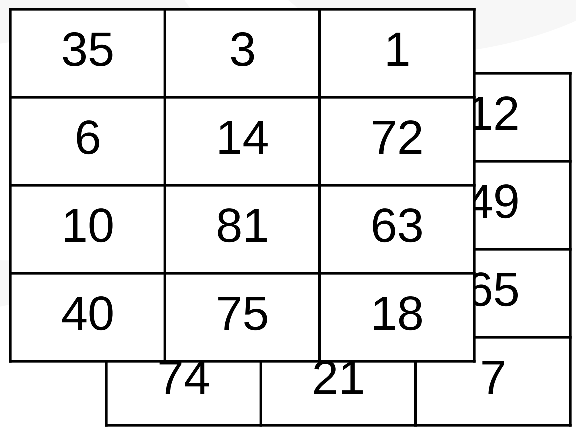
Passing MD Arrays

- **Formal Parameter: Must give dimensions of all but first dimension**
- **Actual Parameter: Still just the array name, i.e., starting address**
- **Why do we have to provide all but the first dimension?**
- **So that the computer can determine where element: data[i][j][k] is actually located in memory**

```
void doit(int my_array[][4][3])
{
  my_array[1][3][2] = 5;
}

int main(int argc, char *argv[])
{
  int data[2][4][3];

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



0	35
1	03
2	01
3	06
4	14
…	…
11	18
12	42
13	08
14	12

Memory

Linearization of MD Arrays

- Analogy: Hotel room layout => 3D
  - Access location w/ 3 indices:
    - Floors, Aisles, Rooms
    - But they don't give you 3 indices, they give you one room number
  - Room #s are a linearization of the 3 dimensions
    - Room 218 => Floor=2, Aisle 1, Room 8
- When "linear"-izing we keep proximity for only lowest dimension
  - Room 218 is next to 217 and 219
- But we lose some proximity info for higher dimensions
  - Presumably room 218 is right below room 318
  - But in the linearization 218 seems very far from 318

	1st Floor	
100		110
101		111
102		112
103		113
104		114
105		115
106		116
107		117
108		118
109		119

	2nd Floor	
200		220
201		211
202		212
203		213
204		214
205		215
206		216
207		217
208		218
209		219

Analogy: Hotel Rooms

1st Digit = Floor
2nd Digit = Aisle
3rd Digit = Room

Linearization of MD Arrays (cont.)

- In a computer, multidimensional arrays must still be stored in memory which is addressed linearly (1-Dimensional)
- C/C++ use a policy that lower dimensions are placed next to each other followed by each higher level dimension

```
int x[2][3];
```

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

Memory

100	00	00	00	05	x[0][0]
104	00	00	00	03	x[0][1]
108	00	00	00	01	x[0][2]
112	00	00	00	06	x[1][0]
116	00	00	00	04	x[1][1]
120	00	00	00	02	x[1][2]
124	d2	19	2d	81	
	…				

Linearization of MD Arrays (cont.)

- In a computer, multidimensional arrays must still be stored in memory which is addressed linearly (1-Dimensional)
- C/C++ use a policy that lower dimensions are placed next to each other followed by each higher level dimension

char y[2][4][3];

35	3	1
6	14	72
10	81	63
40	75	18

42	8	12
67	25	49
14	48	65
74	21	7

0	35
1	03
2	01
3	06
4	14
…	…
11	18
12	42
13	08
14	12

Memory

Linearization (cont.)

- We could re-organize the memory layout (i.e. linearization) while still keeping the same view of the data by changing the order of the dimensions

char y[4][3][2];

35	3	1
6	14	72
10	81	63
40	75	18

42	8	12
67	25	49
14	48	65
74	21	7

0	35
1	42
2	03
3	08
4	01
5	12
6	06
7	67
8	14
…	…

Memory

Linearization (cont.)

- Formula for location of item at row i, column j in an array with NUMR rows and NUMC columns:

Declaration: int x[2][3]; // NUMR=2, NUMC = 3;

	Col. 0	Col. 1	Col. 2
Row 0	5	3	1
Row 1	6	4	2
Row 2	8	9	7
Row 3	15	3	6

Access: x[i][j]:

100	00	00	00	05	x[0][0]
104	00	00	00	03	x[0][1]
108	00	00	00	01	x[0][2]
112	00	00	00	06	x[1][0]
116	00	00	00	04	x[1][1]
120	00	00	00	02	x[1][2]
124	00	00	00	08	x[2][0]
128	00	00	00	09	x[2][1]
132	00	00	00	07	x[2][2]
136	00	00	00	0f	x[3][0]
140	00	00	00	03	x[3][1]
144	00	00	00	06	x[3][2]
	…				

Memory

Linearization (cont.)

- Formula for location of item at plane p, row i, column j in array with NUMP planes, NUMR rows, and NUMC columns

Declaration: int x[2][4][3]; // NUMP=2, NUMR=4, NUMC=3

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

35	3	1
6	14	72
10	81	63
40	75	18

42	8	12
67	25	49
14	48	65
74	21	7

100	35
104	03
108	01
116	06
120	14
…	…

Memory

Revisited: Passing MD Arrays

- Must give dimensions of all but first dimension
- This is so that when you use 'myarray[p][i][j]' the computer can determine where in the linear addresses, that individual index is located
  - [p][i][j] =

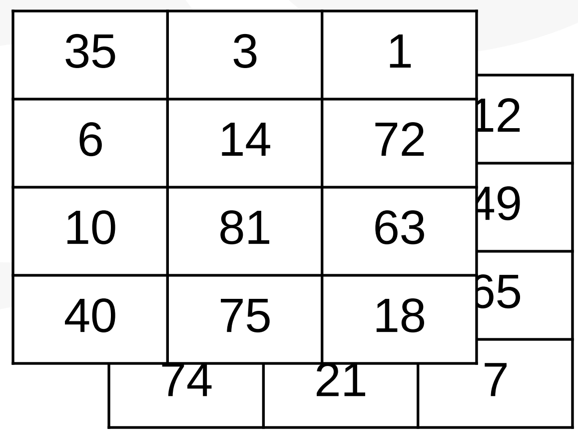
    start address + {p*NUMR*NUMC + i*NUMC + j} * sizeof(int)
  - [1][3][2] in an array of nx4x3

    becomes: 1*(4*3) + 3(3) + 2 = 23 ints = 23*4 = 92 bytes into the array

