CS103 Unit 5 - Arrays

Mark Redekopp

ARRAY BASICS

Motivating Example

- Suppose I need to store the grades for all students so I can then compute statistics, sort them, print them, etc.
- I would need to store them in variables that I could access and use
  - This is easy if I have 3 or 4 students
  - This is painful if I have many students
- Enter **<u>arrays</u>**
  - Collection of many variables referenced by **<u>one name</u>**
  - Individual elements can be accessed with an integer index

```
int main()
{
  int score1, score2, score3;
  cin >> score1 >> score2 >> score3;

  // output scores in sorted order
  if(score1 < score2 &&
     score1 < score3)
  {  /* score 1 is smallest */ }

  /* more */
}
```

```
int main()
{
  int score1,  score2,  score3,
      score4,  score5,  score6,
      score7,  score8,  score9,
      score10, score11, score12,
      score13, score14, score15,
      /* ... */
      score139, score140;
  cin >> score1 >> score2 >> score3
      >> score4 >> score5 >> score6
      /* ... */
```

Arrays: Informal Overview

- Informal Definition:
  - Ordered collection of variables of the same type
- Collection is referred to with <u>one name</u>
- Individual elements referred to by an **offset/index** from the start of the array [in C, first element is at index 0]

Just as an apartment building is known by 1 address but many apartment numbers, an array has one name but can use integer indices to access individual elements

```
int data[20];
data[0] = 103;
data[1] = -1;
data[2] = data[0]+1;
```

data[0]	data[1]	data[2]		data[19]
120	124	128		196
103	-1	104	…	404

Memory

```
char  A[3] = "hi";
```

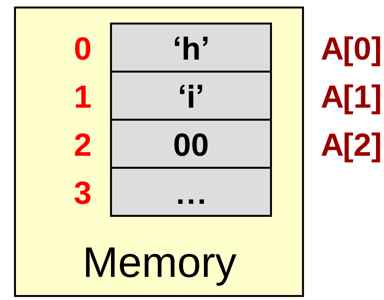
A[0]	A[1]	A[2]				
860	861	862	863	…		
‘h’	‘i’	00	09	05	04	…

Memory

Arrays

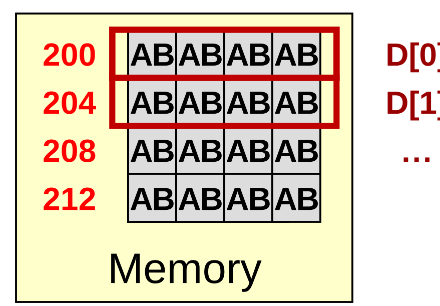
- Formal Def: A statically-sized, contiguously allocated collection of homogenous data elements
- Collection of homogenous data elements
  - Multiple variables of the same data type
- Contiguously allocated in memory
  - One right after the next
- Statically-sized
  - Size of the collection must be a constant and can't be changed after initial declaration/allocation
- Collection is referred to with **one name**
- Individual elements referred to by an **offset/index** from the start of the array [in C, first element is at index 0]

char A[3] = "hi";

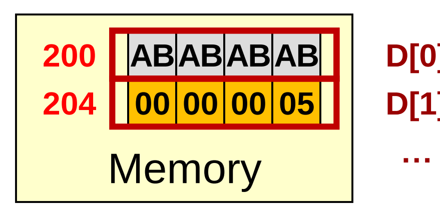


char c = A[0]; // 'h'

int D[20];

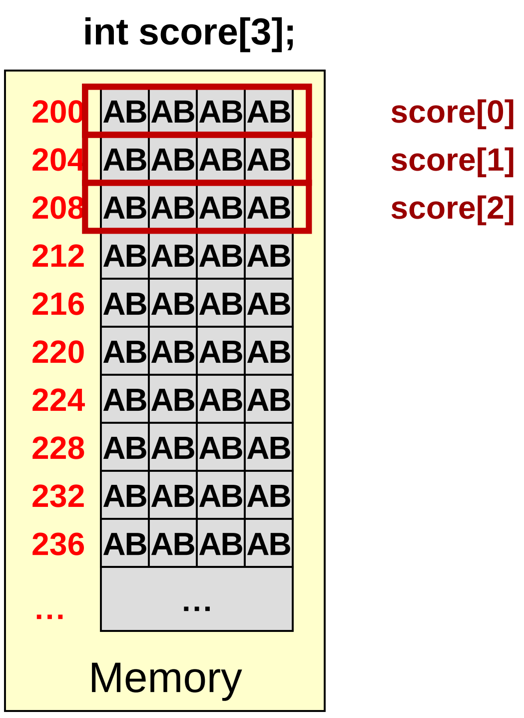


D[1] = 5;



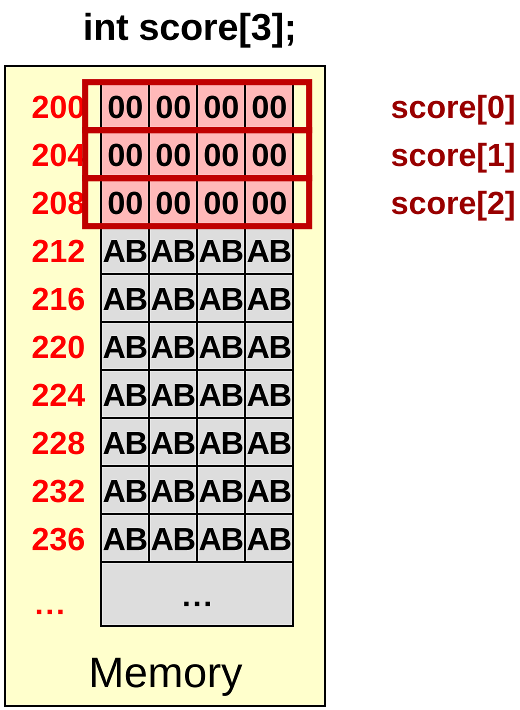
Example: Arrays

- Track the score of 3 players
- Homogenous data set (amount) for multiple people…perfect for an array
  - `int score[3];`
- Recall, memory has garbage values by default. You will need to initialized each element in the array



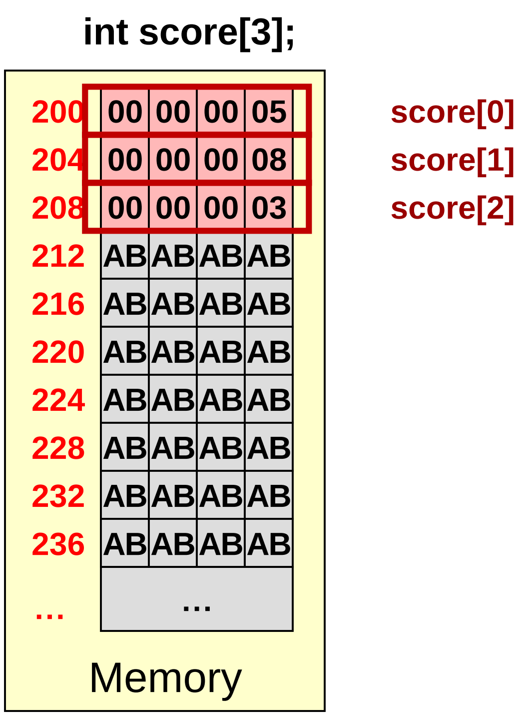
Example: Arrays

- Track the score of 3 players
- Homogenous data set (amount) for multiple people…perfect for an array
  - `int score[3];`
- Must initialize elements of an array
  - ```
    for(int i=0; i < 3; i++)
        score[i] = 0;
    ```


    ```

Arrays

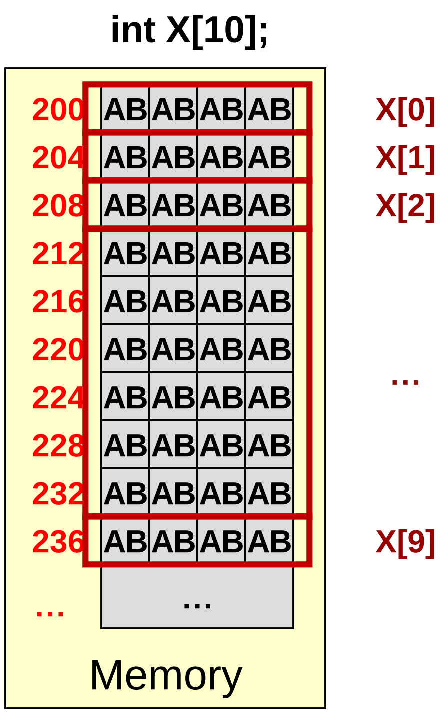
- Track the score of 3 players
- Homogenous data set (amount) for multiple people…perfect for an array
  - `int score[3];`
- Must initialize elements of an array
  - ```
    for(int i=0; i < 3; i++)
        score[i] = 0;
    ```
- Can access each persons amount and perform ops on that value
  - ```
    score[0] = 5;
    score[1] = 8;
    score[2] = score[1] - score[0]
    ```



ARRAY ODDS AND ENDS

Static Size/Allocation

- For now, arrays must be declared as fixed size (i.e. a constant known at compile time)
  - Good:
    - `int x[10];`
    - `#define MAX_ELEMENTS  100`
      `int x[MAX_ELEMENTS];`
    - `const int MAX_ELEMENTS = 100;`
      `int x[MAX_ELEMENTS];`
  - Bad:
    - `int mysize;`
      `cin >> mysize;`
      `int x[mysize];`
    - `int mysize = 10;`
      `int x[mysize];`



Compiler must be able to figure out how much memory to allocate at compile-time

Initializing Arrays

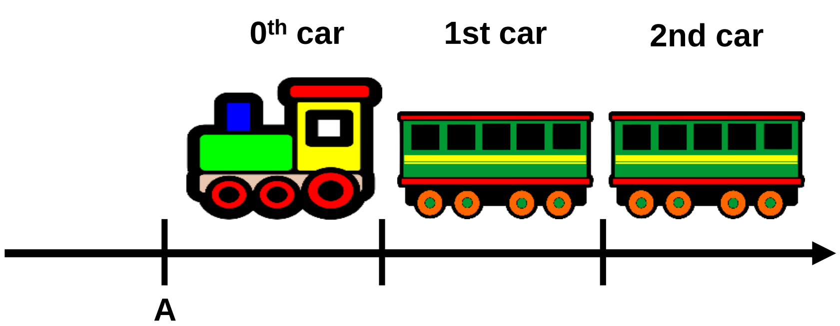
- Integers or floating point types can be initialized by placing a comma separated list of values in curly braces {…}
  - `int data[5] = {4,3,9,6,14};`
  - `char vals[8] = {64,33,18,4,91,76,55,21};`
  - `int vals[100] = {1,2,3};`
    - `If not enough values provided, the remaining elements will be initialized to 0`
- If accompanied w/ initialization list, size doesn't have to be indicated (empty [ ])
  - `double stuff[] = {3.5, 14.22, 9.57}; // = stuff[3]`
- However the list must be of constants, not variables:
  - BAD: `double z = 3.5;     double stuff[] = {z, z, z};`

Understanding array addressing and indexing

ACCESSING DATA IN AN ARRAY

Exercise

- Consider a train of box cars
  - The initial 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?



More on Accessing Elements

- Assume a 5-element int array
  - int x[5] = {8,5,3,9,6};
- When you access x[2], the CPU calculates where that item is in memory by taking the start address of x (i.e. 100) and adding the product of the index, 2, times the size of the data type (i.e. int = 4 bytes)
  - x[2] => int. @ address 100 + 2*4 = 108
  - x[3] => int. @ address 100 + 3*4 = 112
  - x[i] @ start address of array + i * (size of array type)
- C does not stop you from attempting to access an element beyond the end of the array
  - x[6] => int. @ address 100 + 6*4 = 124 (Garbage!!)

100	00	00	00	08	x[0]
104	00	00	00	05	x[1]
108	00	00	00	03	x[2]
112	00	00	00	09	x[3]
116	00	00	00	06	x[4]
120	a4	34	7c	f7	
124	d2	19	2d	81	
	…				

Memory

Compiler must be able to figure out how much memory to allocate at compile-time

Fun Fact 1: If you use the **name of an array** w/o square brackets it will evaluate to the **starting address** in memory of the array (i.e. address of 0^{th} entry)
Fun Fact 2: Fun Fact 1 usually appears as one of the first few questions on the midterm.

Intermediate-Level Array Topics

Passing arrays to other functions

ARRAYS AS ARGUMENTS

Passing Arrays As Arguments

- Syntax:
  - **Step 1**: In the prototype/signature: Put empty square brackets after the **formal** parameter name if it is an array (e.g. `int data[]`)
  - **Step 2**: When you call the function, just provide the name of the array as the **actual** parameter
    - **In C/C++ using an array name without any index evaluates to the starting address of the array**