```
void doit(int my_array[][4][3])
{
  my_array[1][3][2] = 5;
}

int main(int argc, char *argv[])
{
  int data[2][4][3];

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



100	35
104	03
108	01
112	06
116	14
…	…
144	18
148	42
152	08
156	12

Memory

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

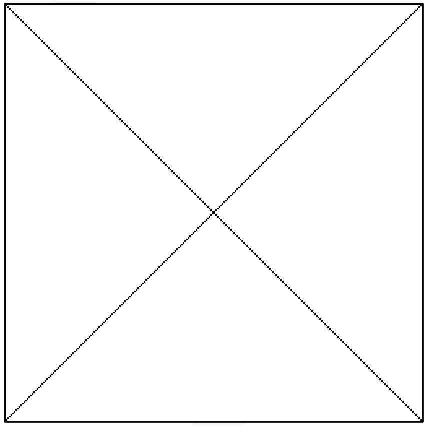
IMAGE PROCESSING

Practice: Drawing

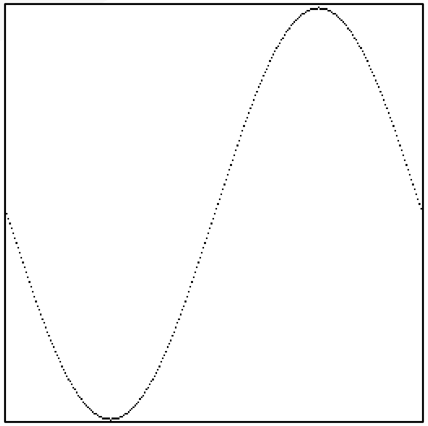
- Download the BMP library code:
  - In your examples directory on your VM
    - $ mkdir gradient; cd gradient
    - $ tar -xvf gradient.tar
    - $ make
    - $ ./demo
    - $ eog cross.bmp &
  - Code to read (open) and write (save) .BMP files is provided in bmplib.h and bmplib.cpp
  - Look at bmplib.h for the prototype of the functions you can use in your main() program in demo.cpp
- demo.cpp contains a main function and two global arrays: image[255][255] and rgbimage[255][255][3]
  - bwimage is a 256x256 image with grayscale pixels (0=black, 255=white)

Practice: Drawing

- Draw an X on the image
  - Try to do it with only a single loop, not two in sequence

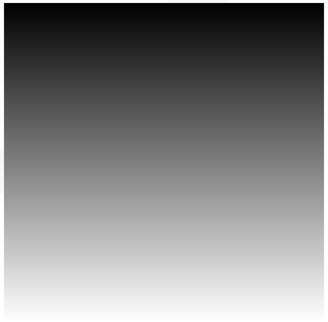


- Draw a single period of a sine wave
  - Hint: enumerate each column, x, with a loop and figure out the appropriate row (y-coordinate)