```
// Function that takes an array
int sum(int data[], int size);

int sum(int data[], int size)
{
  int total = 0;
  for(int i=0; i < size; i++){
    total += data[i];
  }
  return total;
}

int main()
{
  int vals[100];
  /* some code to initialize vals */
  int mysum = sum(vals, 100);
  cout << mysum << endl;
     // prints sum of all numbers
  return 0;
}
```

Pass-by-Value & Pass-by-Reference

- What are the pros and cons of emailing a document by:
  - Attaching it to the email
  - Sending a link (URL) to the document on some cloud service (etc. Google Docs)

- **Pass-by-value** is like emailing an attachment
  - A **copy** is made and sent
- **Pass-by-reference** means emailing a link to the original
  - **No copy is made** and **any modifications by the other party are seen by the originator**

Arrays And Pass-by-Reference

- Single (scalar) variables are **passed-by-value** in C/C++
  - Copies are passed
- Arrays are
- **passed-by-reference**
  - Links are passed
  - This means any change to the array by the function is visible upon return to the caller

```
void dec(int);
int main()
{
  int y = 3;
  dec(y);
  cout << y << endl;
  return 0;
}
void dec(int y)
{  y--;  }
```

Single variables (aka scalars) are passed-by-value but arrays are passed-by-reference

```
void init(int x[], int size);
int main()
{
  int data[10];
  init(data, 10);
  cout << data[9] << endl;
     // prints 0
  return 0;
}
void init(int x[], int size)
{  // x is really a link to data
   for(int i=0; i < size; i++){
     x[i] = 0; // changing data[i]
   }
}
```

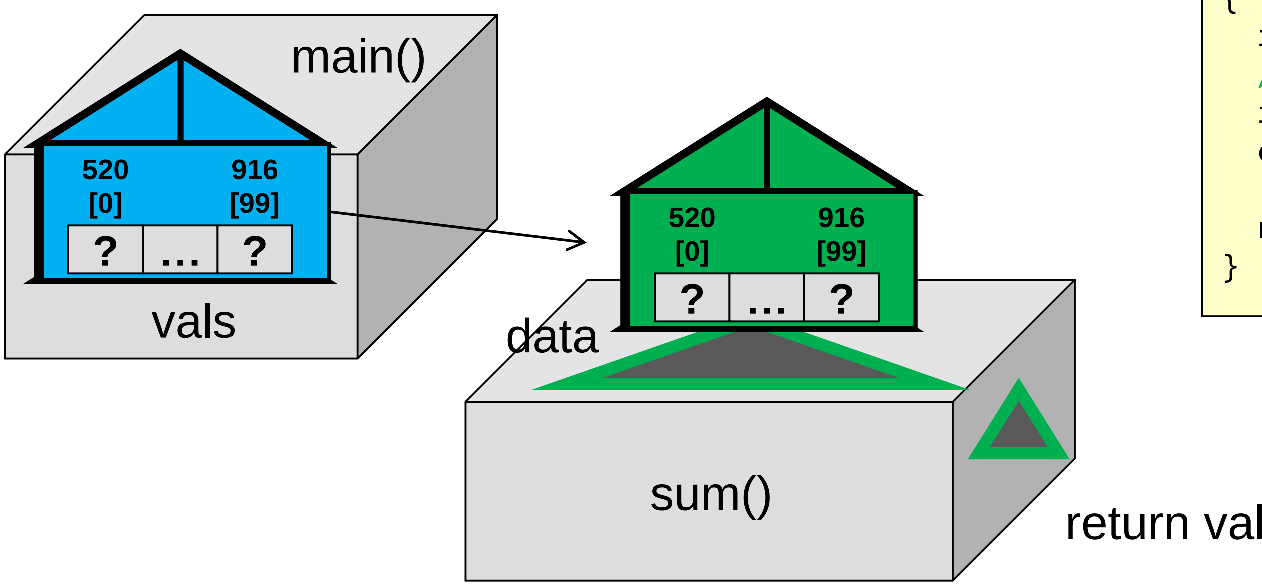
But Why?

- If we used pass-by-value then we'd have to make a copy of a potentially HUGE amount of data (what if the array had a million elements)
- To avoid copying vast amounts of data, we pass a link

```
// Function that takes an array
int sum(int data[], int size);

int sum(int data[], int size)
{
  int total = 0;
  for(int i=0; i < size; i++){
    total += data[i];
  }
  return total;
}

int main()
{
  int vals[100];
  /* some code to initialize vals */
  int mysum = sum(vals, 100);
  cout << mysum << endl;
     // prints sum of all numbers
  return 0;
}
```



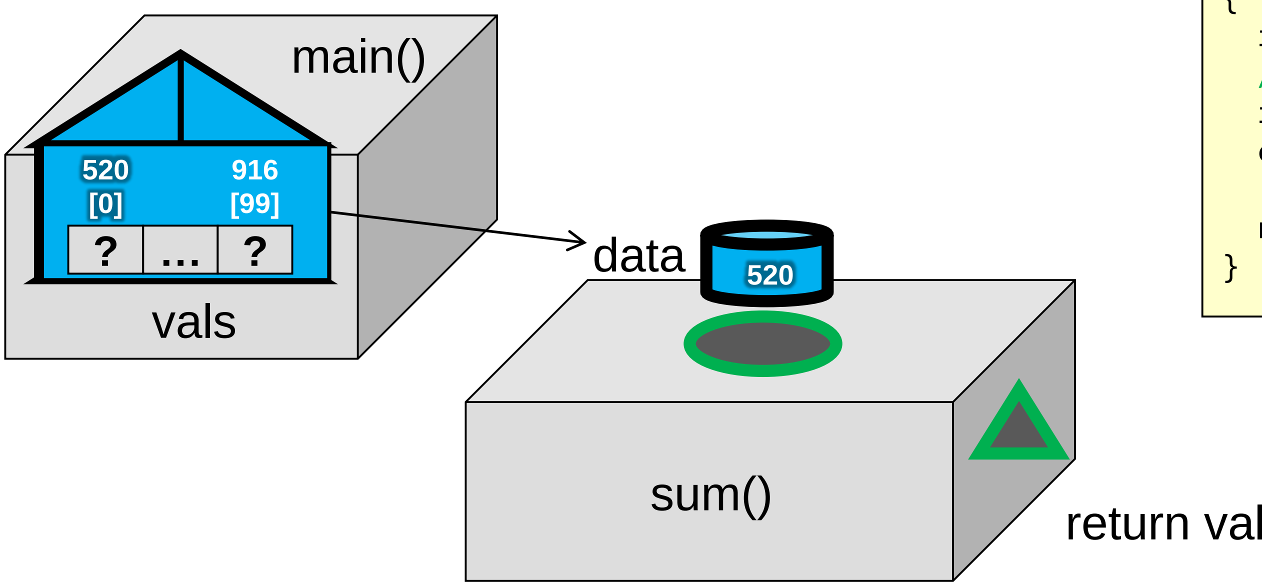
So What Is Actually Passed?

- The "link" that is passed is just the starting address (e.g. 520) of the array in memory
- The called function can now use 520 to access the original array (read it or write new values to it)

```
// Function that takes an array
int sum(int data[], int size);

int sum(int data[], int size)
{
  int total = 0;
  for(int i=0; i < size; i++){
    total += data[i];
  }
  return total;
}

int main()
{
  int vals[100];
  /* some code to initialize vals */
  int mysum = sum(vals, 100);
  cout << mysum << endl;
     // prints sum of all numbers
  return 0;
}
```



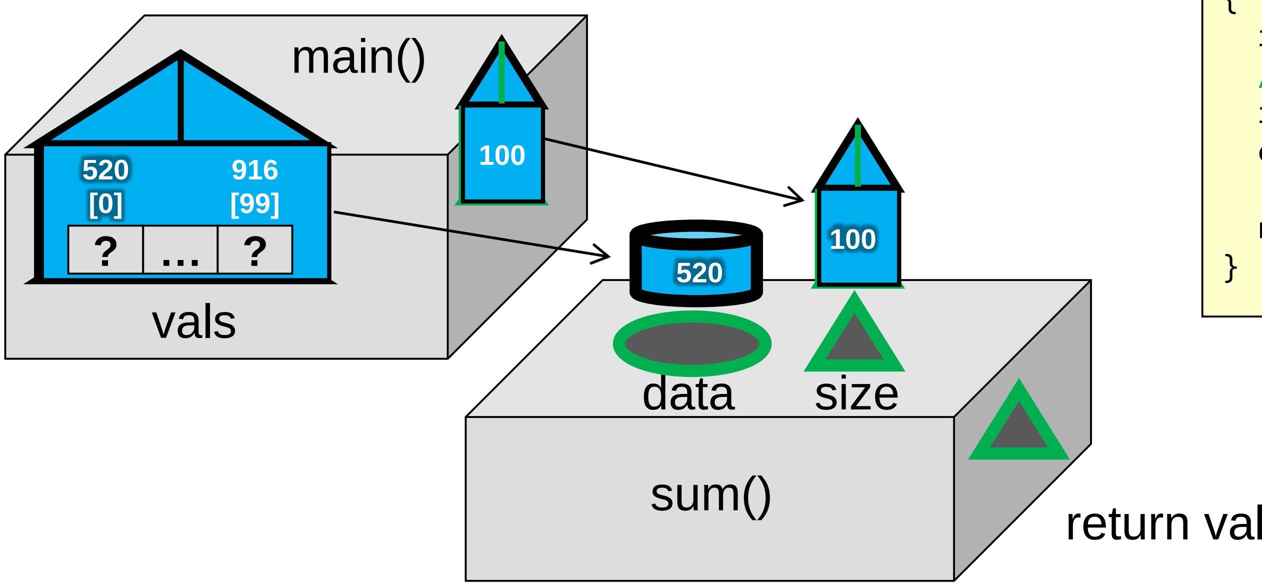
Arrays in C/C++ vs. Other Languages

- Notice that if sum() only has the start address it **would not know** how big the array is
- **Unlike Java** or other languages where you can call some function to give the size of an array, **C/C++ require you to track the size yourself in a separate variable and pass it as a secondary argument**