Practice: Drawing

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



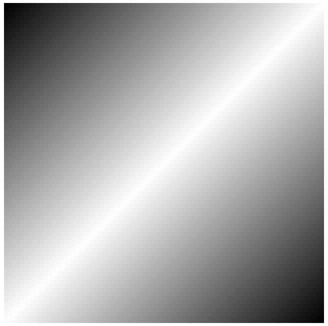
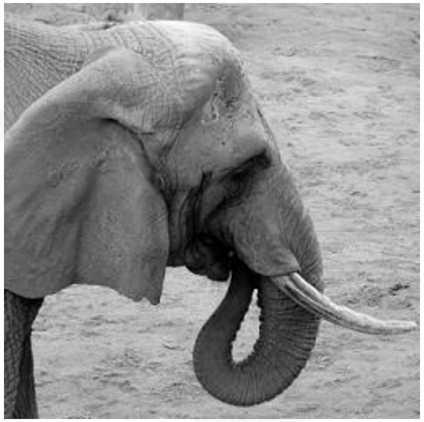


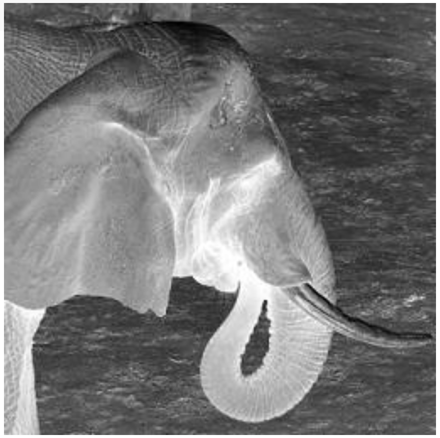
Image Processing

- Go to your gradient directory
  - elephant.bmp
- Here is a first exercise...produce the "negative"



Original

```
#include "bmplib.h"
unsigned char image[SIZE][SIZE];
int main() {
   readGSBMP("elephant.bmp", image);
   for (int i=0; i<SIZE; i++) {
      for (int j=0; j<SIZE; j++) {
         image[i][j] = 255-image[i][j];
         // invert color
      }
   }
   showGSBMP(image);
   return 0;
}
```

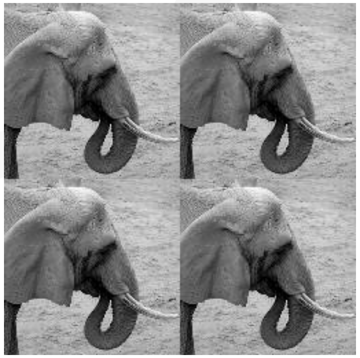


Inverted

Practice: Image Processing

- Diagonal flip

- Tile



- Zoom

Selected Grayscale Solutions

- X
  - x.cpp
- Sin
  - sin.cpp
- Diagonal Gradient
  - gradient_diag.cpp
- Elephant-flip
  - ef.cpp
- Elephant-tile
  - et.cpp
- Elephant-zoom
  - ezoom.cpp

Color Images

- Color images are represented as 3D arrays (256x256x3)
  - The lower dimension are Red, Green, Blue values
- Base Image
- Each color plane inverted
- Grayscaled
  - Using NTSC formula: .299R + .587G + .114B

Color Images (cont.)

- Glass filter
  - Each destination pixel is from a random nearby source pixel
- Edge detection
  - Each destination pixel is the difference of a source pixel with its south-west neighbor

Color Images (cont.)

- **Smooth**
  - **Each destination pixel is average of 8 neighbors**

Original

Smoothed

Selected Color Solutions

- Color fruit – Inverted
  - eg4-1.cpp
- Color fruit – Grayscale
  - eg4-3.cpp
- Color fruit – Glass Effect
  - glass.cpp
- Color fruit – Edge Detection
  - g5-4.cpp
- Color fruit – Smooth
  - smooth.cpp

ENUMERATIONS

Enumerations

- Associates an integer (number) with a symbolic name
- `enum [optional_collection_name] {Item1, Item2, … ItemN}`
  - Item1 = 0
  - Item2 = 1
  - …
  - ItemN = N-1
- Use symbolic item names in your code and compiler will replace the symbolic names with corresponding integer values

```
const int BLACK=0;
const int BROWN=1;
const int RED=2;
const int WHITE=7;

int pixela = RED;
int pixelb = BROWN;
...
```

Hard coding symbolic names with given codes

```
// First enum item is associated with 0
enum Colors {BLACK,BROWN,RED,...,WHITE};

int pixela = RED;   // pixela = 2;
int pixelb = BROWN; // pixelb = 1;
```

Using enumeration to simplify

Exercise

- Explore C++ image processing packages such as BOOST GIL (http://www.boost.org/doc/libs/1_43_0/libs/gil/doc/index.html) or CImg (http://cimg.eu)
- Practice drawing functions in 3D