```
// Function that takes an array
int sum(int data[], int size);

int sum(int data[], int size)
{
  int total = 0;
  for(int i=0; i < size; i++){
    total += data[i];
  }
  return total;
}

int main()
{
  int vals[100];
  /* some code to initialize vals */
  int mysum = sum(vals, 100);
  cout << mysum << endl;
     // prints sum of all numbers
  return 0;
}
```



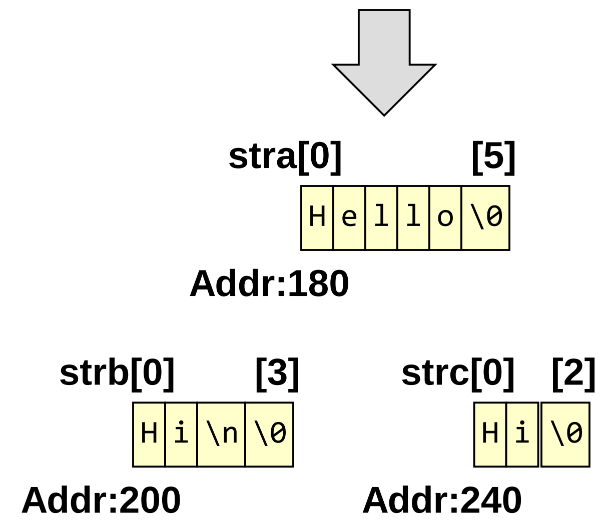
Null terminated character arrays

C-STRINGS

C Strings

- Character arrays (i.e. C strings)
  - Enclosed in double quotes " "
  - Strings of text are simply arrays of chars
  - Can be initialized with a normal C string (in double quotes)
  - C strings have one-byte (char) per character
  - End with a "null" character = 00 dec. = '\0' ASCII
  - cout "knows" that if a char array is provided as an argument it will print the 0th character and keep printing characters until a ‘\0’ (null) character [really just a value of 0] is encountered
  - cin "knows" how to take in a string and fill in a char array (stops at whitepace)
    - Careful it will write beyond the end of an array if the user enters a string that is too long

```
#include<iostream>
using namespace std;
int main()
{
  char stra[6] = "Hello";
  char strb[] = "Hi\n";
  char strc[] = {'H','i','\0'};
  cout << stra << strb;
  cout << strc << endl;
  cout << "Now enter a string: ";
  cin >> stra;
  cout << "You typed: " << stra;
  cout << endl;
}
```



Example: C String Functions

- Write a function to determine the length (number of characters) in a C string
- Write a function to copy the characters in a source string/character array to a destination character array
- Edit and test your program and complete the functions:
  - int strlen(char str[])
  - strcpy(char dst[], char src[])
- Compile and test your functions
  - main() is complete and will call your functions to test them

Using arrays as a lookup table

LOOKUP TABLES

Arrays as Look-Up Tables

- Use the value of one array as the index of another
- Suppose you are given some integers as data [in the range of 0 to 5]
- Suppose computing squares of integers was difficult (no built-in function for it)
- Could compute them yourself, record answer in another array and use data to "look-up" the square

```
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};
// The LUT
int squares[6] = {0,1,4,9,16,25};
```

```
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};
// The LUT
int squares[6] = {0,1,4,9,16,25};

for(int i=0; i < 8; i++){
  int x = data[i]
  int x_sq = squares[x];
  cout << i << "," << sq[i] << endl;
}
```

```
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};
// The LUT
int squares[6] = {0,1,4,9,16,25};

for(int i=0; i < 8; i++){
  int x_sq = squares[data[i]];
  cout << i << "," << sq[i] << endl;
}
```

Example

- Using an array as a Look-Up Table
  - wget http://ee.usc.edu/~redekopp/cs103/cipher.cpp
  - Let's create a cipher code to encrypt text
  - `abcdefghijklmnopqrstuvwxyz` => `ghijklmaefnzyqbcdrstuopvwx`
  - char orig_string[] = "helloworld";
  - char new_string[11];
  - After encryption:
    - new_string = "akzzbpbrzj"
  - Define another array
    - char cipher[27] = "ghijklmaefnzyqbcdrstuopvwx";
    - How could we use the original character to index ("look-up" a value in) the cipher array

MULTIDIMENSIONAL ARRAYS

Multidimensional Arrays

- 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};

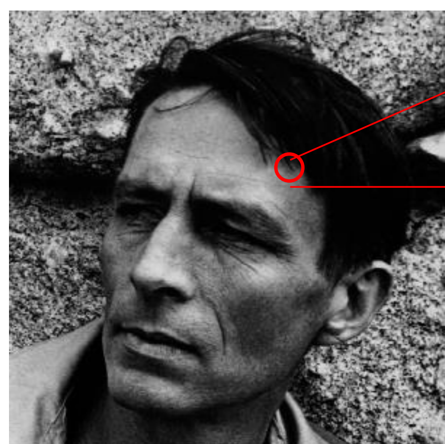
0	1	2	3	4	5	
01	02	03	04	05	06	…

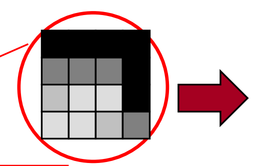
Memory

- 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)

Column Index

Row Index





Individual Pixels

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

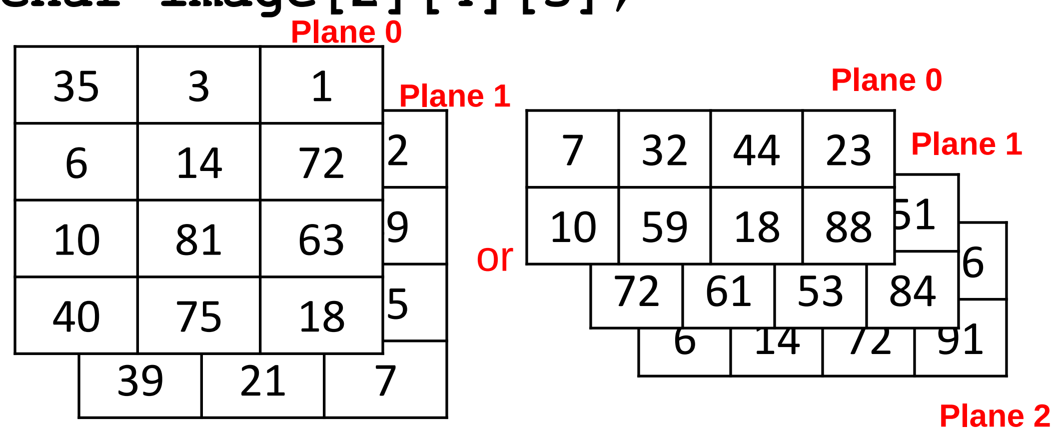
Image taken from the photo "Robin Jeffers at Ton House" (1927) by Edward Weston

Multidimension 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: **`unsigned char image[2][4][3];`**
  - Up to human to interpret meaning of dimensions
    - Planes x Rows x Cols
    - Rows x Cols x Planes



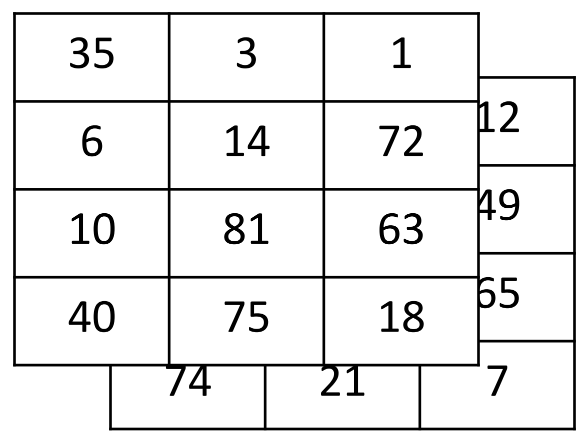
Passing Multi-Dimensional Arrays

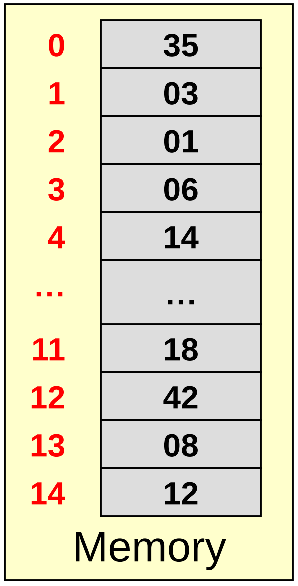
- Formal Parameter: Must give dimensions of all but first dimension
- Actual Parameter: Still just the array name (i.e. starting address)
- Why do we have to provide all but the first dimension?
- So that the computer can determine where element: data[i][j][k] is actually located in memory

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

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

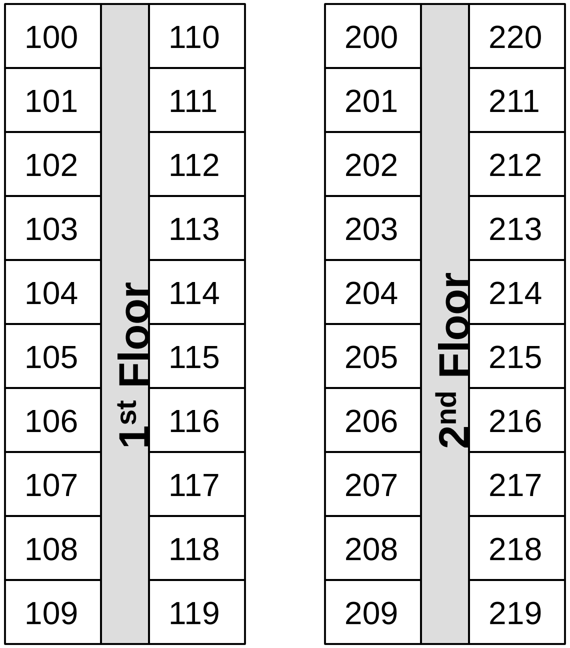
  doit(data);
  ...
  return 0;
}
```





Linearization of Multidimensional 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 one 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



Analogy: Hotel Rooms

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

Linearization of Multidimensional Arrays

- 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

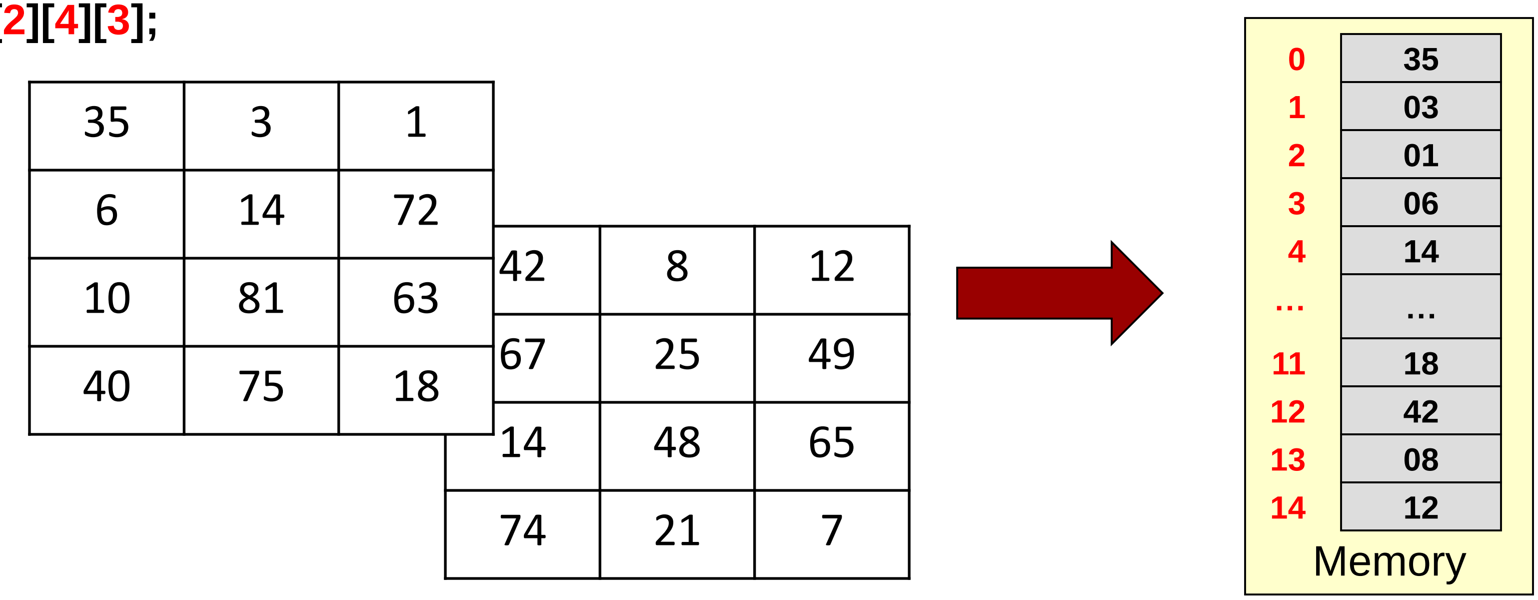
100	00	00	00	05	x[0][0]
104	00	00	00	03	x[0][1]
108	00	00	00	01	x[0][2]
112	00	00	00	06	x[1][0]
116	00	00	00	04	x[1][1]
120	00	00	00	02	x[1][2]
124	d2	19	2d	81	
	…				

Memory

Linearization of Multidimensional Arrays

- 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];
```



Linearization of Multidimensional Arrays

- 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 of Multidimensional Arrays

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

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

Address					Element
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 of Multidimensional Arrays

- 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

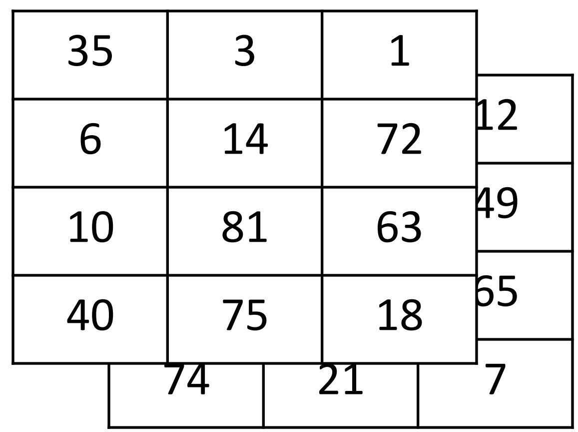
Revisited: Passing Multi-Dimensional Arrays

- Must give dimensions of all but first dimension
- This is so that when you use 'myarray[p][i][j]' the computer and determine where in the linear addresses that individual index is located in the array
  - `[p][i][j] = startAddr + (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 = address 192

```
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

- See Vocareum 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

- To download the code on your own Linux machine or VM
    - $ wget http://bytes.usc.edu/files/cs103/demo-bmplib.tar
    - $ tar -xvf demo-bmplib.tar
    - $ cd demo-bmplib
    - $ make
    - $ ./demo
    - $ eog cross.bmp &

Multi-File Programs

- We need a way to split our code into many separate files so that we can partition our code
  - We often are given code libraries from other developers or companies
  - It can also help to put groups of related functions into a file
- 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
- cross.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

Multi-file Compilation

- Three techniques to compile multiple files into a single application
  - Use 'make' with a 'Makefile' script
    - We will provide you a 'Makefile' whenever possible and it contains directions for how to compile all the files into a single program
    - To use it just type 'make' at the command prompt
  - Compile all the .cpp files together like:

    $ **compile gradient.cpp bmplib.cpp -o gradient**

    - Note: NEVER compile .h files

Multi-file Compilation

- Three techniques to compile multiple files into a single application
  - Compile each .cpp files separately into an "object file" (w/ the –c option) and then link them altogether into one program:

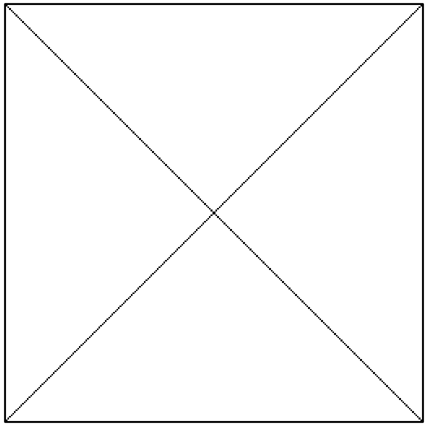
    $ compile **-c** bmplib.cpp -o bmplib.o

    $ compile **-c** gradient.cpp -o gradient.o

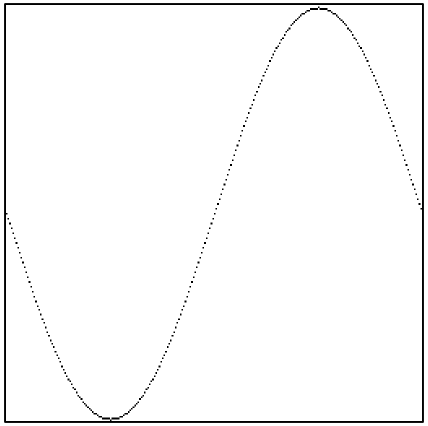
    $ compile gradient.o bmplib.o -o gradient
  - The first two command produce .o (object) files which are non-executable files of 1's and 0's representing the code
  - The last command produces an executable program by putting all the .o files together
  - Don't do this approach in 103, but it is approach 'Makefiles' use and the way most real programs are compiled

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)



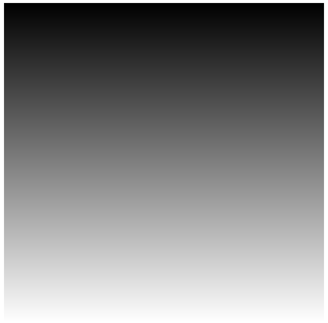
Scratch Workspace

- Identify patterns in indices of what you want to draw

	0	1	2	3	…	254	255
0							
1							
2							
3							
…							
254							
255							

Practice: Drawing

- Modify gradient.cpp to draw a black cross on a white background and save it as 'output1.bmp'
- Modify gradient.cpp to draw a black X down the diagonals on a white background and save it as 'output2.bmp'
- 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

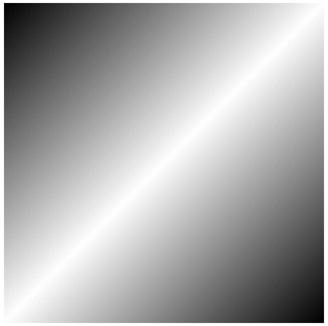
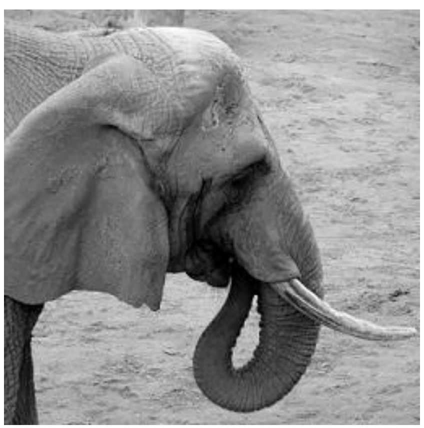


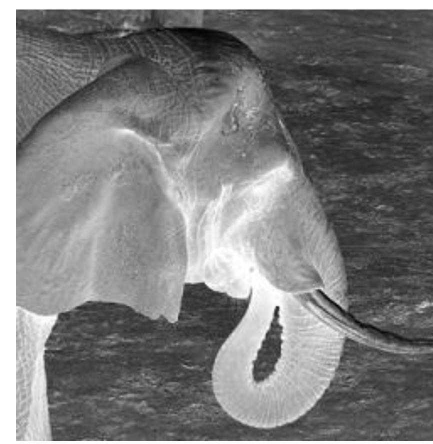
Image Processing

- Go to your gradient directory
  - $ wget http://bits.usc.edu/files/cs103/graphics/elephant.bmp
- Here is a first exercise…produce the "negative"



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] = 255-image[i][j];
         // invert color
      }
   }
   showGSBMP(image);
}
```

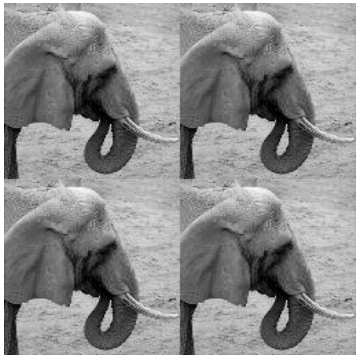


Inverted

Practice: Image Processing

- Perform a diagonal flip

- Tile



- Zoom

Selected Grayscale Solutions

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

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

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

Color Images

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

Original

Smoothed

Selected Color Solutions

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